Modular Probabilistic Models via Algebraic Effects

ANONYMOUS AUTHOR(S)

Probabilistic programming languages (PPLs) allow programmers to construct statistical models and then simulate data or perform inference over them. Many PPLs restrict models to a particular instance of simulation or inference, limiting their reusability. In other PPLs, models are not readily composable. Using Haskell as the host language, we present an embedded domain specific language based on algebraic effects, where probabilistic models are modular, first-class, and reusable for both simulation and inference. We also demonstrate how simulation and inference can be expressed naturally as composable program transformations using algebraic effect handlers.

Additional Key Words and Phrases: Probabilistic programming, algebraic effects, modularity

1 INTRODUCTION

In statistics, a probabilistic model captures a real-world phenomenon as a set of relationships between random variables: the model's parameters, inputs, and outputs. By integrating such notions into general-purpose languages, probabilistic programming languages (PPLs) allow programmers to build and execute probabilistic models. For example, consider a simple linear regression model that assumes a linear relationship between input variables x and output variables y; this can be represented using the standard mathematical notation shown on the left below. Using the language presented in this paper, the right-hand side shows how one could express the same model as a functional program.

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

```
linRegr x = do
  μ ← normal 0 3 #μ
  c ← normal 0 2 #c
  σ ← uniform 1 3 #σ
  y ← normal (μ * x + c, σ) #y
  return y
```

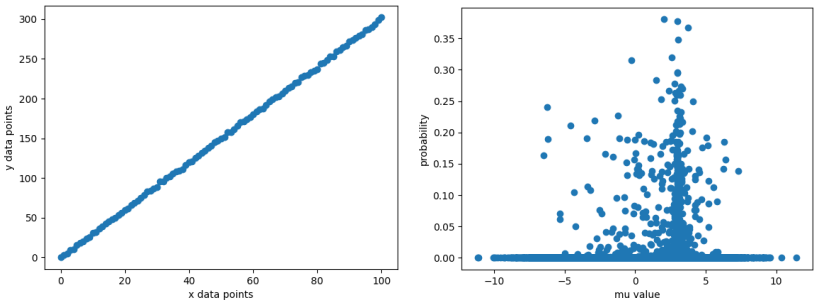
Both representations take an input x and specify the distributions which generate the model parameters μ, c, and σ; the output y is then generated from the normal distribution using mean $\mu * x + c$ and standard deviation σ. In the program representation, each primitive distribution is associated with a corresponding "observable variable", indicated by the # syntax; this is an optional argument, and its purpose will become clear shortly.

Given a probabilistic model, the programmer or data scientist will typically want to use it in at least two different ways. *Simulation* involves providing fixed values for the model parameters and inputs, to generate the resulting model outputs. Conversely, *inference* generally entails providing observed values for the model outputs and inputs, in an attempt to learn the model parameters.

For example, we might simulate from linRegr in our language as follows:

```
let xs  = [ 0 .. 100 ]
    env = (#μ := [3]) • (#c := [0]) • (#σ := [1]) • (#y := [ ]) • nil
in  map (simulate linRegr env) xs
```

First we declare a list of model inputs xs from 0 to 100. Then we define a "model environment" env which assigns values 3, 0, and 1 to observable variables #μ, #c, and #σ. This expresses our intention to *observe* parameters μ, c, and σ — that is, to provide external data 3 as the value of random variable μ whilst conditioning on the likelihood that $\mu = 3$, and similarly for c and σ. On the other hand no values are specified for #y in env, expressing our intent to *sample* the model output y — that is, to draw a value from its probability distribution. We then use library function simulate to simulate a



(a) Simulation (b) Inference (Likelihood Weighting)

Fig. 1. Visualising Linear Regression

single output from the model for each data point in xs using the specified environment, producing the result visualised in Fig. 1a.

Alternatively, we can perform inference on linRegr, for example using the Likelihood Weighting algorithm [van de Meent et al. 2018], as follows:

```
let xs  = [ 0 .. 100 ]
    xys = [(x, env) | x ← xs, let env = (#μ := []) • (#c := []) • (#σ := []) • (#y := [3 ∗ x]) • nil ]
in  map (lw 200 linRegr) xys
```

Here we define xys to pair each model input x with a model environment env that assigns the value $3 * x$ to #y but nothing to #μ, #c, and #σ. This expresses our intention to *observe* y but *sample* μ, c and σ. We then use library function lw to perform 200 iterations of Likelihood Weighting for each pair of model input and environment, producing a trace of weighted parameters μ, c, and σ whose distributions express the most likely parameter values to give rise to y. Fig. 1b visualises the likelihoods of samples for μ, where values around $\mu = 3$ clearly accumulate higher probabilities.

We refer to a model that can be used for both simulation and inference — where random variables can be switched between sample and observe modes without altering the model itself — as a *multimodal model*. While multimodal models have a clear benefit, letting the same model be interacted with for a variety of applications, few existing PPLs support them. Most frameworks, such as MonadBayes [Ścibior et al. 2018] and Anglican [Tolpin et al. 2016], instead require programmers to express models in terms of explicit sample and observe operations, which considerably limits their reusability. If the user wishes to interact with the "same" model in a new way, they have little choice but to reimplement it with a different configuration of sample and observe operations.

Indeed, the number of possible model interpretations extends far beyond the two general scenarios of simulation and inference, potentially including *any* combination of sample and observe operations that can be instantiated for a model's random variables. Depending on available data and uncertainty about the model, it is common to explore the model's output space by partially providing model parameters and randomly sampling the rest [Kline and Tamer 2016], or to alternate between which observable variables are being conditioned on [Moon 1996]. Ideally, all of these possible scenarios would be expressible with a single multimodal model definition, avoiding the need to define and separately maintain a different version of the model for each use case.

While some PPLs *do* support multimodal models, it is usually difficult or impossible to reuse existing models when creating new ones. Stan [Carpenter et al. 2017] and WinBUGS [Lunn et al. 2000] provide a bespoke language construct for models with its own distinctive semantics, but as well as lacking high-level programming features beyond those essential to model specification, model definitions are unable to reuse other model definitions. Languages like Turing [Ge et al. 2018] and Gen [Cusumano-Towner et al. 2019] take a different approach, supporting multimodal

models as macros that are compiled into functions; although they provide some support for compositionality, neither supports models as first-class values. These modularity limitations are especially significant for hierarchical modelling, where the goal is to explicitly define a composite model with independently defined sub-models [Gelman and Hill 2006].

In this paper, we present *Wasabaye*: a deeply embedded PPL in Haskell where probabilistic models are modular, first-class, and multimodal. Our solution uses algebraic effects [Plotkin and Power 2003] and handlers [Plotkin and Pretnar 2013], allowing models to be captured as syntax, and their semantics deferred to a choice of "model environment". By embedding into a functional language, models can then naturally exist as (first-class) functions and leverage all the abstractions and features of the host [Elliott et al. 2003; Gibbons and Wu 2014].

Our approach uses two key type abstractions: polymorphic sums for expressing effect signatures, and extensible records for model environments. Both of these can in fact be subsumed by row polymorphism [Leijen 2005], and so any language with support for polymorphic rows, such as PureScript [Freeman 2017], OCaml [Leroy et al. 2020] or Links [Hillerström and Lindley 2018], or with a type system powerful enough to express something similar (such as Haskell), should be capable of capturing Wasabaye's main features. The other type-level devices we use are ergonomic choices specific to Haskell, and are inessential to the main goals.

We begin by giving the necessary background and language overview in §2. Our contributions are then as follows:

- We demonstrate the features of our language via a realistic case study with real-world applications: the spread of disease during an epidemic (§3).
- We present an embedding technique that is novel in using algebraic effects to represent probabilistic models (§4), demonstrated with Haskell as the host language. To the best of our knowledge, ours is the first PPL to support models that are both multimodal and first-class.
- We provide a modular, type-safe mechanism for associating observed data to the random variables of a model, determining whether probabilistic operations should be interpreted as `sample` or `observe` (§5). The same mechanism is used to trace samples for plotting or debugging.
- We present a new approach to the compositional implementation of simulation and inference, using effect handlers to perform modular program transformations on models (§6). We illustrate the approach using Likelihood Weighting [van de Meent et al. 2018] and Metropolis Hastings [Wingate et al. 2011].
- We evaluate our language empirically, considering performance against two state-of-the-art PPLs, and language features supported across a range of modern PPLs (§7).

We discuss related work in more detail in §8. However, this is not the first time PPLs have been explored with Haskell as a host language. Erwig and Kollmansberger [2006] implement a probability monad for representing distributions in functional languages; Narayanan et al. [2016] use a tagless-final embedding [Kiselyov 2010] to encode inference algorithms as type class instances; Ścibior et al. [2018] use monad transformers [Liang et al. 1995] to demonstrate inference as effect composition. Our approach builds on the techniques offered by algebraic effects and extensible data.

The high-level notion of using interpreters to execute the effects of probabilistic models is an established technique in PPLs, and is similar in spirit to algebraic effects. Many PPLs accomplish this through context managers, coroutines, and continuation-passing style transformations [Bingham et al. 2019; Goodman and Stuhlmüller 2014; Tolpin et al. 2016]. However, these approaches fail to delineate between syntax and semantics, preventing models from being interpreted in a fully multimodal fashion. Moreover, the effect-interpreting mechanisms typically operate in weakly-typed, imperative settings, where effects are not associated with types and can occur unrestrictedly in a program. Algebraic effects have the potential to bring a type-safe, compositional discipline to

probabilistic programming. There is, however, little existing work in this area, and the topic has primarily remained a point of discussion [Moore and Gorinova 2018; Ścibior and Kammar 2015]. We present a novel design and implementation at the intersection of PPLs and algebraic effects.

In addition to the examples in this paper, our embedding has been tested with a range of models implemented in other PPLs, as well as well-known models such as those designed by Gelman and Hill [2006]; the full source code is freely available online.[1]

2 BACKGROUND AND LANGUAGE OVERVIEW

A probabilistic model, expressed using the $\sim$ notation introduced in § 1, describes how a set of random variables are distributed relative to some fixed input. If the model does not condition against any external data, the distribution it describes is the so-called *joint probability distribution*, giving the probabilities of all possible values that its random variables can assume. For example, the linear regression model in § 1 describes the distribution $\mathbb{P}(y, \mu, c, \sigma; x)$ – namely, the joint distribution over random variables y, μ, c, and σ given fixed input x as a non-random parameter.

In real-world applications, we typically have known values for only a subset of these random variables, and are interested in how the other variables are distributed with respect to those known values. Consider providing known data $\hat{y}$ for random variable y in linear regression; we say that we *condition on* y having *observed* $\hat{y}$. Using the well-known chain rule for two random variables:

$$\mathbb{P}(X, Y) = \mathbb{P}(X \mid Y) \cdot \mathbb{P}(Y)$$

we can derive the resulting distribution as the product $\mathbb{P}(\mu, c, \sigma \mid y = \hat{y}; x) \cdot \mathbb{P}(y = \hat{y})$. The first component, called the *conditional distribution*, describes the probabilities of values for each random variable μ, c, σ given $y = \hat{y}$ and some input x; the second component, called the *prior distribution*, gives the probability that y has value $\hat{y}$. Providing observed data to a probabilistic model can therefore be seen as specialising its joint distribution to some product of a conditional that is favourable to modelling and an associated prior.

2.1 Multimodal Models

The chain rule is a powerful tool that allows us to describe a jointly occurring set of events in terms of the variables we will provide data for, and then compute other variables of interest with respect to this; and there are of course as many ways to decompose a joint distribution as there are combinations of variables that can be conditioned on. Since statisticians often have a clear understanding of the variables they wish to learn and those they wish to condition against, models are in practice often specialised to specific conditional distributions (through the chain rule) and expressed as low-level algorithms that explicitly perform sampling and conditioning, such as in Ding et al. [2019]; Polson et al. [2013].

