Composable, Modular Probabilistic Models

Minh Nguyen
min.nguyen@bristol.ac.uk
University Of Bristol
United Kingdom

Roly Perera
roly.perera@bristol.ac.uk
The Alan Turing Institute
United Kingdom

Meng Wang
meng.wang@bristol.ac.uk
University Of Bristol
United Kingdom

Abstract

Probabilistic programming languages (PPLs) allow one to construct statistical models to describe certain problem domains, and then simulate data or perform inference over them [9]. In many PPLs, models lack reusability as they are forced to be defined for a specific use-case: simulation *or* inference. In other PPLs, models lack the ability to be combined or composed.

This paper presents a DSL in Haskell for modularly defining probabilistic models which are combinable and composable, and can be reused for both simulation and inference. We then demonstrate how simulation and inference can be expressed naturally as composable program transformations using algebraic effect handlers.

1 Introduction

1.1 An Outline of Probabilistic Programming

By integrating notions of statistics with general purpose programming languages, PPLs allow one to capture real world phenomena through the construction of a program as a probabilistic model. What distinguishes a probabilistic language from a general purpose one is the ability to perform two computational effects: to *sample*, which is to draw and return a random variable from a probability distribution, and to *observe*, which is to condition against the probability of a distribution giving rise to an observed value and then return the observed value itself. Given these two probabilistic constructs, one then expects to be able to:

(1) Specify a model in terms of mathematical relationships between random variables in a program.
(2) Simulate data from a model, given a set of model parameters.
(3) Infer the parameters of a model, given some observed data to condition against.

1.2 Motivation: Query-Based & Model-Based Languages

To make the workflow of using probabilistic languages as convenient as possible, there are two significant challenges to engage with. To introduce these, we first elaborate on how PPLs can be categorized into query-based languages and model-based languages.

- *Query-based* languages (Anglican [27], WebPPL [8], MonadBayes [23]) work via the user writing a probabilistic query as a normal function but with the freedom to explicitly call sample and observe as effectful computations. This approach benefits from being flexible and general purpose – we can directly express a probabilistic computation for a given problem. Moreover, queries being more-or-less functions means that they can be easily combined (they can call other queries) and sometimes even functionally composed.

However, the explicit use of sample and observe means that query-based PPLs can only express "instances" of models which are specific to how they will used – either for simulation *or* inference. One can see how this soon becomes frustrating when considering the following example program in Anglican:

```
(defquery linearRegression [mu c sigma data_y x]
(let [y (sample (normal (mu * x + c) sigma))]
     {:y y}))
```

Figure 1: Linear Regression (Simulation) in Anglican

```
(defquery linearRegression [data_y x]
(let [mu    (sample (normal 0 3))
      c     (sample (normal 0 2))
      sigma (sample (uniform 1 3))
      y     (observe (normal (mu * x + c) sigma) data_y)]
     {:mu mu :c c :sigma sigma :y y}))
```

Figure 2: Linear Regression (Inference) in Anglican

Figure 1 depicts a linear regression model which is suitable for simulation, whereas Figure 2 defines the same model but for inference instead. This leads to having to redefine the same model for each possible interpretation. It hence becomes tedious to iterate through and evolve models whilst maintaining each model interpretation.

- *Model-based* languages (Turing.jl [6], Gen.jl [5], PyMC3 [20], Stan [3]) instead allow the user to construct models as a description of relationships between random variables, where the semantics of sampling and observing are not tied to the model syntax. For example, consider the following program in Turing.jl:

```
@model function linearRegression(mu, c, sigma, x, y)
  mu    ~ Normal(0, 3)
  c     ~ Normal(0, 2)
  sigma ~ Uniform(1, 3)
  y     ~ Normal(mu * x + c, sigma)
end
```

Figure 3: Linear Regression in Turing.jl

Figure 3 shows how one would generally expect linear regression to be written in a model-based language. Importantly, this allows us to independently specify and develop models whilst being able to simulate and infer on the same model definition.

A considerable limitation of existing model-based languages is that it is either impossible to combine models or the means of doing so is awkward, and it is impossible to compose models. This often stems from it being difficult to design models as first class citizens, resulting in them having to be monolithically defined and unable to be manipulated by constructs such as higher-order functions and functional combinators. Moreover, this makes modular debugging difficult if we are unable to isolate and identify the defective components of a model.

The drawbacks of query-based and model-based languages can be re-expressed as the two following challenges:

(1) How difficult it is to specify and iterate through different models. It is common to want to quickly experiment with many model definitions before considering how they will be used, if at all.
(2) How difficult it is to combine and compose different models. This is essential for modular development of hierarchical models, where compound models are constructed from independently defined sub-models.

Effectively achieving both of these properties is something that is left to be desired amongst current PPLs.

1.3 Contributions

We address the previous issues by presenting an embeddeding of a model-based PPL for implementing modular, composable models. The design relies on algebraic effects and extensible data, where we demonstrate the implementation in Haskell. In particular:

- We achieve the first "model-based", universal PPL in a functional paradigm – models can be assigned multiple different interpretations for both simulation and inference by letting them exist independently of how they will be used and the observed data they condition against. By using freer monads [12] and a distribution data type, our language unifies the syntax for sampling and observing, enabling models to be syntactically constructed as descriptions of generative processes.
- We manage to capture probabilistic models as first-class objects for the first time in a model-based language; they can be combined monadically and composed with higher-order functions such as kleisli composition, fold, map, etc.
- We implement a mechanism for cleanly associating observed data to the distributions within our model in order to specify which probabilistic calls should be interpreted as `sample` or `observe`. This is accomplished using extensible records [15] and affine effects; importantly, this maintains the generality of a model description and does not confine it to any specific semantics.
- We demonstrate how simulation and inference algorithms can be expressed modularly using effect handlers [19] to perform composable program transformations on models. This effectively describes simulation and inference as higher-order functions which take models and embed them into the semantics of a specific algorithm, creating a separation of concerns between models and how they are used.

1.4 Structure of the Paper

We start by identifying the core components and needs of our ideal probabilistic language and then discuss a corresponding optimal embedding strategy; this will allow us to syntactically construct simple models (§ 2). We then review the limitations of our language which arise when considering models with recursive or iterative structure; revising our implementation permits more sophisticated models which can be used generically with higher-order functions (§ 3). Next, we begin to assign basic semantics to models by defining an set of initial effect handlers; these let us reconstruct models into a form which is ripe for simulation or inference (§ 4). Finally, we show how simulation and inference can be implemented as composable program transformations by defining a set of algorithm-specific effect handlers (§ 5).

2 Capturing Probabilistic Models As Syntax

In this section, we develop a DSL for syntactically constructing probabilistic models. We initially discuss the needs of our language and decide on a suitable embedding approach based around algebraic effects (§ 2.1). We then define the key effects of our language: distributions (§ 2.2) and observable variable environments (§ 2.3). Lastly, we detail the mechanism behind how distribution operations are used to construct models (§ 2.4); this allows us to write simple probabilistic models, such as linear regression (§ 2.5).

2.1 Embedding Strategies

First, we discuss how best to embed a language for probabilistic models such that their semantic interpretation is delayed until simulation or inference is called. There are numerous strategies that can be regarded such as deep embeddings [25] and tagless-final shallow [10] but we consider these as excessive for our purposes, and moreover, they are fundamentally limited by their restricted access to host language constructs.

Ideally we should only need to define the minimal constructs necessary for our language which demand their own unique treatment in a probabilistic domain, namely *distributions*. Algebraic effects [18] are an excellent tool for this purpose which provide a natural separation between their syntax (as an interface of effectful operations) and semantics (as effect handlers).

