Composable, Modular Probabilistic Models

MINH NGUYEN*, University Of Bristol, United Kingdom

Probabilistic programming languages (PPLs) allow one to construct statistical models to describe certain problem domains, and then simulate data or perform inference over them [5]. 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 project describes a DSL for modularly defining probabilistic models which are combinable and composable, and can be reused for both simulation and inference.

1 PROBLEM AND MOTIVATION

To make the workflow of using probabilistic languages as convenient as possible, there are two significant challenges to engage with:

(1) How easy is it to specify and iterate through different models. It is common to quickly experiment with many different definitions of models before considering if/how they will be used at all.
(2) How easy is it to combine and compose different models. This is essential for modular development of models and hierarchical modeling 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 PPLs.

2 BACKGROUND AND RELATED WORK

What separates a probabilistic language from a non-probabilistic one is the ability to:

- Sample - To draw a value from a distribution.
- Observe - To condition against the probability of a distribution giving rise to an observed value.

Given these two probabilistic constructs, one then expects to be able to:

(1) Describe a model for a problem. This is specified by its parameters and a description of how it generates data, usually expressed as a mathematical relationship between random variables.
(2) Simulate data from a model given a fixed set of model parameters.
(3) Infer the parameters of a model by providing observed data to condition against.

PPLs can be categorized into being either query-based or model-based languages.

Query-based languages (Anglican [11], WebPPL [4], MonadBayes [10]) operate via the user writing a probabilistic query as a normal function but with the freedom to explicitly use sample and observe - this allows one to directly express probabilistic computations for a given problem. Queries can be easily combined (they can call other queries) and sometimes even functionally composed.

The explicit use of sample and observe, however, means that query-based PPLs can only express "instances" of models which are specific to how they will be used - either for simulation *or* inference. This leads to having to redefine the same model for each use case. It hence becomes frustrating to iterate through and evolve models whilst having to maintain each model interpretation.

Model-based languages (Turing.jl [3], Gen.jl [2], PyMC3 [9], Stan [1]) 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 description. This allows one to independently specify and evolve models, and then later perform simulation and inference on the same model definition.

A significant 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. The result is often that models have to be monolithically defined and can not be manipulated by functional constructs such as higher-order functions and functional combinators.

*Supervised by Dr Meng Wang, Dr Roly Perera. ACM number: 0619104. Graduate category. min.nguyen@.bristol.ac.uk

3 APPROACH AND UNIQUENESS

This project describes a shallow embedded model-based PPL for modular, composable models. The design relies on free monad transformers, type families, type classes, and extensible records, where the implementation is demonstrated in Haskell. The final type definition of a model is given below:

```
type Model s a = FreeT Dist (Reader (MRecord s)) a
```

The rest of this section introduces the components of this definition step-by-step and describes why they were found necessary in the construction of a model.

3.1 Free monad transformers

The first ideal of the language is that a model should be a syntactic description of the relationships between random variables, where simulation and inference are mechanisms which provide semantics to the model. To achieve this separation of concerns, *free monad transformers* [6], FreeT f m a, are used as a means of constructing a syntactic tree, where the structure of its nodes are described by some functor f and embellished with some monadic effect m.

```
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)) }
```

What is left to be decided are the choice of functor f and monad m in the free monad transformer.

3.2 Distributions as functor f

The free monad's functor f completely determines the available information given when deciding how to provide semantics to the model. The shape of our nodes f is described by the functor Dist where each of its constructors represents a primitive distribution.

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

Each distribution constructor takes the following arguments, in order:

(1) Some distribution parameters.
(2) A value of type **Maybe** a, expressing the presence of an observed value to condition against. This effectively unifies the syntax of sample and observe, and decides which one is performed.
(3) A continuation, taking a value the distribution produces to some type a - this lets Dist to be a functor whilst still encoding type information about what values each distribution generates.

3.3 Reader and Extensible Effects as monad m

Whether the user provides observed data to the model decides when sampling and observing occur. The model should avoid taking observed data as explicit function arguments, as this detracts from the model existing independently from the notions of simulation and inference. The Reader env monad is hence used as the effect m in our free monad tree, where its environment contains all observable variables in the model that can be conditioned against.

Forcing each model to have its own fixed reader environment, however, prohibits the ability to combine and compose models. Rather, the environments of models should be kept abstract and only require that certain variables relevant to the model can be referenced. This can be achieved via extensible records [7] and type families. The type MRecord s defines an open record where s is a heterogeneous list of associations between field names and **Maybe** types. The typeclass HasVar s k v then states that a value of type **Maybe** v can be looked up with key k from the record MRecord s.

3.4 Example Program: Hidden Markov Model (HMM)

Below depicts how a HMM [8] can be written in this language. A HMM is defined by two sub-models:

(1) A transition model describes how latent states x are transitioned between. The variable dX is drawn from a Bernoulli distribution (using a smart constructor), where **Nothing** is explicitly passed to indicate the intention to always sampled and never observe.

```
transitionModel :: Double -> Int -> Model s Int
transitionModel transition_p x_prev = do
  dX <- boolToInt <$> bernoulli transition_p Nothing
  return (x_prev + dX)
```

(2) An observation model projects a latent state x to an observable state y. The variable y_data of type **Maybe Int** is referenced from the constraint (HasVar s "y_data" **Int**), and its value decides whether the Binomial distribution is sampled from or observed against.

```
observationModel :: (HasVar s "y_data" Int) => Double -> Int -> Model s Int
observationModel observation_p x = binomial x observation_p y_data
```

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

```
hmm :: (HasVar s "y_data" Int) => Double -> Double -> Int -> Model s Int
hmm transition_p observation_p x_prev = do
  x <- transitionModel transition_p x_prev
  y <- observationModel observation_p x
  return x
```

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

```
hmmNSteps :: (HasVar s "y_data" Int) => Double -> Double -> Int -> (Int -> Model s Int)
hmmNSteps transition_p observation_p n =
  foldl (>=>) return (replicate n (hmm transition_p observation_p))
```

4 CONTRIBUTIONS AND RESULTS

In general, this contributes a language for defining probabilistic models in a strongly-typed, functional paradigm. The following contributions and observations about novelty are noted:

- This achieves a PPL which separates the syntactic construction of a probabilistic model from the semantics of simulation and inference through using free monad transformers with the distribution functor. Ścibior [10] has incorporated free monad transfomers in probabilistic programming, but to implement query-based inference algorithms. Other approaches include macro-compilation [2, 3], or compilation to entirely different languages [1].
- Observable variables of models can be stated minimally as type class constraints and then easily referenced in the model, via extensible records and type families/classes. This isolates details about inference from the term-level and delays interpretation of probabilistic statements until an environment is provided to the Reader monad. Other solutions include macro-compilation [2, 3] or manual addressing of probabilistic statements and providing mappings between addresses and observed data [2, 4, 9]. Extensible records have not previously been seen adopted as a solution.
- Models can be combined monadically. As a consequence, this means that models and primitive distributions can be treated similarly - a feature that is highly lacking in existing model-based languages. Most languages [1, 3, 9] are not capable of calling models from other models. Models can also be composed with functional combinators such as kleisli composition, folding, mapping, etc. This has not been found possible in other current model-based languages.

REFERENCES

[1] 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.

[2] 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

[3] 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

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

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

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

[7] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. 96–107.

[8] Lawrence Rabiner and Biinghwang Juang. 1986. An introduction to hidden Markov models. *ieee assp magazine* 3, 1 (1986), 4–16.

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

[10] 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.

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