GraPPa: Spanning the Expressivity vs. Efficiency Continuum *

Edwin Westbrook Chad Scherrer Nathan Collins Eric Mertens
Galois Inc.
{westbrook,chad.scherrer,conathan,emertens}@galois.com

1. Introduction

Probabilistic Programming Languages (PPLs) are a powerful approach to enable users without significant machine learning expertise to apply machine learning algorithms and techniques to problems in their domains of interest. PPLs facilitate the separation of a particular machine learning problem — the "what" — from the "how" of the particular algorithms or techniques needed to solve the problem. This separation allows *domain experts*, or users with expertise in the specifics of the problem being solved, to focus on describing the important details of a problem, while the solution can be written by someone who has expertise with probability and machine learning.

A machine learning problem is described in a PPL by providing a set of data along with a *generative model* that hypothesizes how the data was created. A generative model represents a joint distribution over a set of random variables, some corresponding to the data and others to unobserved "model parameters". To make generative models easier to write and understand, they models are written as random processes that read from random variables in sequence. A machine learning problem is then solved by applying inference methods. *Inference methods* are algorithms that try to answer questions related to how a set of data was generated by a given model, such as how likely it was that the given model generated the given data or what values are likely for the unobserved model parameters.

An important trade-off in the design of PPLs is between the expressiveness of the language of generative models and the power and efficiency of the inference methods. On one end of the spectrum are languages like Anglican (Wood et al. 2014) and the Haskell monad-bayes package (Ścibior et al. 2015), where generative models can contain any sort of random variable, representing any arbitrary random process. This generality provides a very expressive language for generative models, but, because very little can be known *a priori* about the arbitrary random variables in these models, they must be treated as "black boxes", thereby limiting the set of inference methods that can be used. On the other end of the spectrum, the Stan language (Carpenter et al. 2016) restricts models to range over a fixed, finite set of continuous variables, which allows efficient specialization to hybrid Monte-Carlo algorithms like Hamiltonian sampling and No U-Turn sampling (NUTS).

This trade-off between expressiveness and efficiency requires users to choose a different language, and associated toolchain, depending on the particular problem they are trying to solve. Additionally, even knowing where a particular problem lies on the continuum between expressiveness and efficiency, or where it is likely to lie as the problem evolves, can require a non-trivial amount of expertise and understanding of machine learning, which violates the original promise of separating the "what" and the "how."

In this talk, we will describe ongoing work on GraPPa, the Galois Probabilistic Programming language, that addresses this concern. GraPPa, which is implemented as an embedded domain-specific language (EDSL) in Haskell, is a single PPL that allows users to choose where each model lies on the continuum between expressiveness and efficiency, simply by choosing what sorts of random variables to use in a model. *The key technical idea that enables this approach is an encoding, in the type of each model, of the set of random variables and associated distributions used in that model*. This approach is compositional, meaning that a model with random variables in one set can be combined with a model with random variables in another set, and the type of the resulting model will contain the union of the two sets.

* This work supported by DARPA (contract number FA8750-14-C-0003)

2. Generative Models in GraPPa

Generative models are defined in GraPPa using a variant of the free monad construction, where a model is an element of the initial algebra of two operations (in addition to "return" and "bind"): the "sample" operation, which draws a value from a distribution; and the "score" operation, which associates a probability with the current execution. Initiality allows us to map a GraPPa model into any monad — that is, into any well-defined computation semantics — simply by giving interpretations for the two operations. A theoretical benefit of this approach is that it allows us to define a straightfoward semantics for GraPPa models as measures over traces, which we discuss briefly below. As a practical benefit, initiality allows us to "run" a model in many different ways, by mapping to different monads, such as the state monad. This running is how we define inference methods.

The key idea in the GraPPa design that allows it to express where a model falls in the continuum between model expressiveness and specialization is that the type of a generative model is parameterized by a Haskell *constraint function*, that expresses a set of constraints on the distributions used in the model. This allows the set of distributions used in a model to be encoded in its type.

In more detail, the GraPPa definition of generative models is summarized in Figure 1. The first few lines define the notion of distribution over a random variable. Line 2 starts by defining a Haskell type family, or type-level function, `Support`, where intuitively `Support d` gives the type of the support of distributions of type `d`. Lines 5 – 6 define the `HasPDF` typeclass. This captures the constraint that a distribution type `d` has an associated probability density function (PDF), by requiring there to be an associated function, `density`, that gives the density of a distribution at a point in the support. Following standard convention, densities and probabilities are represented in log-space, which is represented by the Haskell type `LogFloat`. Lines 9 – 12 then show how to define one particular distribution type, `Normal`, with support type `Double`. A `Normal` distribution over `Doubles` has a PDF, though we omit the definition here for succinctness.