2.1.1 Free Monads are a straightforward and general way of inducing an algebraic effect system [26], defined as `Free f a`:

```
data Free f a = Pure a | Free (f (Free f a))
```

They allow us to construct a syntactic tree where its leaves, `Pure`, represent pure values of type `a`, and its nodes, `Free`, are operations shaped by some "effect" functor `f`. One could imagine introducing distributions as a functorial GADT, `Dist`, for `f`:

```
data Dist a where
  NormalDist :: Double -> Double -> (Double -> a) -> Dist a
  deriving Functor
type Model a = Free Dist a
```

The syntax of a probabilistic program would then be constructed as the free monad tree with nodes shaped by distributions. This idea is incomplete however: probabilistic programs naturally require IO effects for sampling random values.

2.1.2 Free Monad Transformers One solution would be to use free monad transformers, `FreeT f m a`, which enable us to interleave monadic effects of a given monad `m` throughout a free monad tree:

```
data FreeF f a x = Pure a | FreeF (f x)
newtype FreeT f m a =
  FreeT { runFreeT :: m (FreeF f a (FreeT f m a)) }
type Model a = FreeT Dist IO a
```

Of course, there are many more effects which one may need to introduce into a program, which can be necessary for either inference or the model itself. Obvious examples include the State effect for retaining trace information of the model's execution, and the List effect to represent populations of samples from the model. To include all of these as a nesting of monad transformers for `m` in `FreeT f m a` would be an unpleasant solution.

2.1.3 Freer Monads The solution we settle on, described by Kiselyov and Ishii [12], uses *open sums* to represent a coproduct of multiple effects:

```
data OpenSum (ts :: [* -> *]) x where
  OpenSum :: Int -> t x -> OpenSum ts x
```

Open sums are containers of data whose type isn't statically known, i.e. is existentially quantified. Here, the type parameter `ts` does not represent a single effect, but a type-level list of possible effects in our program. The `OpenSum` constructor then takes an effectful operation `t x` along with the integer corresponding to `t`'s position in `ts` – this allows us to recover `t`'s concrete type.

Safely *injecting* and *projecting* an effect `t` into and out of an open sum, `OpenSum ts x`, requires us to first determine the position of `t` in `ts`. This is made possible by the type class `FindElem`, whose function `findElem` returns the integer index:

```
newtype Idx t ts = Idx {unIdx :: Int}
class FindElem t ts where
  findElem :: Idx t ts
```

The type class `Member` then specifies that if we can determine `t`'s position in `ts`, then we can inject and project it. Injection, `inj`, takes an operation `tx` and adds it to the open sum `ts`. Projection, `prj`, attempts to extract an operation `tx` from an open sum `ts`.

```
class (FindElem t ts) =>
      Member (t :: * -> *) (ts :: [* -> *]) where
  inj :: t x -> OpenSum ts x
  prj :: OpenSum ts x -> Maybe (t x)
```

To later handle and discharge effects from an open sum, we must also be able to decompose a value of type `OpenSum (t ': ts) v` into either being the first effect `t` or some other effect in `ts`. This is defined as the function `decomp`:

```
decomp :: OpenSum (t ': ts) v -> Either (OpenSum ts v) (t v)
```

The actual effect system which uses open sums is known as the *freer* monad, given below:

```
data Freer ts a where
  Pure :: a -> Freer ts a
  Free :: OpenSum ts x -> (x -> Freer ts a) -> Freer ts a
```

Each `Free` constructor in our tree now contains an argument of type `OpenSum ts x`, allowing individual nodes to be shaped by any of the effects found in `ts`. The act of sending operations to a program can be abstracted into the function `send`, which is equivalent to extending the tree at its leaves:

```
send :: Member t ts => t x -> Freer ts x
send tx = Free (inj tx) Pure
```

A detail to note about the `Freer` monad is that its `Free` constructor now stores an "operation continuation", `(x -> Freer f a)`. This means that data types `t` for effects no longer need to be functors (as opposed to the free monad) and hence can be defined more elegantly without their continuations. This feature allows primitive distributions to be defined more naturally as GADTs which are parameterised by the type of value they generate (as seen in § 2.2), and is partially why we find Kiselyov and Ishii's approach more suitable than other similar alternatives [13, 32].

2.2 Distribution Types

The core effect which our language is comprised of is the distribution data type, `Dist`, where its constructors are various primitive distributions. Of course, distributions cannot be concretely interpreted as any single effect – there are two operations which we expect to be able to perform on them: sampling and observing. To avoid explicitly using these operations in a model, we need the syntax of distributions to somehow unify these two notions.

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

To solve this, each constructor of `Dist` first takes a number of distribution parameters followed by a value of type `Maybe a` where `a` matches the type of value the distribution generates. This indicates the presence of an observed value to condition against, deciding whether we sample or observe.

We can now begin to express the type of models as the following:

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

This states that a `Model` is given by the `Freer` monad where `Dist` is a known member in the list of effects `ts`.

2.3 Environments For Observable Variables

Whether the user provides observed data to a distribution or not decides when sampling and observing occur. The mechanism for how we allow observed data to be specified should ideally only happen when the entire model is necessarily run. It is thus a flawed idea to pass this data as explicit function arguments to a model, as this detracts from the model existing independently from the notions of simulation and inference.

2.3.1 Reading Observed Variables A simple yet effective solution is to incorporate the `Reader` type in our list of `Freer` effects.

```
data Reader env a where
  Ask :: Reader env env
ask :: (Member (Reader env) ts) => Freer ts env
ask = send Ask
```

Its type parameter `env` represents the environment containing all observable variables in the model that may be conditioned against. The smart constructor `ask` can then be used to inject an `Ask` operation into a `Freer f env` computation.

The `Reader` effect would indeed move the details about observed data to the type-level, but using this approach on its own is problematic in that it would force each model to have its own fixed environment. This is overly rigid and prohibits the ability to combine and compose different models. A solution to this is to use *open products* (i.e. extensible records [15]) as environments:

2.3.2 Extensible Environments Ideally, model environments should be kept abstract and only require that the observed variables relevant to the model must be specified. This can be achieved by using *open products* (i.e. extensible records [15]) as environments:

```
data Assoc x v = x :> v
data Any where
  Any :: a -> Any
data OpenProduct (xvs :: [Assoc Symbol k]) where
  OpenProduct :: Vector Any -> OpenProduct xvs
```

An open product, OpenProduct xvs, contains a vector of existentially quantified types. Their type parameter xvs represents our model environment and is a type-level list of associations between observable variable names x_i (of kind Symbol) and values v_i (of kind k); this lets us to keep track of which types are stored in the vector whilst also corresponding them to specific variable names.

To refer to the variables of a record, we require a mechanism for passing types of kind Symbol. The phantom data type Var is defined for this purpose, acting as a container for type-level strings.

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

A well known method for creating values such as Var in an uncumbersome way, is to derive an instance of the IsLabel class using GHC's OverloadedLabels language extension.

```
instance (KnownSymbol x, x ~ x') => IsLabel x (Var x') where
  fromLabel = Var
```

Var values may now be formed using the # syntax; for example, the value #foo will have type Var "foo".