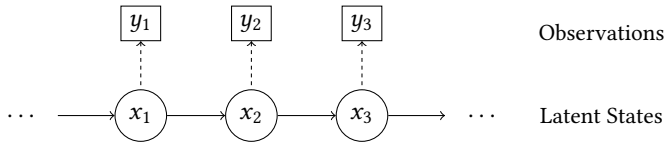
Most PPLs, such as WebPPL [Goodman and Stuhlmüller 2014] and Anglican [Tolpin et al. 2016], are then designed to support the direct translation of these low-level model specifications from paper to program via the operations `sample` and `observe`. These languages are useful for creating model instances tailored to specific situations, but the resulting models are not easy to experiment with. Tasks which should be straightforward, such as exploring random variable behaviours by isolating which ones are sampled from [Idreos et al. 2015] or selectively optimising model parameters [Yekutieli 2012], require alternative specialisations to be created by hand.

2.1.1 Multimodal models via model environments. With multimodal PPLs, the programmer specifies a single model which can be used to generate multiple specialisations, representing specific conditional distributions. Such languages require a mechanism for specifying observed data to random variables, determining whether they are to be sampled or observed. For example, Turing.jl

[1] https://github.com/min-nguyen/wasabaye

lets users choose whether to provide observed values as arguments when invoking a model, with omitting an argument defaulting to sampling [Ge et al. 2018]; in Pyro, users specify mappings between random variables and observed data via *context managers* that later constrain the values of runtime sampling operations [Bingham et al. 2019]. However, these solutions are dynamically typed with no guarantee that the named variables exist or are provided values of the correct type.

Our language supports multimodal models through a novel notion of *model environment*, which we explain in the context of a Hidden Markov Model (HMM) [Rabiner and Juang 1986]:



The idea of a HMM is that we have a series of latent states x_i which are related in some way to observations y_i. The HMM is then defined by two sub-models: a transition model ($\rightarrow$) that determines how latent states x_i are transitioned between, and an observation model ($\upuparrows$) that determines how x_i is projected to an observation y_i. The objective is to learn about x_i given y_i.

A simple HMM expressed in typical statistical pseudocode is shown in Fig. 2a; we describe its corresponding implemention in our language in Fig. 2b:

```
hmm(n, x_0)
  Δ_x ~ Uniform(0, 1)        } Model parameters
  Δ_y ~ Uniform(0, 1)        }
  for i = 1...n:
    δx_i ~ Bernoulli(Δ_x)    } Transition model
    x_i  = x_{i-1} + δx_i    }
    y_i  ~ Binomial(x_i, Δ_y) } Observation model
  return x_n
```

(a) Statistical pseudocode

```
hmm :: (Observables env '["y_i"] Int
       , Observables env '["Δ_x", "Δ_y"] Double)
    ⇒ Int → Int → Model env es Int
hmm n x0 = do
  Δx ← uniform 0 1 #Δx
  Δy ← uniform 0 1 #Δy
  let loop i x_{i-1} | i < n = do
                         δx_i ← bernoulli' Δx
                         let x_i = x_{i-1} + δx_i
                         y_i ← binomial x_i Δy #y_i
                         loop (i + 1) x_i
                     | otherwise = return x_{i-1}
  loop 0 x0
```

(b) Implementation

Fig. 2. Hidden Markov Model

The type of hmm says it is a function that takes two Ints as input and returns a Model env es Int, where env is the model environment, es is the effects which the model can invoke (detailed later in §4), and Int is the type of values the model generates. The constraint Observables states that $\#y_i$ :: Int, $\#\Delta_x$ :: Double, and $\#\Delta_y$ :: Double, are *observable variables* in the model environment env which may be conditioned on later when the model is used.

The function hmm takes the HMM length n and initial latent state x_0 as inputs, and specifies the transition and observation parameters Δ_x and Δ_y to be distributed uniformly. It then iterates over the n nodes, applying the transition and observation models at each step. The transition model computes latent state x_i from state x_{i-1} by adding a value δx_i generated from a Bernoulli distribution bernoulli' Δ_x. The observation model generates observation y_i from x_i via the distribution binomial x_i Δ_y. The final latent state is returned at the end.

The hash syntax # is how the programmer associates variables in the Observables env constraint with specific primitive distributions. They do this to indicate that they may later be interested in providing observed values for these variables to condition on. When they execute a model, they must provide a concrete environment of type env, and the presence or absence of observed values in that environment (for a given variable x) will determine whether the distribution tagged with $\#x$ is to be interpreted as observe or sample. The distribution bernoulli′ Δ_x has no observable variable, indicating that it is not possible to condition on δx_i; this makes sense because values of δx_i are latent, and so it is unlikely that we would ever want to provide data for them. (Primitive distributions like bernoulli come in primed variants that are always interpreted as sample.)

A model can then be interpreted as any of its conditioned forms by specifying an appropriate model environment. In our example, the HMM in its unspecialised form represents the joint distribution over its latent states x_i, observations y_i, and parameters Δ_x, Δ_y (given fixed input x_0 as the first latent state):

$$\mathbb{P}(x_1 \ldots x_n,\ y_1 \ldots y_n,\ \Delta_x,\ \Delta_y;\ x_0)$$

and we can then simulate the HMM (with length $n = 10$ and initial state $x_0 = 0$) by providing values for $\#\Delta_x$ and $\#\Delta_y$ in an environment env:

```
let x₀ = 0; n = 10;
    env = (#Δx := [0.5]) • (#Δy := [0.8]) • (#yi := []) • nil
in  simulate (hmm n) env x₀
```

This indicates that we want to observe 0.5 and 0.8 for Δ_x and Δ_y, and sample for each occurrence of y_i (because we provided no values for $\#y_i$); there are multiple occurrences of y_i at runtime, one for each $i \in \{1 \ldots n\}$, thanks to the iterative structure of the HMM. By the chain rule, the probability density this expresses is:

$$\mathbb{P}(x_1 \ldots x_{10},\ y_1 \ldots y_{10} \mid \Delta_x = 0.5,\ \Delta_y = 0.8;\ x_0 = 0) \cdot \mathbb{P}(\Delta_x = 0.5) \cdot \mathbb{P}(\Delta_y = 0.8)$$

In the case of inference, on the other hand, we provide an observation for each y_i and try to learn Δ_x and Δ_x. This is why (as the reader may already have noticed) a model environment provides a *list* of values for each observable variable, allowing for the situation where the observable variable has multiple dynamic occurrences. In this case we provide 10 observations, one for each $i \in \{1 \ldots n\}$:

```
let x₀ = 0; n = 10;
    env = (#Δx := []) • (#Δy := []) • (#yi := [0, 1, 1, 3, 4, 5, 5, 5, 6, 5]) • nil
in  lw 100 (hmm n) (x₀, env)
```

At runtime, the values associated with $\#y_i$ in the model environment are used to condition against the occurrences of $\#y_i$ that arise during execution, in the order in which they arise. By the chain rule, the probability density expressed by instantiating the model with this environment is:

$$\mathbb{P}(\Delta_x,\ \Delta_y,\ x_1 \ldots x_{10} \mid\ y_1 = 0 \ldots y_{10} = 5;\ x_0 = 0) \cdot \mathbb{P}(y_1 = 0) \cdot \ldots \cdot \mathbb{P}(y_{10} = 5)$$

Although the type system ensures that model environments map observable variables to values of an appropriate type, it does not constrain the number of values that are provided. Should observed values run out for a particular variable, any remaining runtime occurrences of the variable will default to sample; any surplus of values is ignored. While this flexibility could certainly obscure programming errors, other PPLs (such as Turing.jl [Ge et al. 2018]) take a similar approach, and we also note that the correctness of inference is unaffected. We consider alternative designs in §8.2.

2.2 Modular, First-Class Models

Fig. 2a used a single procedure, written in statistical pseudocode, to express a Hidden Markov Model. Such notations are understood by most mathematicians and are widely used in statistical journals. Even when the model is complex, a monolithic style of presentation prevails, where models are defined from scratch each time rather than built out of reusable components. The design of PPLs such as Stan [Carpenter et al. 2017], PyMC3 [Salvatier et al. 2016] and Bugs [Lunn et al. 2000] reflect these non-modular conventions.

Programmers, on the other hand, recognise the importance of modularity to maintainability and reusability: they expect to be able to decompose models into meaningful parts. Fig. 3a shows how the programmer may imagine the same HMM as a composition of parts; our language can then support this treatment of models in Fig. 3b:

```
transModel(Δ_x, x_{i−1})
  δx_i ~ Bernoulli(Δ_x)
  return x_{i−1} + δx_i

obsModel(Δ_y, x_i)
  y_i ~ Binomial(x_i, Δ_y)
  return y_i

hmmNode(Δ_x, Δ_y, x_{i−1})
  x_i ~ transModel(Δ_x, x_{i−1})
  obsModel(Δ_y, x_i)
  return x_i

hmm(n, x_0)
  Δ_x ~ Uniform(0, 1)
  Δ_y ~ Uniform(0, 1)
  for i = 1...n;
    x_i ~ hmmNode(Δ_x, Δ_y, x_{i−1})
  return x_n
```

(a) Statistical Pseudocode

```
transModel :: Double → Int → Model env es Int
transModel Δ_x x_{i−1} = do
  δx_i ← bernoulli' Δ_x
  return x_{i−1} + δx_i

obsModel :: (Observables env '["y_i"] Int)
         ⇒ Double → Int → Model env es Int
obsModel Δ_y x_i = do
  y_i ← binomial x_i Δ_y #y_i
  return y_i

hmmNode :: (Observables env '["y_i"] Int)
        ⇒ Double → Double → Int → Model env es Int
hmmNode Δ_x Δ_y x_{i−1} = do
  x_i ← transModel Δ_x x_{i−1}
  obsModel Δ_y x_i
  return x_i

hmm :: (Observables env '["y_i"] Int, Observables env '["Δ_x","Δ_y"] Double)
    ⇒ Int → Int → Model env es Int
hmm n x_0 = do
  Δ_x ← uniform 0 1 #Δ_x
  Δ_y ← uniform 0 1 #Δ_y
  foldl (>=>) return (replicate n (hmmNode Δ_x Δ_y)) x_0
```

(b) Program Representation

Fig. 3. A Modular Hidden Markov Model

In Fig. 3b we define the transition and observation distributions as separate models transModel and obsModel. These are composed by hmmNode to define the behaviour of a single node, which is in turn used by hmm to create a chain of nodes of length n, using replicate and a fold of Kleisli composition (>=>) to propagate each node's output to the next one in the chain.

```
(>=>) :: (a → Model env es b) → (b → Model env es c) → (a → Model env es c)
```

The observable variables of hmm are now inherited from its sub-models: transModel has none (and so lacks an Observables constraint entirely), whereas obsModel declares $\#y_i$ as its only observable variable.

As well as improving reusability, compositionality also allows programmers to organise models around the structure of the problem domain. For example the functions in Fig. 3b correspond in a straightforward way to the abstract components of a HMM: transModel makes it obvious that latent state x_i depends only on the previous state x_{i-1} (and Δ_x), and obsModel that observation y_i depends only on the state x_i that produced it (and Δ_y).

3 MODULAR, MULTIMODAL MODELS: A CASE STUDY

Before we turn to the details of our embedding approach in §4, we present a case study demonstrating our support for modular, first-class, multimodal models. §3.1 introduces our running example, the SIR (Susceptible-Infected-Recovered) model for the spread of disease [Liang and Li 2021]. §3.2 uses the SIR model to show how our language supports higher-order models which can be easily extended and adapted. §3.3 shows how a multimodal model can be used for both simulation and inference in the same application to facilitate Bayesian bootstrapping.