Constructing an open product can be done by iteratively inserting new variable-value pairs into an initially empty open product, OpenProduct nil. Insertion must ensure that the positions of its value and types are aligned at the term-level and type-level; the function insert hence adds the variable and value type to the head of the type list xvs, and cons's the value to the head of the internal vector:

```
insert :: Var x -> v -> OpenProduct xvs
       -> OpenProduct (x ':> v ': xvs)
insert _ v (OpenProduct vs) = OpenProduct (cons (Any v) vs)
```

In order to look up a field in the open product, we require that we must be able to find its variable name in xvs and determine the type of its correspond value. The former is enforced by defining an instance of the type class FindElem, which returns the index of a variable in an open product. The latter is done by the type family LookupType, which returns the type of the associated value.

```
instance FindElem x ((x ':> v) ': xvs) where
  findElem = Idx 0
instance FindElem x xvs => FindElem x (xv ': xvs) where
  findElem = Idx $ 1 + unIdx (findElem :: Idx x xvs)

type family LookupType x xvs where
  LookupType x ((x ':> v) : xvs) = v
  LookupType x ((x' ':> v) : xvs) = LookupType x xvs
```

To express both of these more conveniently, the type class Lookup is provided which states that we may lookup variable x with a value of type v in list of associations xvs:

```
class (FindElem x xvs, LookupType x xvs ~ v) => Lookup xvs x v
instance (FindElem x xvs, LookupType x xvs ~ v) => Lookup xvs x v
```

Getting and *setting* fields can then be implemented in a safe fashion using TypeApplications, where the syntax @v passes an explicit type argument v to a polymorphic function.

```
getOP :: forall x xvs v. (Lookup xvs x v)
      => Var x -> OpenProduct xvs -> v
getOP _ (OpenProduct vs) =
  unAny (V.unsafeIndex vs (unIdx $ findElem @x @xvs))
  where
    unAny (Any v) = unsafeCoerce v

setOP :: forall x xvs v. (Lookup xvs x v)
    => Var x -> v -> OpenProduct xvs -> OpenProduct xvs
setOP _ v (OpenProduct vs) =
  OpenProduct (vs // [(unIdx (findElem @x @xvs), Any v)])
```

The definition of getOP says: if the variable x can be found in an open product, then we can determine its position using findElem and look it up. A similar logic applies to setting fields with setOP.

2.3.3 Reading Extensible Environments in Models An important remark about how open products are used in the context of probabilistic models, is that all observed variables must be associated with values of type **Maybe** v. This structure can be abstracted away into the type family AsMaybes which maps the **Maybe** type over a list of associations.

```
type family AsMaybes (as :: [k]) = (bs :: [k]) | bs -> as where
  AsMaybes ((x :> v) : xvs)   = ((x :> Maybe v) : AsMaybes xvs)
  AsMaybes '[] = '[]
```

We then introduce the type class Observable to be used as shorthand instead of Lookup for looking up values with type **Maybe** v.

```
class Lookup (AsMaybes xvs) x (Maybe v) => Observable xvs x v
instance Lookup (AsMaybes xvs) x (Maybe v) => Observable xvs x v
```

With this, the definition for Model can finally be updated: we now also include the Reader effect where its environment is an extensible record containing observable variables. The environment is specified as an extra type parameter to Model called env:

```
type MRecord xvs = OpenProduct (AsMaybes xvs)
newtype Model env ts a
  = Model { runModel :: ( Member Dist ts
                        , Member (Reader (MRecord env)) ts)
                        => Freer ts a }
```

What is pragmatic about this solution is that the environments of models can remain abstract, and the specifics about any necessary observable variables are elegantly isolated as type class constraints. This permits different model implementations to be combined and composed.

2.4 Distribution Operations

All models are ultimately made up of a sequence of calls to primitive distributions. There are two possible ways one could call a distribution, hence we define a smart constructor for each.

The first would be to pass the required distribution parameters as well as an observed variable name – this could represent either sampling or observing, based on whether a value exists for that variable. We demonstrate this using the normal distribution:

```
normal :: forall env x ts. (Observable env x Double)
       => Double -> Double -> Var x -> Model env ts Double
normal mu sigma var = Model $ do
  env :: MRecord env <- ask
  let obs = getOP var env
  send (NormalDist mu sigma obs)
```

The function normal takes a mean, mu, and standard deviation, sigma, as parameters, followed by an observed variable name x. We first send an ask operation to retrieve the environment of observed data and extract from it the value corresponding to the observed variable. Finally, this is used to send an operation for the normal distribution. Other primitive distributions are implemented similarly.

Realistically, only a subset of distribution calls would be observed against, so requiring an observed variable to be given to every distribution would be laborious. Hence, the second way to call a distribution would be to pass only the required distribution parameters and use Nothing for the observed value:

```
normal' :: Double -> Double -> Model env ts Double
normal' mu sigma = Model $ do
  send (NormalDist mu sigma Nothing)
```

Semantically, this would be equivalent to always sampling.

2.5 Example: Linear Regression

The methodology described so far gives us a solid foundation on which we can formulate basic models. The following example expresses linear regression, i.e. a model that assumes a linear relationship between input variables x and output variables y:

```
prior :: (Observables env '["mu", "c", "std"] Double)
      => Model env ts (Double, Double, Double)
prior = do
  mu  <- normal 0 3 #mu
  c   <- normal 0 2 #c
  std <- uniform 1 3 #std
  return (mu, c, std)

linRegr :: (Observables env '["y", "mu", "c", "std"] Double)
        => Double -> Model env ts (Double, Double)
linRegr x = do
  (mu, c, std) <- prior
  y <- normal (mu * x + c) std #y
  return (x, y)
```

The linear regression model, linRegr, takes x as input. It calls a submodel, prior, to generate the gradient mu and intercept c from a normal distribution, and the noise around the line std from an uniform distribution. The output y is then computed from the normal distribution using a mean mu * x + c and standard deviation std.

The constraint "Observables" is short-hand for simultaneously specifying multiple Observable variables which have the same type. Here, we state that "y", "mu", "c", and "std" are Doubles that can be observed against, and we associate them with their appropriate distributions by using the OverloadedLabels hash syntax.

Our example program, although contrived, demonstrates our language's ability to combine different model definitions; this property becomes more obviously desirable when dealing with sophisticated, multi-level models. Moreover, our model is now ripe for both simulation *and* inference (assuming an effect handler for executing models is in place). *Simulation* could be performed passing an empty environment of observed values, allowing all random variables in the model to be newly sampled each iteration. Alternatively, one could simulate by providing an environment with a fixed gradient mu and noise std – the data generated would then be characterized by that line. *Inference* would work by passing a list of environments containing an expected output "y", each of which corresponds to a known input x. This would generate a trace of parameters mu and std whose distribution expresses the most likely latent parameters to give rise to the observed data.

3 Improving Our PPL Mechanism

Our current implementation provides a decent infrastructure for writing elementary probabilistic models in. Attempting to express more complex models, however, reveals that there are two fundamental mechanisms which we lack. In this section, we introduce affine environments (§ 3.1) and addresses (§ 3.2) to yield the revised, final definition of models (§ 3.3). This allows us to write more complex, recursive models, which we demonstrate in § 3.4.

3.1 Affine Environments

The method of using **Maybe** types for observable values is very straightforward, and initially, it seems sufficient for consolidating the effects of sampling and observing. Its shortcomings soon become clear when considering the two following problems which can be considered different sides of the same coin:

(1) It is currently possible for the value of an observed variable to be referenced and conditioned against multiple times in a model which often results in the model becoming ill-defined. In most situations, this would be considered as accidental behaviour resulting from an oversight by the user. The actual intention of the model hence becomes ambiguous.
(2) A probabilistic call can only be associated with a single observed variable whose value remains the same throughout the course of a model. Following on from the first problem, the fact that we can recursively or iteratively observe becomes useless if we are incapable of conditioning against different values. This is massively detrimental to our ability to use models with higher-order functions such as replicate, fold, and map.

To resolve this, we acknowledge two objectives.

3.1.1 List Environments The first objective is that an observed variable must be able to correspond to multiple observed values, each of which has a known position. We achieve this by first changing our environment such that each observed variable is associated with a value of type **List** a rather than **Maybe** a. Hence, we now instead use following type family AsLists, which maps the **List** type constructor over the values in an open product. We also redefine the Observable type class to incorporate this:

```
type family AsLists (as :: [k]) = (bs :: [k]) | bs -> as where
  AsLists ((x :> v) : xvs)  = ((x :> [v]) : AsLists xvs)
  AsLists '[] = '[]

class Lookup (AsLists xvs) x [v] => Observable xvs x v
```

```
instance Lookup (AsLists xvs) x [v] => Observable xvs x v
```

The empty list [] can be viewed as equivalent to Nothing and the non-empty list (x:xs) as equivalent to Just x. Although there are many data structures and forms of indexing values that one could use, lists capture the most general treatment of environments for our purpose: this is because probabilistic operations ultimately occur sequentially, and importantly, the execution trace of observation calls in a probabilistic program must transpire in a known order.

3.1.2 Affine Effects The second objective is that referencing an observed variable must permanently consume one of its values – this is analogous to affine types, where a resource can be used at most once. Haskell does not support an affine type system, and it would be excessive to require one. However, we can simulate affine types as an *effect* by defining an "affine reader":

```
data AffReader env a where
  Ask :: Observable env x v => Var x -> AffReader env (Maybe v)

ask :: (Member (AffReader env) ts, Observable env x v)
    => Var x -> Freer ts (Maybe v)
ask var = Free (inj $ Ask var) Pure
```

AffReader can be seen as a refinement of our previous generic Reader type, and implicitly specifies that the environment must be an open product where its values are lists. The Ask operation no longer returns the entire environment, but instead takes a variable name to look up its corresponding list value and return the head element if it exists. This safely restricts which observed values can be accessed at any one time. The actual discarding of consumed values (and maintenance of the environment) in response to Ask operations is entrusted to the AffReader effect handler (implemented in § 4.1) which updates the environment with the tail of the list it accessed.

3.2 Addressing Probabilistic Operations

Probabilistic languages require unique addresses to be assigned to each dynamic occurrence of a probabilistic operation – this is mainly for the correctness and implementation of generic inference algorithms [31], so that operations between different program executions may be corresponded. A secondary benefit of addresses is the ability to identify and extract specific random variables from the execution trace.

Our addressing mechanism allows observed variable names to be used as tags to distribution calls. We can easily extend the current implementation to incorporate addresses by allowing distributions, Dist, to take an extra argument of type **Maybe** Tag.

```
type Tag = String
data Dist a where
  NormalDist :: Double -> Double -> Maybe Double
             -> Maybe Tag -> Dist Double
```

Of course, tags only identify static occurrences of probabilistic operations – each dynamic occurrence requires its own indexing, and this is taken care of in the implementation of distribution effect handlers (§ 4.2).

3.3 Models and Distribution Operations, Revised

With the introduction of the AffReader effect, we update the definition of Model to give its final form:

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

This simply replaces the old effect, Reader (MRecord env), with the type AffReader env so that models rely on an affine environments of list values rather than immutable **Maybe** values.

We then redefine our smart constructors for primitive distributions to account for affine environments and addresses:

```
varToStr :: forall x. Var x -> String

normal :: forall env x ts. (Observable env x Double)
       => Double -> Double -> Var x -> Model env ts Double
normal mu sigma var = Model $ do
  let tag = Just (varToStr var)
  obs <- ask var
  send (NormalDist mu sigma obs tag)

normal' :: Double -> Double -> Model env ts Double
normal' mu sigma = Model $ do
  send (NormalDist mu sigma Nothing Nothing)
```

The changes are minor. The function normal now converts the observed variable name from a type-level string to the string value tag, and the associated observed value obs is accessed directly from the environment. When no observed variable is provided, the function normal' simply sets its tag as Nothing.

3.4 Example: Hidden Markov Model

The addition of affine environments lets us define models with recursive or iterative structure; they can now be used safely with higher-order functions. An excellent example of this is a hidden markov model (HMM) [1], illustrated in Figure 4.

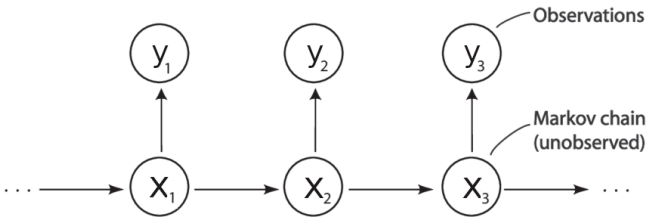

Figure 4: Hidden Markov Model

The idea is that there are a series of latent states $\mathbf{x}_i$ which are related in some way to a series of observable states $\mathbf{y}_i$. The objective is to learn about $\mathbf{x}$ given $\mathbf{y}$. A HMM is defined by two sub-models:

(1) A transition model describes how the latent states $\mathbf{x}_i$ are transitioned between:

```
transitionModel :: Double -> Int -> Model env ts Int
transitionModel transition_p x_{i-1} = do
  dX <- boolToInt <$> bernoulli ' transition_p
  return (dX + x_{i-1})
```

Here, the variable dX is drawn from a Bernoulli distribution, converted to an integer, and added to the previous latent state to yield $\mathbf{x}_i = \mathbf{x}_{i-1} +$ dX.

(2) An observation model projects a latent state $\mathbf{x}_i$ to an observable state $\mathbf{y}_i$:

```
observationModel :: (Observable env "y_i" Int)
                 => Double -> Int -> Model env ts Int
observationModel observation_p x_i = do
  binomial x_i observation_p #y_i
```

This states that $\mathbf{y}_i$ is related to $\mathbf{x}_i$ via the Binomial distribution.

These sub-models can then be *combined* to define a HMM for a single node:

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

Furthermore, this HMM can be functionally composed to create a chain of nodes. We do this by creating a list of HMMs and folding over them with kleisli composition (>=>).

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

This program fully defines a HMM. By assigning a list of observed data to the variable $\mathbf{y}_i$, the AffReader effect allows us to recursively condition against each of the values in $\mathbf{y}_i$ in chronological order.

Although this only captures a specific example, there are a multitude of cases in which it is pragmatic to use models with higher-order functions, e.g. conditioning a matrix against a one-dimensional distribution (using replicateM), or mapping a model against a list of different parameters (using mapM).

4 Inference: Basic Handlers

Perfoming simulation or inference can now be achieved by defining appropriate effect handlers which assign semantics to the syntactic operations of a model. Every model initially only consists of the distribution effect, Dist, and the affine reader effect, AffReader env (as defined in § 3.3). Both of these effects can actually be handled in a universal way before even having to decide how a model will be run and what specific algorithm will be used to run it.

In this section, we implement these handlers so that all observed variable requests are managed (§ 4.1) and all distributions are concretised as either sample and observe operations (§ 4.2). The resulting composition of these handlers (§ 4.3) will reconstruct a probabilistic program into a form which is ready to be interpreted by all kinds of inference algorithms.

4.1 Handling AffReader

We start by defining the handler for the AffReader type; as mentioned in § 3.1, this must ensure that observed values are permanently consumed when read. This is given as runAffReader:

```
runAffReader :: forall env ts a. OpenProduct (AsLists env)
             -> Freer (AffReader env ': ts) a -> Freer ts a
runAffReader _   (Pure a)   = return a
runAffReader env (Free u k) = case decomp u of
  Right (Ask var) -> do
    let ys   = getOP var env
        y    = safeHead ys
        env' = setOP var (safeTail ys) env
    runAffReader env' (k y)
  Left u' -> Free u' (runAffReader env . k)
```

This function is run by providing an initial environment env containing of a list of values for each observed variable. Upon receiving an Ask request to read from an observed variable var, we acquire the corresponding list ys. The environment is then updated with the tail of the list, and the head element y (if it exists) is returned as the observed value to the continuation k.