3.1 The SIR Model

The SIR model predicts the spread of disease in a fixed population of size n partitioned into three groups: s for *susceptible to infection*, i for *infected*, or r for *recovered* (where $s + i + r = n$). The model tracks how s, i, and r vary over discrete time t measured in days. Because testing is both incomplete and unreliable, the true sir values for the population cannot be directly observed; however, we can observe the number of reported infections ξ. This problem is thus a good fit for a Hidden Markov Model (§2), where the sir values play the role of latent states of type Popl, and ξ as the observations of type Reported:

```
data Popl     = Popl { s :: Int, i :: Int, r :: Int }
type Reported = Int
```

We now show how our language can be used to implement the SIR model as a modular HMM, starting with the transition and observation models.

SIR transition model. The transition model describes how the sir values change over a single day; we model two specific dynamics. First, susceptible individuals s transition to infected i at a rate determined by the values of s and i and the contact rate β between the two groups. We use a binomial distribution to model each person in s having a $1 - e^{-\beta i/n}$ probability of becoming infected, and update s and i accordingly:

```
trans_si :: Double → Popl → Model env es Popl
trans_si β (Popl s i r) = do
  let n = s + i + r
  δsi ← binomial' s (1 − exp ((−β * i) / n))
  return (Popl (s − δsi) (i + δsi) r)
```

Second, infected individuals i transition to the recovered group r, where a fixed fraction γ of people will recover in a given day. Again we use a binomial to model each person in i as having a $1 - e^{-\gamma}$ probability of recovering, and use this to update i and r:

```
trans_ir :: Double → Popl → Model env es Popl
trans_ir γ (Popl s i r) = do
  δir ← binomial' i (1 − exp (−γ))
  return (Popl s (i − δir) (r + δir))
```

The overall transition model trans_{sir} is simply the sequential composition of trans_{si} and trans_{ir}. Given β and γ (aggregated into the type TransParams), trans_{sir} computes the changes from s to i and then i to r to yield the updated sir population over a single day:

```
data TransParams = TransParams { β :: Double, γ :: Double }

transsir :: TransParams → Popl → Model env es Popl
transsir (TransParams β γ) = transsi γ >=> transir β
```

SIR observation model. For the observation model, we assume that the reported infections ξ depends only on the number of infected individuals i, of which a fixed fraction ρ will be reported. We use the Poisson distribution to model reports occurring with a mean rate of $\rho * i$:

```
type ObsParams = Double

obssir :: Observables env '["ξ"] Int ⇒ ObsParams → Popl → Model env es Reported
obssir ρ (Popl _ i _) = do
  ξ ← poisson (ρ * i) #ξ
  return ξ
```

Since we intend this as the observation model, we declare observable variable $\#\xi :: \text{Int}$ in the Observables constraint and attach it to the Poisson distribution so we can condition on it later. (Binding the local name ξ and immediately returning it is technically redundant but emphasises the connection to the ~ notation used by statisticians.)

HMM for the SIR model. Now the transition and observation models can be combined into a HMM. We build on the modular design in Fig. 3, but go a step further by defining a HMM as a higher-order model that is parameterised by sub-models of type TransModel and ObsModel:

```
type TransModel env es ps lat     = ps → lat → Model env es lat
type ObsModel  env es ps lat obs = ps → lat → Model env es obs
```

Here ps represents the types of the model parameters, and lat and obs are the types of latent states and observations. The higher-order HMM is then defined as:

```
hmm :: Model env es ps1              → Model env es ps2
    → TransModel env es ps1 lat → ObsModel env es ps2 lat obs
    → Int → lat → Model env es lat
hmm transPrior obsPrior trans obs n x0 = do
  θ ← transPrior
  ϕ ← obsPrior
  hmmNode xi−1 = do xi ← trans θ xi−1
                    yi ← obs ϕ xi
                    return xi
  foldl (>=>) return (replicate n hmmNode) x0
```

The input models transPrior and obsPrior are first used to generate model parameters θ and ϕ, and trans and obs are arbitrary transition and observation models parameterised by θ and ϕ respectively. The last line creates a HMM of length n with initial state x_0.

Our SIR transition and observation parameters will be provided by models transPrior_{sir} and obsPrior_{sir} below, using primitive distributions gamma and beta; their observable variables $\#\beta$, $\#\gamma$, and $\#\rho$ will let us condition on those parameters later:

```
transPrior_sir :: (Observables env '["β", "γ"] Double)
    ⇒ Model env es TransParams
transPrior_sir = do
  β ← gamma 2 1 #β
  γ ← gamma 1 (1/8) #γ
  return (TransParams β γ)
```

```
obsPrior_sir :: (Observables env '["ρ"] Double)
    ⇒ Model env es ObsParams
obsPrior_sir = do
  ρ ← beta 2 7 #ρ
  return ρ
```

We can now define the complete SIR model. From an initial population sir of susceptible, infected, and recovered individuals, hmm_{sir} models the change in sir over n days given reported infections ξ:

```
hmm_sir :: (Observables env '["ξ"] Int, Observables env '["β", "γ", "ρ"] Double)
    ⇒ Int → Popl → Model env es Popl
hmm_sir = hmm transPrior_sir obsPrior_sir trans_sir obs_sir
```

We can simulate over this model, perhaps to explore some expected model behaviours, by specifying an input model environment sim_env_{in} of type Env SIRenv that provides specific values for $\#\beta$, $\#\gamma$, and $\#\rho$, but provides no values for reported infections $\#\xi$ (ensuring that we always sample for ξ). Applying simulate to hmm_{sir} 100 and input population sir_0 simulates the spread of the disease over 100 days.

```
type SIRenv = '[ "β" := Double, "γ" := Double, "ρ" := Double, "ξ" := Int ]
simulateSIR :: IO (Popl, Env SIRenv)
simulateSIR = do
  let sim_env_in = (#β := [0.7]) • (#γ := [0.009]) • (#ρ := [0.3]) • (#ξ := []) • nil
      sir_0      = Popl { s = 762, i = 1, r = 0 }
  simulate (hmm_sir 100) sim_env_in sir_0
```

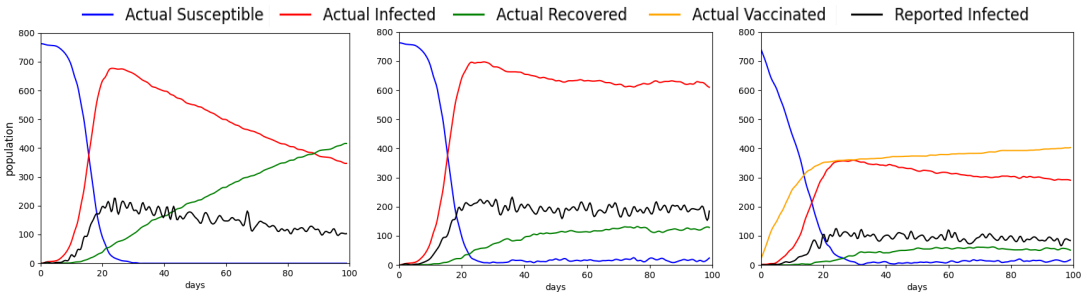
This returns the final population sir_{100} plus an *output* model environment sim_env_{out} mapping each observable variable to the values sampled for that variable during simulation. From this we can extract the reported infections ξs:

```
do (sir_100 :: Popl, sim_env_out :: Env SIRenv) ← simulateSIR
   let ξs :: [Reported] = get #ξ sim_env_out
   ...
```

Fig. 4a shows a plot of these ξ values and their corresponding latent (population) states; but note that as it stands, the model provides no external access to the latent states shown in the plot (except the final one sir_{100}). To generate this kind of output, it would be ideal if we could access those latent states without polluting the type signature of our HMM abstraction, which concisely corresponds to our statistical definition of a HMM. Fortunately, the setting of algebraic effects means users can orthogonally extend a model with a desired *effect*, such as State, Reader, or in this case a Writer effect for writing the intermediate latent states to a stream; this process is straightforward and shown later in §5.5.

3.2 Modular Extensions to the SIR Model

Although the SIR model is simplistic, realistic models may be uneconomical to run or too specific to be useful. When designing models, statisticians aim to strike a balance between complexity and precision, and modular models make it easier to incrementally explore this trade-off. We support this claim by showing how two possible extensions of the SIR model are made easy in our language; while these are by no means the most modular solutions possible, they should suffice to make our point.



(a) SIR (b) with resusceptible (c) with resusceptible + vacc

Fig. 4. SIR Hidden Markov Model Simulation

Suppose our disease does not confer long-lasting immunity, so that recovered individuals r transition back to being susceptible s after a period of time [Shi et al. 2008]. We can model this with a new transition behaviour:

```
data TransParams = TransParams { β :: Double, γ :: Double, ρ :: Double, η :: Double }

trans_rs :: Double → Popl → Model env es Popl
trans_rs η (Popl s i r) = do
  δrs ← binomial r (1 − exp (−η))
  return (Popl (s + δrs) i (r − δrs))

trans_sir :: TransModel env es TransParams Popl
trans_sir (TransParams β γ η) =
  trans_si β >=> trans_ir γ >=> trans_rs η
```

We need only modify TransParams to include a new parameter η; define a new transition sub-model trans_{rs} (parameterised by η) that stochastically moves individuals from recovered r to susceptible s; and then adjust trans_{sir} to compose trans_{rs} with our existing transition behaviours. A simulation of the resulting system is shown in Fig. 4b.

Now consider adding a variant where susceptible individuals s can become vaccinated v [Ameen et al. 2020]. This involves adding a new sub-population v to the latent state:

```
data Popl = Popl { s :: Int, i :: Int, r :: Int, v :: Int }
data TransParams = TransParams { β :: Double, γ :: Double, ρ :: Double, η :: Double, ω :: Double }

trans_sv :: Double → Popl → Model env es Popl
trans_sv ω (Popl s i r v) = do
  δsv ← binomial s (1 − exp (−ω))
  return (Popl (s − δsv) i r (v + δsv))

trans_sir :: TransModel env es TransParams Popl
trans_sir (TransParams β γ η ω) =
  trans_si β >=> trans_ir γ >=>
  trans_rs η >=> trans_sv ω
```

We add field v to Popl representing vaccinated individuals, and add ω to TransParams, determining the rate at which s individuals transition to v. The new behaviour is expressed by trans_{sv} and then composed into a new transition model trans_{sir}. A simulation of this is shown in Fig. 4c.