4.2 Handling Distributions

The purpose of the distribution handler is twofold: to map distributions to either sample or observe effects, and to provide a dynamic address to each probabilistic operation. We introduce the type of addresses below, as well as the effects of sampling and observing:

```
type Addr = (Tag, Int)
data Sample a where
  Sample :: Dist a -> Addr -> Sample a
data Observe a where
  Observe :: Dist a -> a -> Addr -> Observe a
```

An address, Addr, consists of a tag along with an integer representing its occurrence in the program. The type Sample takes a distribution to sample from and an address. The type Observe takes a distribution, an observed value, and an address.

The distribution handler, runDist, is then implemented indirectly by calling the function loop:

```
runDist :: forall ts a. (Member Sample ts, Member Observe ts)
        => Freer (Dist : ts) a -> Freer ts a
runDist = loop 0 Map.empty
  where
  loop :: (Member Sample ts, Member Observe ts)
    => Int -> Map Tag Int -> Freer (Dist : ts) a -> Freer ts a
```

The function loop traverses and handles all the distribution calls in the program. This also takes and maintains: a counter, used to assign default tags to untagged distributions, and a tagMap, used to track the number of dynamic occurrences of a tag in a model.

```
1  loop :: (Member Sample ts, Member Observe ts)
2       => Int -> Map Tag Int -> Freer (Dist : ts) a -> Freer ts a
3  loop _ _ (Pure a) = return a
4  loop counter tagMap (Free u k) = case decomp u of
5    Right d ->
6      let tag       = fromMaybe (show counter) (getTag d)
7          tagIdx    = Map.findWithDefault 0 tag tagMap
8          tagMap'   = Map.insert tag (tagIdx + 1) tagMap
9      in case getObs d of
10       Just y  -> do x <- send (Observe d y (tag, tagIdx))
11                     loop (counter + 1) tagMap' $ k x
12       Nothing -> do x <- send (Sample d (tag, tagIdx))
```

```
13                                  loop (counter + 1) tagMap' $ k x
14      Left u' -> Free u' (loop counter tagMap . k)
```

On encountering a distribution operation in **Right** d, we first determine its tag by extracting it from the distribution via getTag d – if this is Nothing, we set it to the string-form of the counter (line 6). The number of occurrences of this tag, tagIdx, is looked up from the tagMap and set to 0 if it has not yet occurred (line 7); the tagMap is then updated to increment the number of occurrences (line 8). Finally, we attempt to retrieve the observed value using getObs d (line 9). If this exists, then we inject an Observe operation into our model, otherwise, a Sample operation is injected instead.

4.3 Composing Handlers: AffReader and Dist

For clarity, we give the type definition for the composition of these two handlers below as the function runInit:

```
runInit :: (Member Observe ts, Member Sample ts)
        => OpenProduct (AsLists env)
        -> Model env (Dist : AffReader env : ts) a
        -> Freer ts a
runInit env = runAffReader env . runDist . runModel
```

The result of running this on a model is a fully-addressed probabilistic program where distributions have been replaced with concrete Sample and Observe calls.

5 Inference As Effect Handler Composition

The general implementation of simulation and inference algorithms in probabilistic programming can be boiled down to the semantics that they assign to Sample and Observe operations. This concept cooperates extremely naturally with algebraic effect handlers. *Theoretically*, one would only ever need to define two effect handlers for any particular execution of the model: runSample and runObserve. This idea is short-lived however, as we encounter algorithms which demand further side-effects on top of sampling and observing. In this section, we illustrate how the program transformation described in § 4.3 can be taken and intelligently composed with a range of other effect handlers, giving rise to simulation (§ 5.1) and inference (§ 5.2) over models.

5.1 Simulation

Simulation can be considered the most basic form of model execution. It simply runs the model as a generative process to return its output, using observed data when provided and otherwise drawing new samples. This can be written rather easily:

For Observe requests, no conditioning side-effects are performed; the handler runObserve simply needs to return the observed value y to its continuation k:

```
runObserve :: Freer (Observe : ts) a -> Freer ts a
runObserve (Pure a)     = return a
runObserve (Free u k) = case decomp u of
  Right (Observe d y addr) -> runObserve (k y)
  Left u'                  -> Free u' (runObserve . k)
```

For Sample requests, the handler runSample returns a sampled value x from the provided distribution d. In our implementation, the function sample uses an external statistics library to accomplish this:

```
sample :: Dist a -> IO a

runSample :: Freer '[Sample] a -> IO a
runSample (Pure a)     = return a
runSample (Free u k) = case prj u of
  Just (Sample d addr) -> do x <- sample d
                             runSample (k x)
  _                    -> error "Impossible: Nothing can occur"
```

Running this dispatches the final effect in the Freer monad to produce an **IO** effect. Hence, runSample always needs to be executed as the last effect handler where only Sample operations are allowed to occur in the program.

We can now give the complete definition for simulation below:

```
runSimulate :: (ts ~ '[AffReader env, Dist, Observe, Sample])
            => OpenProduct (AsLists env) -> Model env ts a -> IO a
runSimulate env
  = runSample . runObserve . runDist . runAffReader env . runModel
```

This implementation depicts how the logic of model execution can be made transparent by decomposing it into a pleasingly modular system. Moreover, this approach allows for fine-grained modifications and extensions which we discuss in § 5.2.

We demonstrate simulation using a Hidden Markov Model (similar to § 3.4) to model the spread of disease during an epidemic – this is known as an SIR model [30]. The observed states, RecInf, represent the *recorded* amount of infected people. The latent states, LatentSt, characterize three variables: the *actual* amount of susceptible, infected, and recovered people:

```
type RecInf    = Int
data LatentSt = LatentSt { sus :: Int, inf :: Int, recov :: Int }
```

The function hmmSIR defines our actual model. It takes as arguments the number of time-steps and the initial latent state, and produces a series of latent and observed states. How these states change over time is determined by three parameters: ρ, β, and γ.

```
hmmSIR :: (Observables env '["ρ", "β", "γ"] Double,
           Observable env "recInf" Int)
  => Int -> LatentSt -> Model env ts ([LatentSt], [RecInf])
```

To simulate from this, we first construct an environment of observable variables (using some pre-defined operators as syntactic sugar): this fixes values to our parameters ρ, β, and γ, but assigns no values to reflnf as this is only used during inference. Next, we set our initial latent state to a population of 763 people who are all susceptible to disease. The entire model can then be handled by calling runSimulate.

```
let env = (#ρ @= [0.3] <: #β @= [0.7] <: #γ @= [0.009] <:
           #recInf @= [] <: nil)
    latentSt = LatentSt { sus = 763, inf = 0, recov = 0 }
in  runSimulate env (hmmSIR 100 latentSt)
```

The output simulation from this program is visualized in Figure 5.

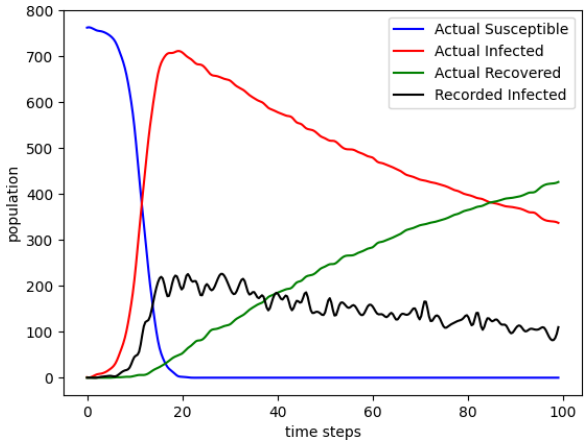

Figure 5: SIR Hidden Markov Model - Simulation

5.2 Inference

Approximative Bayesian inference attempts to learn the posterior distribution of a model's parameters given some observed data. Unlike simulation which is straightforward, inference algorithms require more complex mechanisms to keep track of information such as sample traces, probability mappings, and model reparameterization – attempting to capture all of this inside just runSample and runObserve would lead to a rather large and unwieldy implementation. It is also important to note that many inference algorithms such as Monte-Carlo methods can be said to decompose into more basic building blocks, and through modifying and combining these components, different variants of algorithms may be constructed. It is therefore crucial that our implementation can express this notion well in a compositional manner.