3.3 Exploring Multimodality in the SIR Model

We now show how our support for higher-order, modular models is complemented by the flexibility of multimodal models. Suppose the goal were to infer SIR model parameters β, γ, ρ given data on reported infections ξ. Ideally we would have a real dataset of reported infections to condition on. But what if our dataset were sparse, or if we were interested in quick hypothesis testing? A common option is to use simulated data as observed data, a method called Bayesian bootstrapping [Fushiki 2010]. This task is made simple with multimodal models, because we can take the outputs from simulation over a model and plug them into an environment that specifies inference over the same model:

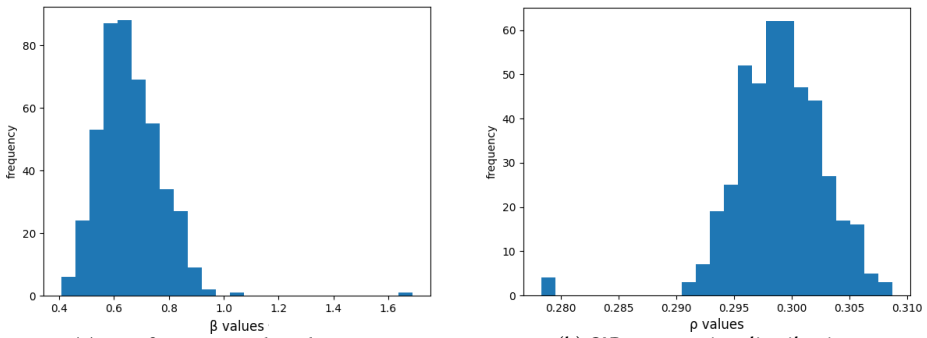
```
inferSIR :: Env SIRenv → IO (Env SIRenv)
inferSIR sim_env_out = do
  let ξs          = get #ξ sim_env_out
      mh_env_in = (#β := []) • (#γ := [0.0085]) • (#ρ := []) • (#ξ := ξs) • nil
      sir_0         = Popl { s = 762, i = 1, r = 0 }
  mh 50000 (hmm_sir 100) (sir_0, mh_env_in)
```

Here we take the output model environment sim_env_{out} produced by simulateSIR, and use its ξ values to define an input model environment mh_env_{in} that conditions against $\#\xi$. Moreover, suppose we already have some confidence about a particular model parameter, such as the recovery rate γ; for efficiency, we can avoid inference on γ by setting $\#\gamma$ to an estimate 0.0085 and sampling only for the remaining parameters $\#\beta$ and $\#\rho$. We then run Metropolis Hastings [Wingate et al. 2011] for 50,000 iterations, which returns an output environment (also of type Env SIRenv) containing the values sampled from all iterations.

The inferred posterior distributions for β and ρ can then be obtained simply by extracting them from that environment:

```
do mh_env_out ← inferSIR ξs
  let βs = get #β mh_env_out
      ρs = get #ρ mh_env_out
  ...
```

These are visualised in Fig. 5. Values around $\beta = 0.7$ and $\rho = 0.3$ occur more frequently, because these were the parameter values provided in simulateSIR in §3.1.



(a) SIR β posterior distribution (b) SIR ρ posterior distribution

Fig. 5. SIR Inference (Metropolis Hastings)

4 A SYNTACTIC EMBEDDING OF MULTIMODAL MODELS

To support multimodality, a key requirement is that models have a purely syntactic representation which can be assigned a semantics at a later stage. This will allow us to defer the interpretation of primitive distributions as either sample or observe operations until we know the variables we want to condition on. An embedding technique that supports this is an *algebraic effect* embedding [Plotkin and Power 2003]. In this approach, an effectful program which computes a value of type a has the following type:

```
prog :: Prog es a
```

The parameter *es* is called an *effect signature*, in our embedding represented as a type-level list, containing the computational effects that the program may perform. Such programs are syntax

trees whose nodes contain operation calls belonging to effect types $e \in es$ and whose leaves contain pure values of type a.

Programs in this form can be interpreted by *algebraic effect handlers* [Plotkin and Pretnar 2013] which handle specific effects in the program:

```
handle_e :: Prog (e : es) a → Prog es b
```

Such a handler assigns partial meaning to a program by interpreting all operations op of type e, discharging e from the effect signature, and transforming the return value of type a into a value of type b. Effect handlers are modular building blocks that can be composed into various program interpretations.

When we use this approach to represent a probabilistic computation, nodes of the syntax tree will contain calls to probabilistic operations, and inference and simulation will be implemented as effect handlers that interpret those operation calls. Mechanisms (such as sample tracing or particle simulation) specific to particular simulation or inference algorithms can be supported via effects that inject additional computational structure into the model, such as statefulness or non-determinism.

In this rest of this section (§4), we describe our syntactic representation of multimodal models; providing semantics to this embedding via effect handlers will follow in §5.

4.1 An Infrastructure for Algebraic Effects

There are many ways to represent Prog es a, programs indexed by arbitrary effects; see Hillerström [2015] for a useful summary. Our embedding uses a simple version of the *freer* monad [Kiselyov and Ishii 2015]:

```
data Prog es a where
  Val :: a → Prog es a
  Op  :: forall x. EffectSum es x → (x → Prog es a) → Prog es a
```

Here, a term of type Prog es a is a syntax tree whose leaves, Val x, contain a pure value x of type a, and whose nodes, Op op k, contain an *operation* op of the abstract datatype EffectSum es x:

```
data EffectSum (es :: [Type → Type]) (x :: Type) where ...
```

A value of type EffectSum es x represents a single operation of type e x, where e is some effect type constructor that occurs in the type-level list es, and x is the type of value the operation produces. The freer monad thus supports multiple effects directly via the constructor Op, avoiding the need for explicit coproducts (cf. Swierstra [2008]). Effect sums cannot be constructed directly; their implementation is kept abstract, and the following type class, Member, is instead provided for working with them:

```
class FindElem e es ⇒ Member e es where
  inj :: e x → EffectSum es x
  prj :: EffectSum es x → Maybe (e x)
```

The constraint Member e es asserts that if we can determine e's position in es via the type class FindElem e es (omitted), then we can safely inject and project an effectful operation, of type e x, into and out of EffectSum es x.

The second argument of the Op constructor is a continuation k of type x → Prog es a that takes the result of the operation and constructs the remainder of the syntax tree. This encoding means effect types e need not be functors, in constrast to approaches based on the free monad. In turn this means the type of distributions (§4.2.1 below) can be expressed as a GADT, and is in part why we

prefer Kiselyov and Ishii's approach to alternatives such as Wu and Schrijvers [2015] and Kiselyov et al. [2013]. Other details of their design specific to performance are omitted for simplicity.

4.2 Multimodal Models as Effectful Programs

We now define a probabilistic model, or simply *model*, to be an effectful program of type Prog es a where es includes at least two specific effects: Dist and ObsReader env. The *distribution* effect Dist allows the model to make use of primitive distributions such as normal and uniform distributions; the *observable-reader* effect ObsReader env allows the model to read and update the values of observable variables in a model environment of type env.

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

The Member constraints specify that es contains Dist and ObsReader env. While these two effects suffice for model specification, others may be useful for model execution, and this type allows the model to remain polymorphic in any such additional effects.

4.2.1 Effects for distributions. The core computational effect of a probabilistic model is the distribution effect Dist, allowing models to be formulated in terms of primitive probability distributions. The constructors represent the operations of the effect type, and thus correspond to various primitive distributions. We present a representative selection below:

```
data Dist a where
  Normal    :: Double → Double → Maybe Double → Dist Double
  Uniform   :: Double → Double → Maybe Double → Dist Double
  Bernoulli :: Double → Maybe Bool → Dist Bool
  Discrete  :: [(a, Double)] → Maybe a → Dist a
  ...
```

Every use of a primitive distribution must, at runtime, be interpretable as either sample or observe, depending on the availability of an observed value. Each Dist operation therefore takes an additional parameter of type Maybe a (where a is the base type of the distribution) indicating the presence or absence of a value to condition on. This detail is hidden from the user, but is used internally to determine how operation calls are to be interpreted, as we discuss next.

4.2.2 Effects for reading observable variables. To support conditioning on observable variables via model environments, we require an *observable-reader* effect, ObsReader env, with an operation that can read and update values of observable variables in env. This is similar in spirit to the well-known Reader type.

```
data ObsReader env a where
  Ask :: Observable env x a ⇒ ObsVar x → ObsReader env (Maybe a)
```

The single operation Ask of ObsReader env has the constraint Observable env x a. This associates a *list* of values of type a with observable variable x in any environment of type env, supporting conditioning on multiple dynamic instances of the same observable variable. The ObsVar x argument represents the observable variable name (specified using #). The result is then of type Maybe a indicating the presence or absence of a value to condition on, suitable for passing directly to one of the Dist constructors above. We defer the concrete implementations of these types to §5.1.

However, the user is not expected to specify ObsReader effects directly. Rather, we provide an interface of *smart constructors* [Swierstra 2008] which manage the requests to read from observable variables. For example:

```
normal :: (Observable env x Double) ⇒ Double → Double → ObsVar x → Model env es Double
normal mu sigma x′ = Model (do maybe_v ← call (Ask x′); call (Normal mu sigma maybe_v))
```

Recall that, in addition to the usual normal distribution parameters mu and sigma, the constructor Normal of Dist also expects an argument of type Maybe Double representing the presence or absence of an observed value. The smart constructor normal has a similar signature, but with an observable variable name x′ in place of the Maybe Double, thereby associating the distribution with a random variable (rather than particular value) to condition on. The role of the smart constructor is to insert an Ask operation into the program which retrieves the corresponding value maybe_v of type Maybe Double from the model environment, which is then used to call the Normal operation.

The helper function call simplifies the construction of operation calls, taking care of the injection into EffectSum es x and supplying the leaf continuation Val:

```
call :: Member e es ⇒ e x → Prog es x
call op = Op (inj op) Val
```

A primed variant of each smart constructor is also provided, using Nothing as the observed value, for the common case when a primitive distribution does not need to be conditioned on:

```
normal′ :: Double → Double → Model env es Double
normal′ mu sigma = Model (call (Normal mu sigma Nothing))
```

As an illustration of how the smart constructors work, consider the simple model in Fig. 6 which generates a bias p from a uniform distribution and uses it to parameterise a Bernoulli distribution, determining whether the outcome of a coin flip y is more likely to be heads (True) or tails (False). Fig. 6a shows how the user might write the program (omitting the type signature); Fig. 6b shows the equivalent program written without smart constructors.

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

(a) User code, with smart constructors

```
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
```

(b) Without smart constructors

Fig. 6. Behaviour of smart constructors

5 INTERPRETING MULTIMODAL MODELS

We now turn to using effect handlers to assign semantics to models. Interpreting a multimodal model has two stages: specialising the model into the conditional form determined by a model environment, and then executing the resulting specialised model using a particular simulation or inference algorithm.

The first of these stages is the focus of this section. We start with the implementation of model environments (§5.1), and then define effect handlers for reading observed values (§5.2) and interpreting primitive distributions in response to whether observed values have been provided (§5.3). Effect handlers for simulation and inference are the topic of §6.

5.1 Model Environments

Model environments allow the user to assign values to observable variables which the model can then read from. For the sake of compositionality, models should only need to mention the observable

variables they make use of, and be polymorphic in the rest. Moreover, a given observable variable may bind multiple successive values at runtime – one for each time the variable is evaluated.

These two design constraints suggest a representation of model environments as extensible records, where the fields are observable variable names and the values are *lists* of observed values, as expressed by the datatype Env:

```
data Env (env :: [Assign Symbol Type]) where
  ENil  :: Env '[]
  ECons :: [a] → Env env → Env ((x := a) : env)
data Assign x a = x := a
```

The type parameter env represents the type of the model environment as a type-level list of pairs Assign x a, associating type-level variable names x of kind Symbol with value types a of kind Type; this tracks the variables in the environment and their corresponding types. The constructor ENil is the empty environment, and the constructor ECons takes a list of values of type a and an environment of type env and prepends a new entry for x, producing an environment of type (x := a) : env.