Our approach uses effect handlers to perform a series of further composable program transformations on the model; one can view this as iteratively reconstructing a probabilistic program by mutating existing effects into new ones. The result of this process is a model embedded into the context of a specific inference algorithm; we demonstrate this for Likelihood Weighting, and single site Metropolis-Hastings [31].

5.2.1 Likelihood Weighting is a form of importance sampling [7] which approximates the posterior distribution with a set of weighted samples. The idea is that whenever we draw a sample, this is equivalent to generating a proposed model parameter from the prior distribution. When provided observed data, we compute the log-probability of these proposed parameters generating that data; the sum of these log-probabilities represent the total log-likelihood. Performing this over multiple iterations gives us a trace of sample maps of proposed parameters and their likelihoods.

First, a data structure is required to record the sample map during a model execution. We define this as SMap, a map from addresses to sampled values:

```
type PrimVal = '[Int, Double, Bool, String]
type SMap    = Map Addr (OpenSum' PrimVal)
```

This map needs to accommodate any value which can be produced from a primitive distribution (given by PrimVal), however maps are monomorphic in their value type. We hence use OpenSum' which is a variant of OpenSum but instead contains types of kind $(*)$.

To be able to update a sample map, we introduce the State effect with a single operation, Modify. The function updateSMap then modifies the map by inserting a new address and value:

```
data State s a where
  Modify :: (s -> s) -> State s ()
modify f = Free (inj $ Modify f) Pure

updateSMap :: (Member (State SMap) ts, Member v PrimVal)
           => Addr -> v -> Freer ts ()
updateSMap addr v = modify (Map.insert addr (inj v))
```

The core idea is then to perform a program transformation to our model to embed it in the context of Likelihood-Weighting. This is defined as the function transformLW, where we can see from its type signature that it does not discharge any effects in ts:

```
1  transformLW :: (Member (State SMap) ts, Member Sample ts)
2              => Freer ts a -> Freer ts a
3  transformLW (Pure a)    = return a
4  transformLW (Free u k) = case prj u of
5    Just (Sample d addr) -> case d of
6      DistInt (Just d)     ->
7        Free u (\x -> do updateSMap addr (unsafeCoerce x :: Int)
8                         transformLW (k x))
9      DistBool (Just d)    ->
10       Free u (\x -> do updateSMap addr (unsafeCoerce x :: Bool)
11                        transformLW (k x))
12     ...
13   _ -> Free u (transformLW . k)
```

For every Sample operation, this injects a State SMap effect that stores the sampled value into our map (lines 7 & 10). Note that because the type of the sampled value x passed to the continuation is existentially quantified, we need to pattern match on the sampled distribution in order to correctly determine its concrete type. Here we use pattern synonyms such as DistInt and DistBool to cover all primitive distributions which generate those types; this allows us to safely coerce x and insert it into our map.

Now all that is left is to define handlers for Observe and Sample (we omit the implementation of runState). The handler runSample is implemented in the same way as for simulation (§ 5.1). However, runObserve now needs to accumulate and sum all the log probabilities of the observed values provided:

```
logProb :: Dist a -> a -> Double

runObserve :: Member Sample rs
           => Freer (Observe : rs) a -> Freer rs (a, Double)
runObserve = loop 0
  where
  loop p (Pure a)    = return (a, p)
  loop p (Free u k) = case decomp u of
      Right (Observe d y _) -> let p' = logProb d y
                               in  loop (p + p') (k y)
      Left u'               -> Free u' (loop p . k)
```

The complete definition of Likelihood Weighting is shown below:

```
runLW :: ts ~ '[AffReader env, Dist, State SMap, Observe, Sample]
      => OpenProduct (AsLists env) -> Model env ts a
      -> IO ((a, SMap), Double)
runLW env = runSample . runObserve
          . runState Map.empty . transformLW
          . runDist . runAffReader env . runModel
```

This function resembles that of runSimulate, except we also insert a handler composition on line 5 which embeds and handles an effect for recording sample maps. Running this on a model will yield the model's output, its sampled values, and the log likelihood of its sampled values giving rise to the observed data env.

We demonstrate Likelihood-Weighting on our previous linear regression model in § 2.5:

```
linRegr :: (Observables env ["y", "mu", "c", "std"] Double)
        => Double -> Model env ts (Double, Double)


let mkEnvY yv = (#y @= yv) <: (#mu @= []) <:
                (#c @= []) <: (#std @= []) <: nil
    xs        = [0 .. 100]
    ys        = map mkEnvY (map (* 3) [0 .. 100])
in lw linRegr xs ys
```

First we define a list of model inputs xs and their corresponding observed values ys such that they are related by $y = 3 * x$. Inference is then performed by calling a top-level function lw which performs multiple iterations of runLW for different model inputs and environments. This produces a trace of weighted samples; in Figure 6, we visualise the likelihoods of different samples for parameter mu, where values around mu = 3 clearly accumulate higher probabilities.

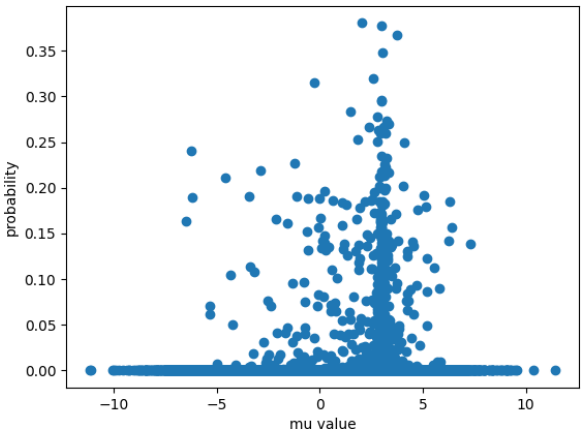

Figure 6: Linear Regression - Likelihood Weighting (mu)

Likelihood Weighting, however, loses effectiveness as the number of random variables we sample from grows. This is because every iteration, it works by freshly generating new samples for *all* Sample operations. Achieving a sample map with a high likelihood is dependent on being fortunate enough to sample an entire set of good values, and it becomes ambiguous as to which variables have contributed more. An alternative importance sampling method which offers a solution is Metropolis-Hastings.

5.2.2 Metropolis-Hastings (MH) keeps track of both a sample map *and* a log probability map. For each MH iteration, the idea is to first randomly pick a single proposal site α. When sampling, we only draw a new value if the address matches α, otherwise, we reuse the value from the previous iteration's sample map whenever possible; this allows us to incrementally propose new parameters. When both sampling and observing, we always compute their log-probabilities – these are used at the end of an MH iteration to decide whether to accept the proposed parameter. Performing this over multiple iterations produces a trace of sample maps representing the posterior distribution over the model's parameters.