The observable variable names in Env, being type-level strings of kind Symbol, have no value representation; to use them as record fields at the value-level, we give the singleton datatype ObsVar:

```
data ObsVar (x :: Symbol) where
  ObsVar :: KnownSymbol x ⇒ ObsVar x
```

This acts as a container for Symbols, storing them as a phantom parameter x. String values can be neatly promoted to such containers by deriving an instance of the IsLabel class, using Haskell's OverloadedLabels language extension:

```
instance (KnownSymbol x, x ~ x′) ⇒ IsLabel x (ObsVar x′) where
```

This enables values of type ObsVar to be created using the # syntax, so that for example value #foo has type ObsVar "foo". Model environments are then constructed using the following interface, which provides nil for the empty environment, and an infix cons-like operator (•) which makes use of the := notation at the value-level:

```
(•) :: Assign (ObsVar x) [a] → Env env → Env ((x := a) : env)
nil :: Env '[]
```

Finally, constraining a polymorphic model environment is done via the type class Observable, which provides type-safe access and updates to observable variables:

```
class (FindElem x env, LookupType x env ~ a) ⇒ Observable env x a
  get :: ObsVar x → Env env → [a]
  set :: ObsVar x → [a] → Env env → Env env
type family LookupType x env where
  LookupType x ((x := a) : env) = a
  LookupType x ((x′ := a) : env) = LookupType x env
```

The methods get and set use the type class FindElem (used in §4.1) to find the position of a variable x in env, and the type family LookupType to retrieve its type a. Many observable variables of the same type can be specified with the type family Observables env xs a below, which returns a nested tuple of constraints Observable env x a for each variable x in xs:

```
type family Observables env (xs :: [Symbol]) a :: Constraint where
  Observables env (x : xs) a = (Observable env x a, Observables env xs a)
  Observables env '[] v = ()
```

Although other designs are possible, using lists to represent observations for random variables is justified by the fact that, for correctness, general-purpose inference algorithms must compute the same distribution on traces (sequences of sampled or observed values) [Tolpin et al. 2016].

5.2 Handling Reading of Observable Variables

The first step in specialising a model to a particular model environment is to handle the Ask operations of ObsReader, representing environment read requests.

In the setting of algebraic effects, denoting that an effect e has been handled is done by *discharging* it from the front of an effect signature e : es. For this, we introduce a helper function discharge which pattern-matches an operation of type EffectSum (e : es) a to determine whether it inhabits the leftmost component e of the sum:

```
discharge :: EffectSum (e : es) a → Either (EffectSum es a) (e a)
```

If the operation indeed belongs to e, it is returned as Right op where op has type e a. Otherwise the operation belongs to an effect in es and it is returned as Left op where op has type EffectSum es a, discharging e from the effect signature.

We then define a handler which, given a model environment env and a program with effect signature (ObsReader env : es), discharges the ObsReader env effect by interpreting Ask operations:

```
handle_Read :: Env env → Prog (ObsReader env : es) a → Prog es a
handle_Read env (Op op k) = case discharge op of
  Right (Ask x) → let vs      = get x env
                      maybe_v = safeHead vs
                      env'    = set x (safeTail vs) env
                  in  handle_Read env' (k maybe_v)
  Left op'      → Op op' (handle_Read env . k)
handle_Read env (Val x)   = return x
```

There are three cases. On matching an operation as Right (Ask x) containing a request to read from x, the list of values vs associated with x is looked up in env. If vs has a head element, contained in maybe_v, that value becomes the current observation of x and is removed from env; this ensures that no observed value is conditioned on more than once during an execution, and that the order in which observations are consumed matches the execution order. Otherwise there is no observation of x and env is unchanged. The resulting maybe_v is provided to the continuation k to construct the remainder of the program, which is then recursively handled by handle$_{\text{Read}}$ using the updated environment.

If we match an operation as Left op′, then op′ does not belong to ObsReader. The operation is left intact and the remainder of the program is handled via (handle$_{\text{Read}}$ env . k). Lastly, reaching the non-operation Val x will simply return value x as the program's output.

In the handler definitions that follow, we omit the Left op′ and Val x clauses when their implementation follows the pattern above. For an overview of effect handlers we refer the reader to Kiselyov and Ishii [2015].

5.3 Handling Distributions

The second step of model specialisation is the handler for the Dist effect, which interprets a primitive distribution call as a sampling or observing operation depending on the presence or absence of an observed value. This requires two new effects, Sample and Observe:

```
data Sample a where
  Sample :: Dist a → Sample a

data Observe a where
  Observe :: Dist a → a → Observe a
```

Each has a single operation. Sample takes a distribution to sample from, whereas Observe takes a distribution and an observed value. The distribution handler is then given by:

```
handle_Dist :: (Member Sample es, Member Observe es) ⇒ Prog (Dist : es) a → Prog es a
handle_Dist (Op op k) = case discharge op of
  Right d → case getObs d of Just v  → (do x ← call (Observe d v)
                                           handle_Dist (k x))
                             Nothing → (do x ← call (Sample d)
                                           handle_Dist (k x))

getObs :: Dist a → Maybe a
```

The accessor function getObs retrieves the optional observation associated with a primitive distribution (§4.2.1). On encountering a distribution d, the handler uses getObs to try to retrieve its observed value; if there is such a value v, we call a corresponding Observe operation, and otherwise we call Sample.

5.4 Specialising Multimodal Models

Together, these two handlers specialise a multimodal model into the conditional form determined by a particular model environment. Their net effect on the type of the model is illustrated by the following composite handler:

```
handle_core :: (Member Observe es, Member Sample es)
            ⇒ Env env → Model env (ObsReader env : Dist : es) a → Prog es a
handle_core env = handle_Dist . (handle_Read env) . runModel
```

This handles both Dist and ObsReader, replacing all primitive distributions with explicit calls to Sample and Observe. As an example, consider the coinFlip model presented earlier in Fig. 6b. Initially applying $\text{handle}_{\text{Read}}$ using the example environment ((#p := [0.5]) • (#y := []) • nil) would produce coinFlip′ in Fig. 7a as an intermediate program, in which all distributions calls have been parameterised by either a concrete observed value or Nothing; the updated environment would be ((#p := []) • (#y := []) • nil) where #p is fully consumed and #y is unchanged. Then applying $\text{handle}_{\text{Dist}}$ to coinFlip′ would yield Fig. 7b; observed values in primitive distributions are retained, although they are rendered redundant by the information in the Observe constructor.

```
handle_Read ((#p :=[0.5]) • (#y :=[]) • nil) coinFlip
                          ⇓
coinFlip′ = do
  p ← call (Uniform 0 1 (Just 0.5))
  y ← call (Bernoulli p Nothing)
  return y
```

(a) Reading from the model environment

```
handle_Dist coinFlip′
          ⇓
coinFlip″ = do
  p ← call (Observe (Uniform 0 1 (Just 0.5)) 0.5)
  y ← call (Sample (Bernoulli p Nothing))
  return y
```

(b) Distributions interpreted to sample or observe

Fig. 7. coinFlip model: handling effects

This initial pipeline of effect handling turns a multimodal model into a form suitable for further specialisation by a simulation or inference algorithm, which we show in §6.

5.5 Extending Models with Additional Effects

When building models, users are not restricted to using only the two base effects Dist and ObsReader: the effect signature es in Model env es a can be easily extended with an arbitrary desired effect e, by first constraining the model with Member e es, and then handling e with a corresponding handler.

As an example, we revisit the SIR model in §3.1 where Fig. 4 plotted *all* intermediate values Popl of susceptible, infected, and recovered individuals over n days, despite our implementation hmm_{sir} only returning *final* one:

```
hmm_sir :: (Observables env '["ξ"] Int, Observables env '["β", "γ", "ρ"] Double)
        ⇒ Int → Popl → Model env es Popl
hmm_sir n = hmm transPrior_sir obsPrior_sir trans_sir obs_sir n
```

To record all *sir* values produced by hmm_{sir}, we introduce the constraint Member (Writer w) es to require es to contain the effect Writer w, representing computations that produce a stream of data of type w; here we choose w to be a list of *sir* values [Popl]. The transition model trans_{sir} can then use the Writer operation tell to concatenate each new *sir* value to the existing trace of values:

```
tell      :: Member (Writer w) es ⇒ w → Model env es ()
trans_sir :: Member (Writer [Popl]) es ⇒ TransModel env es TransParams Popl
trans_sir (TransParams β γ) sir = do
  sir' ← (trans_si β >=> trans_ir γ) sir
  tell [sir']
  return sir'
```

Models with user-specified effects can then be easily reduced into a form suitable for specialisation under a model environment (§5.4), by handling those effects beforehand with a suitable handler:

```
handle_Writer :: Monoid w ⇒ Model env (Writer w : es) a → Model env es (a, w)
hmm'_sir :: (Observables env '["ξ"] Int, Observables env '["β", "γ", "ρ"] Double)
         ⇒ Int → Popl → Model env es (Popl, [Popl])
hmm'_sir n = handle_Writer . hmm_sir n
```

Here, composing hmm_{sir} with $\mathsf{handle}_{\mathsf{Writer}}$ interprets the tell operations arising from the transition model, producing a new SIR model hmm'_{sir} that returns the trace of *sir* values as an additional output of type [Popl].

6 SIMULATION AND INFERENCE AS EFFECT HANDLERS

Simulation or inference over a model is expressed through the semantics we assign to its Sample and Observe effects; in principle we could therefore define model execution in terms of two handlers, one for each of those effects. However, many algorithmic approaches such as Monte-Carlo methods [Chong et al. 2010] often rely on common mechanisms, such as sample tracing, probability mapping, and model reparameterisation [Robert and Titterington 1998]. By identifying some of these mechanisms as basic building blocks, the possibility arises of composing them in novel ways, giving rise to many useful variants of algorithms [Ścibior et al. 2018]. This motivates a compositional approach in which aspects of model execution can be defined and extended modularly.

In this section, we use effect handlers to define a series of composable program transformations for probabilistic programs, which iteratively refines a model by installing new effects around existing ones. The result is an interpretation of the model in the context of specific algorithms for simulation in §6.1 and inference in §6.2.

6.1 Simulation as Effect Handlers

Simulation can be considered the most basic form of model execution. It runs the provided model as a generative process, using observed data from the model environment when available and otherwise drawing new samples. At the end, it returns a model output, plus a *sample trace* uniquely

identifying each *runtime* Sample operation with its sampled value; this becomes especially pertinent (§6.2.2) for the correctness and implementation of generic inference algorithms [Tolpin et al. 2016].

To support sample traces, each Sample occurrence thus needs a unique dynamic address α assigned; in our actual embedding, this feature is implemented by $\text{handle}_{\text{Dist}}$. For simplicity, we omit these low-level details, and assume operations Sample and Observe are now parameterised by an address α of abstract type Addr. The type of sample traces, STrace, is then a map from addresses α to values of abstract type PrimVal that primitive distributions can generate (concretely, PrimVal is an open sum, but one can avoid this using dependent maps):

```
type STrace = Map Addr PrimVal
```

Updating this sample trace is to be performed by the well-known State effect:

```
data State s a where
  Modify :: (s → s) → State s ()

handle_State :: s → Prog (State s : es) a → Prog es (a, s)
```

Its operation Modify takes a function of type s → s and applies this to state s. Its handler $\text{handle}_{\text{State}}$, given an initial state, additionally returns the final state that results from handling a program.

We implement the tracing of samples as the *program transformation* traceSamples below, which is a handler that installs a runtime State STrace effect after each Sample operation:

```
traceSamples :: (Member Sample es, Member (State STrace) es) ⇒ Prog es a → Prog es a
traceSamples (Op op k) = case prj op of
  Just (Sample d α) → Op op (λx → do call (Modify (Map.insert α x))
                                     traceSamples (k x))
```

Notice that we apply prj (§4.1) to op, rather than discharge; this is because we do not intend to handle any effects, and so we place no constraints on the order of the effect signature es. Instead, we leave the Sample operation unhandled as op in the program, and construct a new continuation: this takes the future output x from Sample and calls a Modify operation to store x at address α in the sample trace, before continuing with the original continuation k.