We first define a new data structure LPMap to record the log-probabilities during model execution, and a function updateLPMap to update this map using the State effect:

```
type LPMap = Map Addr Double
updateLPMap :: Member (State LPMap) ts
            => Addr -> Dist a -> a -> Freer ts ()
updateLPMap addr d x = modify (Map.insert addr (logProb d x))
```

A model can then be transformed into the context of Metropolis-Hastings, given by transformMH:

```
transformMH :: (Member (State SMap) ts, Member (State LPMap) ts
                , Member Sample ts, Member Observe ts)
            => Freer ts a -> Freer ts a
transformMH (Pure a)    = return a
transformMH (Free u k) = case prj u of
  Just (Sample d addr) -> case d of
      DistInt (Just d) ->
        Free u (\x -> do updateSMap addr (unsafeCoerce x :: Int)
                         updateLPMap addr (unsafeCoerce x :: Int)
                         transformLW (k x))
      ...
  Just (Observe d y addr) ->
        Free u (\x -> do updateLPMap addr y
                         transformMH (k x))
  _ -> Free u (transformMH . k)
```

We inject State effects to 1) store any values generated from Sample operations (line 8), and 2) compute and store the log probabilities for both Sample and Observe operations (lines 9 & 13).

All we need to do now is define how Observe and Sample are interpreted. As all conditioning is taken care of by the State LPMap effect, the runObserve handler simply needs to return its observed value y – hence its implementation is the same as for simulation (§ 5.1). The runSample handler, however, needs to be implemented such that it reuses old samples whenever it can:

```
lookupSample :: SMap -> Dist a -> Addr -> Addr -> Maybe a

runSample :: Addr -> SMap -> Freer '[Sample] a -> IO a
runSample α sMap = loop
  where
  loop (Pure a)    = return a
  loop (Free u k) = case prj u of
    Just (Sample d addr) -> case d of
        DistInt (Just d) -> do
          x <- fromMaybe <$> sample d
               <*> lookupSample sMap d addr α
          (loop . k . unsafeCoerce) x
        ...
```

It now uses the function lookupSample: this returns Nothing if the current address matches the proposal site α or if it is not found in the previous sample map – in this case, we generate a new sample.

We can now give the complete definition of the Metropolis-Hastings handler as runMH:

```
runMH :: ts ~ '[ AffReader env, Dist, State SMap, State LPMap
               , Observe, Sample ]
      => OpenProduct (AsLists env) -> SMap -> Addr
      -> Model env ts a -> IO ((a, SMap), LPMap)
runMH env sMap α = do
  runSample α sMap . runObserve
  . runState Map.empty . runState Map.empty . transformMH
  . runDist . runAffReader env . runModel
```

In addition to the environment, env, this takes two extra arguments: the previous iteration's sample map, sMap, and the current proposal site α. The handler composition on line 7 embeds and handles the effects for recording sample and log-probability maps which are later returned in the final output.

This can then be used to implement a single iteration of the Metropolis-Hastings algorithm, mhStep:

```
1  type MHTrace a = [(a, SMap, LPMap)]
2  mhStep :: ts ~ '[ AffReader env, Dist, State SMap, State LPMap
3                  , Observe, Sample ]
4         => OpenProduct (AsLists env) -> Model env ts a
5         -> MHTrace a -> IO (MHTrace a)
6  mhStep env model trace = do
7    let (x, sMap, lpMap) = head trace
8    α_idx <- discrUniform 0 (Map.size sMap - 1)
9    let α = fst $ Map.elemAt α_idx sMap
10   ((x', sMap'), lpMap') <- runMH env sMap α model
11   let accept_ratio = accept α sMap sMap' lpMap lpMap'
12   u <- uniform 0 1
13   if accept_ratio > u
14     then return ((x', sMap', lpMap'): trace)
15     else return trace
```

This first extracts the previous iteration's sample and log-probability maps from the head of the accumulated trace (line 7). Then, we uniformly choose a proposal site α from the existing sample addresses (lines 8 - 9). These are used to execute the model using the runMH handler, which returns a new sample and log probability map (line 10). To decide whether these should be added to the trace, we compute an acceptance ratio by comparing them with the previous maps (line 11); if this ratio is larger than $u \sim \text{uniform}(0, 1)$, then we accept (lines 12 - 15).

We demonstrate Metropolis-Hastings on a topic model (also known as Latent Dirichlet Allocation), whose aim is to group similar word patterns to identify topics in a set of text documents. Each document has its own topic distribution, expressing how much a certain topic occurs in that document. Each topic distribution then has its own word distribution which specifies how likely certain words are to occur under that topic. This is visualised in Figure 7.

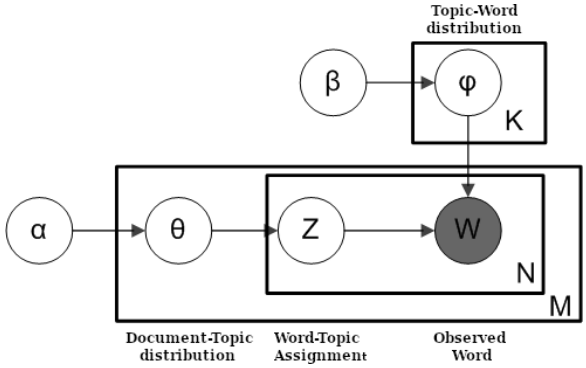

Figure 7: LDA

This diagram highlights well why the ability to express hierarchical models in a modular way is desirable; below, we give a simple outline of how the implementation looks in our language:

```
topicWordDist :: Observable env "w" String
  => [String] -> [Double] -> Model env ts String

docTopicDist :: Observable "θ" [Double]
  => Int -> Model env ts [Double]

docModel :: (Observables env ["ϕ", "θ"] [Double],
             Observable env "w" String)
  => Int -> [String] -> Model ts env [String]
```

To perform Metropolis-Hastings on this, we first define the variable doc as the document of words we are conditioning against; for simplicity, we use a list of strings containing only "DNA" and "evolution". We then state the full potential vocabulary of a document, and declare that there are two possible topics. Lastly, we call a top-level function mh on our model, which essentially executes mhStep for 100 iterations.

```
let mkRecTopic words = (#ϕ @= []) <: (#θ @= [])
                     <: (#w @= words) <: nil
    doc    = mkRecTopic ["DNA", "evolution", "DNA", "evolution"]
    vocab  = ["DNA", "evolution", "parsing", "phonology"]
    num_topics = 2
in  mh 100 (docModel num_topics) [vocabulary] [doc]
```

This returns us a trace of samples representing the posterior distribution over the model parameters. By randomly selecting a set of parameters from the trace, we can then visualize the predictive distribution of our model in Figure 8. This expresses how we believe topics and words are distributed over the document.

6 Related Work

Here we discuss relevant work with respect to our contributions, making observations about novelty and alternative approaches.

1. Our work achieves a model-based PPL for syntactically constructing a general-purpose probabilistic model from which multiple interpretations can be given rise to. Our embedding technique consists mainly of freer monads and a distribution effect.

Ścibior et al. also take approaches which can be considered similar to ours, but for different purposes. Their initial language encodes

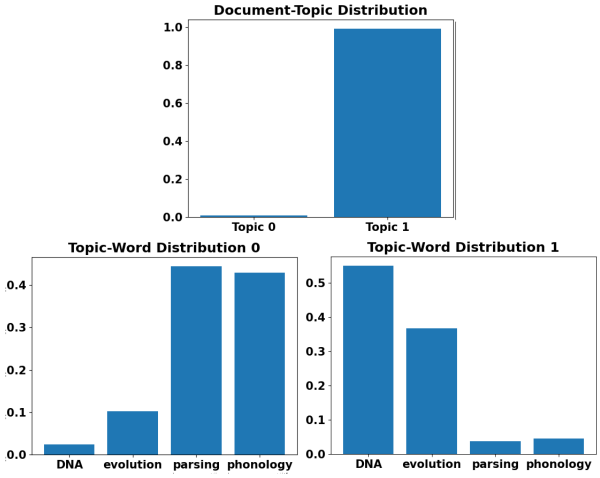

Figure 8: LDA - Metropolis-Hastings Predictive

distributions as an intermediate free monad representation (MonadBayes2016 [21]) as proof-of-concept that the monad abstraction can be used to construct probabilistic models whilst still offering good performance during inference. They also later incorporate free monad transformers but for the purposes of inference (MonadBayes [23]). Their work differs from ours in that their languages are query-based.