We can now define the handlers for Observe and Sample orthogonally from this transformation. For Observe operations, $\text{handle}_{\text{Obs}}$ performs no conditioning side-effects, and simply needs to return the observed value y to its continuation k:

```
handle_Obs :: Prog (Observe : es) a → Prog es a
handle_Obs (Op op k) = case discharge op of
  Right (Observe d y _) → handle_Obs (k y)
```

For Sample operations, $\text{handle}_{\text{Samp}}$ takes the provided distribution d and applies the function sampleIO, defined using an external statistics library; the generated value v is then passed to k:

```
handle_Samp :: Prog '[Sample] a → IO (a, STrace)
handle_Samp (Op op k) = case discharge op of
  Right (Sample d _) → do v ← sampleIO d
                          handle_Samp (k v)

sampleIO :: Dist a → IO a
```

Running this dispatches the final effect in Prog to produce an IO effect, hence it is always executed as the last handler where only Sample operations can occur in the program.

The complete definition for simulation is then given by the handler composition runSimulate:

```
runSimulate :: es ~ '[ObsReader env, Dist, State STrace, Observe, Sample]
            ⇒ Env env → Model env es a → IO (a, STrace)
runSimulate env = handle_Samp . handle_Obs . (handle_State Map.empty) . traceSamples . (handle_core env)
```

Above depicts how the logic of model execution can be decomposed into a modular and transparent system. The concrete effects in es, specified by the type coercion (~), are kept abstract to the user interested in simply building and using models; simultaneously, the compositional nature of handlers makes it easy for programmers who wish to implement and extend new forms of model execution, as demonstrated next in §6.2.

As a last remark, the reader may notice that the top-level function simulate (as seen in §3.1), which uses runSimulate, will differ slightly in its type signature, which we give now:

```
simulate :: es ~ '[ObsReader env, Dist, State STrace, Observe, Sample]
         ⇒ (x → Model env es a) → Env env → x → IO (a, Env env)
```

This allows for better composition when simulating a model over many inputs x. Then for type-safe user-access to an STrace structure, this is reified into an *output* environment of type Env env; for this, we use simple type-level programming to extract values from the trace whose addresses α (in the full implementation) are indexed by an observable variable name from the input environment env. A similar approach is taken for Likelihood Weighting (lw) and Metropolis Hastings (mh).

6.2 Inference as Effect Handlers

Approximative Bayesian inference attempts to learn the posterior distribution of a model's parameters given some observed data. We reuse the ideas introduced in §6.1 to implement Likelihood Weighting and Metropolis Hastings as inference algorithms.

6.2.1 Likelihood Weighting (LW). If one uses simulation as a process for randomly proposing model parameters, the LW algorithm [van de Meent et al. 2018] then assigns these proposals a *weight*, that is, the total likelihood of them having generated some specified observed data.

Its implementation is in fact extremely simple, as most of the work has been done when implementing runSimulate. The only change is to how the Observe effect is interpreted:

```
handle_ObsLW :: Double → Prog (Observe : es) a → Prog es (a, Double)
handle_ObsLW lp (Op op k) = case discharge op of
  Right (Observe d y _) → handle_ObsLW (lp + logProb d y) (k y)
handle_ObsLW lp (Val x)   = return (x, lp)

logProb :: Dist a → a → Double
```

The function $\text{handle}_{\text{ObsLW}}$ is now parameterised by a log probability lp, where at each operation Observe d y α, it computes and adds to lp the log probability of distribution d having generated observed value y. The total log probability is returned upon reaching Val x.

The complete definition for a single iteration of LW is given as runLW:

```
runLW :: es ~ '[ObsReader env, Dist, State STrace, Observe, Sample]
      ⇒ Env env → Model env es a → IO ((a, STrace), Double)
runLW env = handle_Samp . (handle_ObsLW 0) . (handle_State Map.empty) . traceSamples . (handle_core env)
```

This is identical to simulation but we now use $\text{handle}_{\text{ObsLW}}$ instead of $\text{handle}_{\text{Obs}}$; running this returns the log-likelihood of the values in STrace giving rise to the observed data in provided environment env. Similar to runSimulate and simulate, runLW is called via a top-level function lw (omitted), but is instead performed iteratively to produce a trace of weighted parameter proposals (Fig. 1b).

Likelihood Weighting, however, becomes ineffective as the number of random variables sampled from increases: as values are freshly generated for *all* Sample operations, achieving a high likelihood means sampling an entire set of likely proposals. Our next, final example offers a solution to this:

6.2.2 Metropolis Hastings (MH). MH [Wingate et al. 2011] supports *incremental* parameter proposals by randomly choosing a "proposal address" α_0 at the start of each iteration; this denotes the address from which a new sample is to be drawn, where all other addresses are to instead reuse old samples from previous iterations. At the end of an iteration, we decide whether to accept the new sample by comparing the *individual* log probabilities of each probabilistic operation.

To support this, we take a similar approach to the one shown for traceSamples: we define the type LPTrace to map addresses of probabilistic operations to their log probabilities, and then the program transformation traceLPs to update this as a state.

```
type LPTrace = Map Addr Double

traceLPs :: (Member Observe es, Member (State LPTrace) es) ⇒ Prog es a → Prog es a
traceLPs (Op op k) = case prj op of
  Just (Observe d y α) → Op op (λx → do call (Modify (Map.insert α (logProb d y)))
                                        traceLP (k x)
```

On matching against Observe d y α, the above installs a State LPTrace effect by defining a new continuation that stores the log probability of d having generated y. The same is done for Sample operations, but we compute the log probability of the values sampled.

All of the conditioning is in fact taken care of by traceLPs, so the handler for Observe operations only needs to return any observed values to their continuations – its implementation is therefore the same as for simulation (§6.1). What is left to define is how Sample is interpreted:

```
handle_SampMH :: STrace → Addr → Prog '[Sample] a → IO a
handle_SampMH sTrace α₀ (Op op k) = case discharge op of
    Right (Sample d α) → do x ← lookupSample sTrace d α α₀
                            handle_SampMH α₀ sTrace (k x)

lookupSample :: STrace → Dist a → Addr → Addr → IO a
```

The handler $\text{handle}_{\text{SampMH}}$ takes the sample trace sTrace of the previous MH iteration, and an address α_0 denoting the proposed sample site. On matching against an operation Sample d α, the function lookupSample will generate a new sample if α matches α_0, or if no previous MH iterations have yet generated a sample for it; otherwise, it reuses the old sample value of α found in sTrace.

The final definition for a single iteration of MH is given as runMH:

```
runMH :: es ~ '[ObsReader env, Dist, State STrace, State LPTrace, Observe, Sample]
      ⇒ Env env → STrace → Addr → Model env es a → IO ((a, STrace), LPTrace)
runMH env sTrace α₀ = (handle_SampMH sTrace α₀) . handle_Obs
                    . (handle_State Map.empty) . (handle_State Map.empty)
                    . traceLPs . traceSamples . (handle_core env)
```

This reuses many of the building blocks of simulation, and extends these with the transformation traceLPs and handler for State LPTrace; the $\text{handle}_{\text{SampMH}}$ variant is then used instead of $\text{handle}_{\text{Samp}}$.

Although runMH adheres to the MH algorithm in terms of how it samples and conditions, it does not decide the proposal address α_0 at the start of an MH iteration, nor whether newly proposed parameters are accepted at the end of an iteration. We keep this logic distinct from effect handlers, and instead implement it in a wrapper function mh (omitted) which folds over iterations of runMH, propagating information from the previous MH iteration to support decision making in the next; the end result generates a trace of accepted parameter proposals (Fig. 5).

7 EVALUATION

7.1 Quantitative Evaluation

We compare our language's performance, which for simplicity is implemented using freer-simple[2], against two state-of-the-art PPLs. First, MonadBayes [Ścibior et al. 2018], a Haskell embedded language which uses a monad transformer library (mtl[3]) approach as an effect system for PPLs, but does not support multimodal models. Second, Turing [Ge et al. 2018], which achieves multimodal models via macro compilation in Julia as a host language.

Our evaluation strategy uses a set of popular benchmarks [Kulkarni et al. 2020], comparing simulation (SIM), Likelihood Weighting (LW), and Metropolis Hastings (MH) as forms of execution algorithms. We apply these to the following example models: linear regression, hidden Markov model, and latent dirichlet allocation. These benchmarks, given fully in Appendix A, are performed on an AMD Ryzen 5 1600 Six-Core Processor with 16GB of RAM.

Across all models, our performance scales linearly with the number of samples each algorithm generates; this remains mostly the case when varying the dataset size to models, except for inference on hidden Markov model where we begin to scale quadratically.

Against Turing, we are on average 4.0x and 1.2x faster for SIM and LW. As our benchmarks do not consider the overhead from Turing's macro-compilation stage for specialising multimodal models, this may indicate our runtime approach to model specialisation does not notably affect performance. MonadBayes is on average 3.3x faster than us for SIM and has close to constant-time performance for LW, making it difficult to compare the latter. In contrast to us, they choose to not store and update sample traces for SIM and LW; this effectively means their performance is impacted only by the cost of sampling operations (of which there are naturally fewer for LW), whereas we incur overhead from additional State operations. Another factor we heed is the performative difference between freer-simple and mtl for large, non-synthetic programs, which still requires investigation.

For MH in particular, Turing and MonadBayes are on average 3.1x and 1.8x faster. Our implementation is naive in that we traverse the whole sample trace for MH updates despite a fixed number of variables being updated – this is straightforward to amend [Wingate et al. 2011]. We also remark that Turing and MonadBayes only perform MH updates to *one* variable per iteration, and do not consider the dependent variables which consequently need updating [Kiselyov 2016b]; our implementation does account for this and hence incurs costs for updating groups of dependencies, which can be larger for models with more complex dependency graphs.

We find our language competitively performant despite not yet having explored potential optimisation, and expect to benefit greatly from off-the-shelf Haskell techniques such as inlining and more efficient data structures. Exploring performance of effect handlers with PPLs in general is an important and challenging topic we plan to investigate separately, including techniques specific to algebraic effects, such as the codensity monad [Voigtländer 2008], and alternative representations of effectful programs [Maguire 2019; Wu and Schrijvers 2015]. We also observe that *all* of our handlers are applied on each model execution, whereas we suspect partially pre-evaluated models can be executed by reconsidering the order of handling; this may lead to substantial performance gains.

7.2 Qualitative Evaluation

Finally, we compare our supported features across a larger range of modern PPLs in Table 1. To the best of our knowledge, our language is the first to fully support both multimodal and higher-order models. Models in Gen [Cusumano-Towner et al. 2019], denoted with the special @gen syntax,

[2] https://github.com/lexi-lambda/freer-simple
[3] https://github.com/haskell/mtl

Table 1. Comparison of PPLs in terms of supported features for models, where ● is full support, ◐ is partial support, and ○ is no support. Modular models are those that can be defined in terms of other models.

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

have support for higher-order interactions with other @gen terms; however, applying a standard higher-order function to a model, at least without programmer intervention, will escape the tracing of the model's computation. In Pyro [Bingham et al. 2019], models are Python functions and as such are first-class, but support for multimodal models is limited: random variables may have only *one* observed value at runtime, and so Pyro expects this value to be a matrix where all contained data is conditioned on simultaneously. This approach cannot be used for models with sequential behaviour such as HMMs.

As far as we know, our language is also the first general-purpose PPL with multimodal models in a statically typed paradigm. Stan [Carpenter et al. 2017] is type-safe but special-purpose. Other general-purpose PPLs with multimodality are dynamically typed (Pyro) or use just-in-time compilation (Turing, Gen), and so do not guarantee that the observed values assigned to random variables are correctly typed, or that these variables exist; however, some progress has been made towards a type-safe version of Gen [Lew et al. 2019]. Although MonadBayes is statically typed, we are not aware of any attempts to extend its mtl approach to support multimodal models.