PPLs embedded in functional languages (Hakaru10 [11], Hansei [14], Hakaru [17]) often employ tagless-final shallow embedding Kiselyov [10] which encodes the syntax of the probabilistic language as a type class and its semantics as type class instances. These are fundamentally limited by having restricted compatibility with host language features; this results in having to redefine well-known constructs, such as arithmetic operations and lambda calculus terms, into the domain of their language. It is also commonly the case that these languages are not fully model-based, requiring multiple model versions having to be implemented for different interpretations.

PPLs which do achieve a proper model-based implementation do this by taking advantage of the macro-compilation features of their host language (Turing.jl [6], Gen.jl [5]) or by compiling to entirely different languages (Stan [3]).

2. We allow observed variables of models to be stated as type class constraints which can then be referenced inside the model via the AffReader effect and extensible records. This delays interpretation of probabilistic statements until observed data is provided.

Alternative solutions which model-based languages use include macro-compilation and default behaviour for missing function arguments (Turing.jl [6], Gen.jl [5]). A more common method is for the user to manually address any probabilistic operations and provide a mapping between addresses and observed data (WebPPL [8], Gen.jl [5], PyMC3 [20]). Extensible records have not previously been seen adopted as a solution.

3. We encode simulation and inference as effect handler composition to perform a series of program transformations over a model description; this effectively embeds a model into the context of a specific algorithm.

The notion of using algebraic effect handlers for probabilistic programming is still an emerging idea with little formal covering material. This has been discussed briefly as a workshop paper by Scibior and Kammar [22]; Moore and Gorinova [16] then provide an extended abstract reviewing the use of effect handling for composable program transformations in Pyro [2] and Edward2 [28]. We hope to contribute a concrete and detailed implementation of this method and an argument for its potential, especially in a strongly functional paradigm.

4. Models are first-class objects, and as a consequence, this means that models and primitive distributions can be treated similarly. They can be combined and composed monadically, using functional combinators such as kleisli composition, folding, mapping, etc. Most model-based languages (Turing.jl [6], PyMC3 [20], Stan [3]) are not capable of calling models from other models, and to the best of our knowledge, none allow models to be composed or used generically in higher-order functions.

7 Conclusion

The main focus of our work was to develop an embedding strategy for a probabilistic language in a functional paradigm which allows models to be constructed modularly. Our original ideals were that models should only need to be defined once and then interpreted for simulation/inference only when necessary. Moreover, models need to be first class citizens for them to be considered modular – one should be able to treat them as functions, compose them with other models, and use them freely with higher-order functions. Using algebraic effects and extensible data, we have managed to achieve both of the advantages that query-based and model-based languages offer whilst still allowing models to be written in an elegant, minimal way, as demonstrated across a variety of examples.

There are two related topics of future work which are orthogonal to the goals of this paper. The first is to investigate deeper into how effect handler techniques can be used to implement more sophisticated, compositional inference algorithms than Likelihood-Weighting and single-site Metropolis-Hastings, e.g. SMC [24], SMC^2 [4], PMMH [29]. The notion of using effect handlers to perform compositional program transformations appeared extremely natural to us in the context of probabilistic programming, and we were surprised that it was not a more popular discussion point; we believe this has interesting potential and we aim to properly explore its ability to design inference algorithms more modularly as compositional building blocks.

The second future goal is to consider how the performance of inference using effect handlers can be improved. Our language's performance was adequate for determining whether we could learn realistic and interesting hierarchical models, but would be impractical for cases involving very large data sets and complex higher-dimensional models. Of course, it is optimistic to hope to achieve the performance of specialised inference-driven languages, but to be competitive with the performance of general-purpose PPLs whilst still being able to gain full leverage of functional programming features would be considered a success.

References

[1] Leonard E Baum and Ted Petrie. 1966. Statistical inference for probabilistic functions of finite state Markov chains. *The annals of mathematical statistics* 37, 6 (1966), 1554–1563.

[2] 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

[3] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: a probabilistic programming language. *Grantee Submission* 76, 1 (2017), 1–32.

[4] Nicolas Chopin, Pierre E Jacob, and Omiros Papaspiliopoulos. 2013. SMC2: an efficient algorithm for sequential analysis of state space models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 75, 3 (2013), 397–426.

[5] 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)*. ACM, New York, NY, USA, 221–236. https://doi.org/10.1145/3314221.3314642

[6] 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

[7] Peter W Glynn and Donald L Iglehart. 1989. Importance sampling for stochastic simulations. *Management science* 35, 11 (1989), 1367–1392.

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

[9] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. 2014. Probabilistic programming. In *Future of Software Engineering Proceedings*. 167–181.

[10] Oleg Kiselyov. 2012. Typed tagless final interpreters. In *Generic and Indexed Programming*. Springer, 130–174.

[11] Oleg Kiselyov. 2016. Probabilistic programming language and its incremental evaluation. In *Asian Symposium on Programming Languages and Systems*. Springer, 357–376.

[12] Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. *ACM SIGPLAN Notices* 50, 12 (2015), 94–105.

[13] Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible effects: an alternative to monad transformers. *ACM SIGPLAN Notices* 48, 12 (2013), 59–70.

[14] Oleg Kiselyov and Chung-chieh Shan. 2009. Embedded probabilistic programming. In *IFIP Working Conference on Domain-Specific Languages*. Springer, 360–384.

[15] Daan Leijen. 2005. Extensible records with scoped labels. *Trends in Functional Programming* 6 (2005), 179–194.

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

[17] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *International Symposium on Functional and Logic Programming*. Springer, 62–79.

[18] Gordon Plotkin and John Power. 2003. Algebraic operations and generic effects. *Applied categorical structures* 11, 1 (2003), 69–94.

[19] Gordon D Plotkin and Matija Pretnar. 2013. Handling algebraic effects. *arXiv preprint arXiv:1312.1399* (2013).

[20] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* 2 (2016), e55.

[21] Adam Ścibior, Zoubin Ghahramani, and Andrew D Gordon. 2015. Practical probabilistic programming with monads. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. 165–176.

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

[23] Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. 2018. Functional programming for modular Bayesian inference. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–29.

[24] Adrian Smith. 2013. *Sequential Monte Carlo methods in practice*. Springer Science & Business Media.

[25] Josef Svenningsson and Emil Axelsson. 2015. Combining deep and shallow embedding of domain-specific languages. *Computer Languages, Systems & Structures* 44 (2015), 143–165.

[26] Wouter Swierstra. 2008. Data types à la carte. *Journal of functional programming* 18, 4 (2008), 423–436.

[27] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and implementation of probabilistic programming language anglican. (2016), 1–12.

[28] Dustin Tran, Matthew D. Hoffman, Dave Moore, Christopher Suter, Srinivas Vasudevan, Alexey Radul, Matthew Johnson, and Rif A. Saurous. 2018. Simple, Distributed, and Accelerated Probabilistic Programming. In *Neural Information Processing Systems*.

[29] Tuyet Vu, Ba-Ngu Vo, and Rob Evans. 2014. A particle marginal Metropolis-Hastings multi-target tracker. *IEEE Transactions on Signal Processing* 62, 15 (2014), 3953–3964.

[30] Howard Howie Weiss. 2013. The SIR model and the foundations of public health. *Materials matematics* (2013), 0001–17.

[31] David Wingate, Andreas Stuhlmüller, 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*. JMLR Workshop and Conference Proceedings, 770–778.

[32] Nicolas Wu and Tom Schrijvers. 2015. Fusion for free. In *International Conference on Mathematics of Program Construction*. Springer, 302–322.