We note that as our language is experimental, the range of model execution algorithms we present so far is small. Also, whilst we provide basic utilities such as IO debugging and type-safe interfacing with sample traces, richer PPL features such as run-time inference diagnostics have not yet been implemented. We believe our language's infrastructure, having embedded into an algebraic effect setting, can readily support extensions in both of these areas.

8 CONCLUSION

Probabilistic programming is an active research topic, with applications ranging from artificial intelligence [Tran et al. 2017] to market research [Letham et al. 2016]. Many existing PPLs are either too low-level to capture multimodal descriptions of models, or too rigid to allow models to be easily reused. In this paper we used an algebraic effects embedding to implement a PPL where multimodal models are first-class citizens that can be modularly defined and easily combined.

8.1 Related Work

8.1.1 Effect abstractions for PPLs. Although the design pattern of using interpreters to execute models is common in PPLs [Carpenter et al. 2017; Salvatier et al. 2016; Tolpin et al. 2016], viewing these interpreters formally as algebraic effect handlers has little existing literature. Ścibior and Kammar [2015] first demonstrate in Haskell how rejection sampling can be abstracted into handlers. Moore and Gorinova [2018] take inspiration from effect handlers and use Python *context managers* for sample tracing and conditioning; we distinguish these from algebraic effects, mainly because the operations are methods rather than syntax, handlers are coroutines invoked *by* programs, and operations and handlers are not associated with effects.

A notable alternative effect system is the *monad transformer library* (mtl) [Liang et al. 1995] approach adopted in MonadBayes [Ścibior et al. 2018]. The observation behind MonadBayes is that inference algorithms can be decomposed into monadic building blocks which together form a monad

transformer stack. This is comparable to our method, in that it also uses program transformations to augment models with additional inference-specific behaviours, but whereas we transform a syntactic representation of models via handler composition, Ścibior et al. directly execute their models via composition of monadic functions. Our handlers always produce intermediate programs of type `Prog es a`, and are therefore compositional; however, making handlers compose for arbitrary effects requires additional infrastructure to appropriately forward the effects in a way that is compatible with monad transformers [Schrijvers et al. 2019].

For constructing models, we find the `mtl` approach of MonadBayes offers a similar user experience to algebraic effects: in particular both support abstract effect signatures for model components, allowing effect constraints to be propagated bottom-up through the program. For implementing inference algorithms, the relative advantages of monads vs. algebraic effects are more apparent. Monads appear to be more lightweight building blocks when making incremental extensions to inference, allowing for finer-grained composition. On the other hand, working with monads can be tricky when building sophisticated algorithms: the `mtl` design pattern of MonadBayes is to have several monads invoke their own side-effects when sampling or observing, arranged by having each monad "`lift`" the computation up through the transformer stack. To invoke only a selection of particular monads requires the programmer to correctly position a monad in the stack and have it terminate the sequence of `lift`s early. Moreover as `mtl` performs all `lift`s implicitly via type-class instances, this tends to obscure the current monadic context. By contrast, expressing such logic in a single effect handler can yield a more cohesive solution. Monads also come with pre-defined semantics, whereas in inference it is often advantageous to override certain behaviours e.g. static vs. dynamic tracing of random choices [Ścibior et al. 2018]. To achieve this, the same monad along with *all* of its operations and type-class instances must be redefined afresh in separate modules; this results in a proliferation of redundant boilerplate, and can make arduous the task of parameterising algorithms by new semantics.

8.1.2 Embedding PPLs. Tagless-final shallow embedding, as conceived by Kiselyov [2010], has often been used to correspond PPL syntax to type class methods [Kiselyov 2016a; Kiselyov and Shan 2009; Narayanan et al. 2016], where type class instances correspond to interpretations of programs under particular inference algorithms. This is well suited for applying a semantic domain uniformly over a program, but we found it difficult to iterate transformations (such as interpretations to `sample` and `observe`) over models to implement inference compositionally.

Free monads have also seen use, where Ścibior et al. [2015] embed primitive and conditional distributions as an intermediate free monad representation. This bears an initial resemblence to us, but the semantics are instead provided using type classes, and their direct encoding of conditional distributions means models are not multimodal. Later work by Ścibior et al. [2018] more closely coincides with our approach, where their implementation of Metropolis-Hastings uses free monad transformers [Schrijvers et al. 2019] to encode sampling operations as syntax; this allows models to be executed so that sampling can either invoke an `IO` effect or reuse previous samples.

8.1.3 Observed data for multimodal models. In terms of specifying observed data to multimodal models, Pyro [Bingham et al. 2019] and Gen [Cusumano-Towner et al. 2019] take the most similar approach to us initially: letting users provide mappings between variable names and observed values to a top-level function. Pyro then instead wraps models in a runtime context manager that constrains the values of variables to observed data, whereas Gen uses the data to macro-expand models into either a static computation graph or standard function. To support *many* observed values for the same variable, Pyro relies on these to be presented as a single matrix to be conditioned on together (§7.2), and Gen requires values to be individually mapped to *runtime* occurrences of a variable; our approach instead allows traces (lists) of values to be assigned to the same variable.

Special-purpose PPLs such as Stan [Carpenter et al. 2017] and WinBUGS [Lunn et al. 2000] specify models and observed data separately via bespoke language constructs, where common variable names are consolidated during compilation to a different target language.

Although our specific approach to model environments appears to be novel, closely related is work by Lew et al. [2019]. They investigate row polymorphism for assigning types to names of random choices when tracing probabilistic computations; we hope to build on this idea when formalising our language, in particular, for characterising the execution space of a multimodal model under a certain model environment.

8.2 Future Work

Our implementation currently responds to running out of observed values in the model environment by switching to sampling, as is typical in PPLs [Bingham et al. 2019; Ge et al. 2018]. However, it may be possible in some settings to use type-level naturals to statically constrain the number of observations required. Alternatively, a dynamic check to signal when too many or too few observations are provided would be easy to implement.

We are also investigating how naming conflicts between observable variables should be resolved when combining models. This is not considered an issue by most existing PPLs; many require programmers to uniquely name each dynamic random variable instance [Cusumano-Towner et al. 2019; Salvatier et al. 2016], perhaps failing at runtime if this condition is not met [Bingham et al. 2019]. Our language is type-safe in allowing model environments to have typed, orthogonally combinable variables (via constraint kinds), but for additional modularity, we are also considering a renaming mechanism for rebinding observable variables when name clashes arise.

A related topic is when the programmer statically refers to the same observable variable more than once in a model:

```
do x1 ← normal 0 1 #x
   x2 ← normal 0 2 #x
   return (x1 + x2)
```

Technically, this is an invalid use of a random variable, resulting in an ill-formed model where `#x` is (confusingly) distributed according to two different distributions. For now, programmers must take care not to misuse observable variables in this way; a solution we intend to explore is an affine type system for model contexts which will disallow multiple static uses of the same observable variable. We are also working on a formalisation of our language and embedding technique.

Lastly, we aim to explore how using effect handlers as program transformations can help with the implementation of sophisticated, compositional inference algorithms such as SMC^2 [Doucet et al. 2001] and PMMH [Chopin 2002]; we anticipate needing to reason about interactions with new effects, for example with non-determinism and exceptions, and whether such effects distribute over [Wu et al. 2014] and commute with each other [Gibbons and Hinze 2011] in a probabilistic setting. For scaling up to advanced techniques that require gradient information, e.g. HMC and NUTS [Hoffman et al. 2014], models also need to be differentiable. This is generally done in PPLs via *automatic differentation* (AD), a procedure for instantiating standard programs as differentiable functions. We believe it possible for our language's infrastructure to support this as a new effect, perhaps by building on top of recent work by Sigal [2021] who show it indeed possible to express AD through effect handlers. A simpler option may be to handle a model into a form suitable for use in an existing AD library [Kmett et al. 2021]. In either case, we would need to investigate the set of valid operations or syntactic forms that enable an AD interpretation of multimodal models in our language.

REFERENCES

I Ameen, Dumitru Baleanu, and Hegagi Mohamed Ali. 2020. An efficient algorithm for solving the fractional optimal control of SIRV epidemic model with a combination of vaccination and treatment. *Chaos, Solitons & Fractals* 137 (2020), 109892.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20 (2019), 28:1–28:6. http://jmlr.org/papers/v20/18-403.html

Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan : A Probabilistic Programming Language. *Journal of Statistical Software* 76 (2017). https://doi.org/10.18637/jss.v076.i01

Jike Chong, Ekaterina Gonina, and Kurt Keutzer. 2010. Monte Carlo Methods: A Computational Pattern for Our Pattern Language. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns* (Carefree, Arizona, USA) *(ParaPLoP '10)*. Association for Computing Machinery, New York, NY, USA, Article 15, 10 pages. https://doi.org/10.1145/1953611.1953626

Nicolas Chopin. 2002. A sequential particle filter method for static models. *Biometrika* 89, 3 (2002), 539–552. https://doi.org/10.1093/biomet/89.3.539

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-Purpose Probabilistic Programming System with Programmable Inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 221–236. https://doi.org/10.1145/3314221.3314642

Zhenghao Ding, Jun Li, and Hong Hao. 2019. Structural damage identification using improved Jaya algorithm based on sparse regularization and Bayesian inference. *Mechanical Systems and Signal Processing* 132 (2019), 211–231.

Arnaud Doucet, Nando de Freitas, and Neil Gordon. 2001. *An Introduction to Sequential Monte Carlo Methods*. Springer New York, New York, NY, 3–14. https://doi.org/10.1007/978-1-4757-3437-9_1

Conal Elliott, Sigbjørn Finne, and Oege De Moor. 2003. Compiling Embedded Languages. *J. Funct. Program.* 13, 3 (2003), 455–481. https://doi.org/10.1017/S0956796802004574

Martin Erwig and Steve Kollmansberger. 2006. Functional Pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.* 16, 1 (2006), 21–34. https://doi.org/10.1017/S0956796805005721

Phil Freeman. 2017. PureScript by Example.

Tadayoshi Fushiki. 2010. Bayesian bootstrap prediction. *Journal of statistical planning and inference* 140, 1 (2010), 65–74.

Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: a language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*. 1682–1690. http://proceedings.mlr.press/v84/ge18b.html

Andrew Gelman and Jennifer Hill. 2006. *Data analysis using regression and multilevel/hierarchical models*. Cambridge university press.

Jeremy Gibbons and Ralf Hinze. 2011. Just do it: simple monadic equational reasoning. *ACM SIGPLAN Notices* 46, 9 (2011), 2–14.

Jeremy Gibbons and Nicolas Wu. 2014. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 339–347. https://doi.org/10.1145/2628136.2628138

Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2021-5-24.

Daniel Hillerström. 2015. *Handlers for algebraic effects in Links*. Ph. D. Dissertation. MSc thesis, The University of Edinburgh, Scotland.

Daniel Hillerström and Sam Lindley. 2018. Shallow effect handlers. In *Asian Symposium on Programming Languages and Systems*. Springer, 415–435.

Matthew D Hoffman, Andrew Gelman, et al. 2014. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.* 15, 1 (2014), 1593–1623.

Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. 2015. Overview of data exploration techniques. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 277–281.

Oleg Kiselyov. 2010. Typed Tagless Final Interpreters. In *Proceedings of the 2010 International Spring School Conference on Generic and Indexed Programming* (Oxford, UK) *(SSGIP'10)*. Springer-Verlag, Berlin, Heidelberg, 130–174. https://doi.org/10.1007/978-3-642-32202-0_3

Oleg Kiselyov. 2016a. Probabilistic programming language and its incremental evaluation. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Proceedings*, Atsushi Igarashi (Ed.). Springer Verlag, 357–376. https://doi.org/10.1007/978-3-319-47958-3_19

Oleg Kiselyov. 2016b. Problems of the lightweight implementation of probabilistic programming. In *Proceedings of Workshop on Probabilistic Programming Semantics*.

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, Canada) *(Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2804302.2804319

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible Effects: An Alternative to Monad Transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell* (Boston, Massachusetts, USA) *(Haskell '13)*. Association for Computing Machinery, New York, NY, USA, 59–70. https://doi.org/10.1145/2503778.2503791

Oleg Kiselyov and Chung-Chieh Shan. 2009. Embedded Probabilistic Programming. In *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages* (Oxford, UK) *(DSL '09)*. Springer-Verlag, Berlin, Heidelberg, 360–384. https://doi.org/10.1007/978-3-642-03034-5_17

Brendan Kline and Elie Tamer. 2016. Bayesian inference in a class of partially identified models. *Quantitative Economics* 7, 2 (2016), 329–366.

Edward Kmett, Barak Pearlmutter, and Jeffrey Mark Siskind. 2010-2021. ad: Automatic Differentiation. Haskell package at https://hackage.haskell.org/package/ad.

Sourabh Kulkarni, Kinjal Divesh Shah, Nimar S. Arora, Xiaoyan Wang, Yucen Lily Li, Nazanin Khosravani Tehrani, Michael Tingley, David Noursi, Narjes Torabi, Sepehr Akhavan-Masouleh, Eric Lippert, and Erik Meijer. 2020. PPL Bench: Evaluation Framework For Probabilistic Programming Languages. *CoRR* abs/2010.08886 (2020). arXiv:2010.08886

Daan Leijen. 2005. Extensible records with scoped labels. In *Proceedings of the 2005 Symposium on Trends in Functional Programming*. 297–312.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2020. *The OCaml system release 4.11: Documentation and user's manual*. Ph. D. Dissertation. Inria.

Benjamin Letham, Lydia M. Letham, and Cynthia Rudin. 2016. Bayesian Inference of Arrival Rate and Substitution Behavior from Sales Transaction Data with Stockouts. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 1695–1704. https://doi.org/10.1145/2939672.2939810

Alexander K Lew, Marco F Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K Mansinghka. 2019. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32.

Feng Liang and Ziyuan Li. 2021. Statistical Analysis on COVID-19 Based on SIR Model. In *2020 4th International Conference on Computational Biology and Bioinformatics* (Bali Island, Indonesia) *(ICCBB 2020)*. Association for Computing Machinery, New York, NY, USA, 14–19. https://doi.org/10.1145/3449258.3449261

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 333–343. https://doi.org/10.1145/199448.199528

David J. Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. 2000. WinBUGS – A Bayesian Modelling Framework: Concepts, Structure, and Extensibility. *Statistics and Computing* 10, 4 (2000), 325–337. https://doi.org/10.1023/A:1008929526011

Sandy Maguire. 2019. Polysemy. https://github.com/polysemy-research/polysemy.

Todd K Moon. 1996. The expectation-maximization algorithm. *IEEE Signal processing magazine* 13, 6 (1996), 47–60.

Dave Moore and Maria I Gorinova. 2018. Effect handling for composable program transformations in edward2. *arXiv preprint arXiv:1811.06150* (2018).

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru. In *International Symposium on Functional and Logic Programming*. Springer, 62–79. https://doi.org/10.1007/978-3-319-29604-3_5

Gordon Plotkin and John Power. 2003. Algebraic operations and generic effects. *Applied categorical structures* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

Gordon Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013). https://doi.org/10.2168/lmcs-9(4:23)2013

Nicholas G Polson, James G Scott, and Jesse Windle. 2013. Bayesian inference for logistic models using Pólya–Gamma latent variables. *Journal of the American statistical Association* 108, 504 (2013), 1339–1349.

L. Rabiner and B. Juang. 1986. An introduction to hidden Markov models. *IEEE ASSP Magazine* 3, 1 (1986), 4–16. https://doi.org/10.1109/MASSP.1986.1165342

Christian P Robert and DM Titterington. 1998. Reparameterization strategies for hidden Markov models and Bayesian approaches to maximum likelihood estimation. *Statistics and Computing* 8, 2 (1998), 145–158.

John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* 2 (2016), e55. https://doi.org/10.7717/peerj-cs.55

Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. 2019. Monad Transformers and Modular Algebraic Effects: What Binds Them Together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell* (Berlin,

Germany) *(Haskell 2019)*. Association for Computing Machinery, New York, NY, USA, 98–113. https://doi.org/10.1145/3331545.3342595

Adam Ścibior, Zoubin Ghahramani, and Andrew D. Gordon. 2015. Practical Probabilistic Programming with Monads. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, Canada) *(Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/2804302.2804317

Adam Ścibior and Ohad Kammar. 2015. Effects in Bayesian inference. In *Workshop on Higher-Order Programming with Effects (HOPE)*.

Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. 2018. Functional Programming for Modular Bayesian Inference. *Proc. ACM Program. Lang.* 2, ICFP, Article 83 (2018), 29 pages. https://doi.org/10.1145/3236778

Hongjing Shi, Zhisheng Duan, and Guanrong Chen. 2008. An SIS model with infective medium on complex networks. *Physica A: Statistical Mechanics and its Applications* 387, 8-9 (2008), 2133–2144.

Jesse Sigal. 2021. Automatic differentiation via effects and handlers: An implementation in Frank. *arXiv preprint arXiv:2101.08095* (2021).

Wouter Swierstra. 2008. Data types à La Carte. *J. Funct. Program.* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages* (Leuven, Belgium) *(IFL 2016)*. Association for Computing Machinery, New York, NY, USA, Article 6, 12 pages. https://doi.org/10.1145/3064899.3064910

Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. arXiv:1701.03757 [stat.ML]

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756* (2018).

Janis Voigtländer. 2008. Asymptotic improvement of computations over free monads. In *International Conference on Mathematics of Program Construction*. Springer, 388–403.

David Wingate, Andreas Stuhlmueller, and Noah Goodman. 2011. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 15)*, Geoffrey Gordon, David Dunson, and Miroslav Dudík (Eds.). PMLR, Fort Lauderdale, FL, USA, 770–778. https://proceedings.mlr.press/v15/wingate11a.html

N Wu and T Schrijvers. 2015. Fusion for Free Efficient Algebraic Effect Handlers. Springer-Verlag Berlin, 302–322. https://doi.org/10.1007/978-3-319-19797-5_15

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. 1–12.

Daniel Yekutieli. 2012. Adjusted Bayesian inference for selected parameters. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 74, 3 (2012), 515–541.

A PERFORMANCE BENCHMARKS

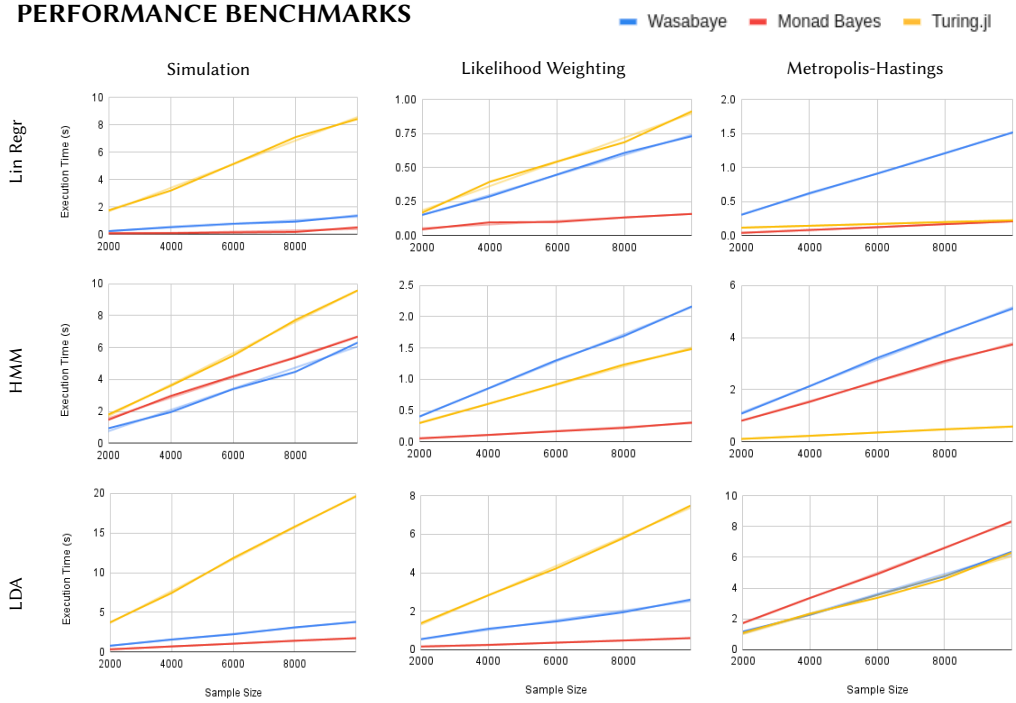


Fig. 8. Execution times of algorithms (seen at top) applied to models (seen on left) with respect to the number of samples each algorithm generates (i.e. number of algorithm iterations).

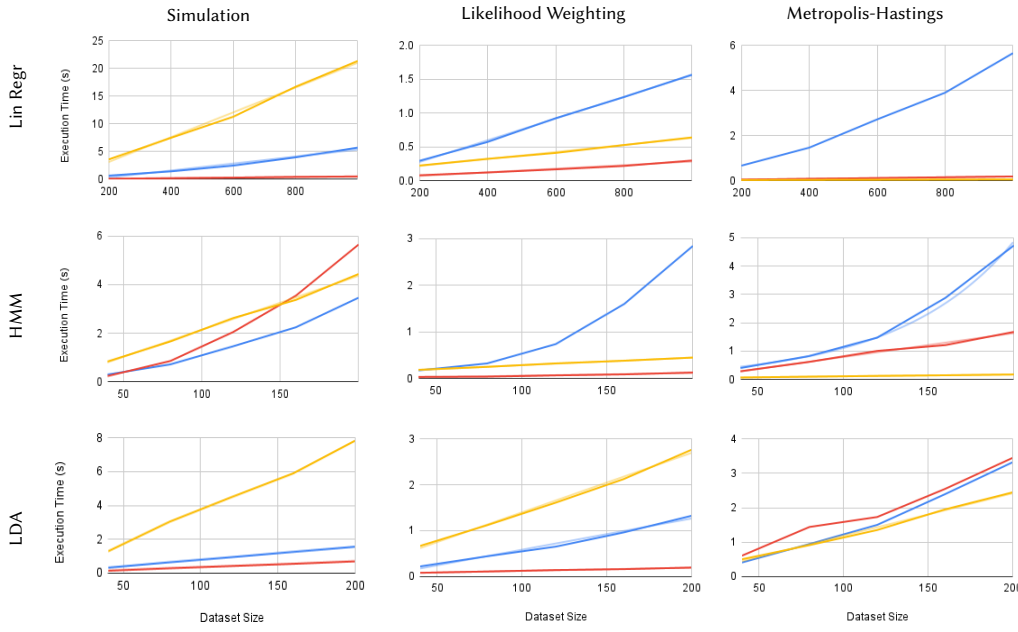


Fig. 9. Execution times of algorithms (at top) applied to models (on left) with respect to dataset size provided to models. The data we vary over is: the number of data points in linear regression, the number of Markov nodes in the hidden Markov model, and the length of text document for latent dirichlet allocation.