Lines 15 – 18 define the type `ModelOp c a`, which captures the primitive operations that can be used in a model. These include operations for sampling from a distribution and for incorporating a

```
1  -- Defines the support type of a distribution, i.e., the space it samples from
2  type family Support (d :: *) :: *
3
4  -- Defines a distribution as having an associated density function
5  class HasPDF d where
6    density :: d -> Support d -> LogFloat
7
8  -- The normal distribution; other distributions are similar...
9  data Normal = Normal Double Double
10 type instance Support Normal = Double
11 instance HasPDF Normal where
12   density (Normal mu sigma) r = ...
13
14 -- Primitive operations in a model
15 data ModelOp (c :: * -> Constraint) a where
16   MOpSample :: c d => d ->
17                ModelOp c (Support d)
18   MOpScore :: LogFloat -> ModelOp c ()
19
20 -- Models are either done, or are an op followed by a continuation
21 data Model (c :: * -> Constraint) a where
22   ModelDone :: a -> Model c a
23   ModelStep :: ModelOp c b ->
24                (b -> Model c a) -> Model c a
25
26 -- Apply an operation in a model
27 appOp :: ModelOp c a -> Model c a
28 appOp op = ModelStep op return
```

Figure 1. GraPPa Generative Models (Simplified Version)

probability score into a computation. The type `a` associated a return type with an operation: sampling operations have the support type of the supplied distribution as their type, while scoring operations have unit type.

The key novelty in the design of GraPPa is the `c` argument in the `ModelOp` type. This has kind `* -> Constraint`, meaning that `c d` for any type `d` is a constraint on `d`. This `c` argument is used to constrain the types of the distributions used for sampling; e.g., the type `ModelOp HasPDF a` can only sample from distributions with associated PDFs.

Lines 11 – 17 define the notion of generative model used in GraPPa. This is a version of the free monad, where every computation is either done, with a final value, or it is an intermediate step, with a `ModelOp` that is waiting to be interpreted and a continuation that will determine the next `Model` depending on the value given for that `ModelOp`. These options are represented with the `ModelDone` and `ModelStep` constructors, respectively. Note that, for efficiency reasons, GraPPa actually uses an optimized, continuation-based version of the free monad (Kmett 2011), which we ignore here for simplicity. Although we omit the `Monad` instance for how to compose `Model`s, lines 27 – 28 show how to apply `ModelOp`s; this is standard in the free monad literature.

One way to view an element of the `Model` type is as a DFA, where a `ModelDone` represents a terminal state and a `ModelStep` represents a non-terminal state that will transition to a new state depending on the value of the given `ModelOp`. Each (finite) run of this DFA is a trace of successive values for the "sample" operations, along with a final value for the terminal state. Assuming that each of the distributions in a "sample" operation associates a well-defined probability to each element of its support, whether or not that probability is computable with a `HasPDF` instance, each such trace can also be associated with a probability, by multiplying these associated probabilities of the "sample" operations along with the probabilities given by the "score" operations. Technically speaking, this defines a measure and not a probability distribution, since score operations could make the total probability not sum to 1.

```
1  -- The type of a random data or parameter variable of type a
2  data Var a = VData a | VParam
3
4  -- Use the value of a variable drawn from a given distribution
5  use :: (c d, HasPDF d) => Var (Support d) ->
6         d -> Model c (Support d)
7  use(VData x) d = appOp (MOpScore (density d x))
8  use VParam d   = appOp (MOpSample d)
9
10 -- A model with two variables, where first affects the second's distribution
11 simpleModel :: (c Normal, c Uniform) =>
12                Var Double -> Var Double ->
13                Model c (Double, Double)
14 simpleModel mu_v x_v =
15   do mu <- use mu_v (Uniform (-100) 100)
16      x  <- use x_v  (Normal mu 10)
17      return (mu,x)
```

Figure 2. Writing Models in GraPPa

3. The Expressivity vs. Efficiency Continuum

Figure 2 shows how we write models in GraPPa. The random variables of a model are made explicit with the `Var` type (line 2), which are either *data* variables, with an associated value that has been observed, or *parameter* variables, which represent hidden values in a model that have not been observed. To use a variable in a model, the `use` function (lines 5 – 8) is called with the variable and the distribution it is associated with; parameter variables are sampled from the distribution and data variables have their values scored against the distribution. This approach allows users to write a single model and use it in multiple different ways, supplying or omitting values for the random variables of the model. This is useful, for example, in data imputation, where some of the data is missing and must be inferred.

Lines 11 – 17 use this approach to define a simple model, with two variables, `mu_v`, drawn from a uniform distribution between `-100` and `100`, and `x_v`, drawn from a normal distribution around the value of `mu_v`. The type of this model leaves the constraint function, `c`, abstract. The only requirements on `c`, expressed to the left of the Haskell `=>` arrow, are that it is satisfied by the `Normal` and `Uniform` distribution types. This intuitively puts a lower bound on the set of distributions used in the `Model`. This approach is compositional, because the Haskell type-checker knows how to combine constraints written in this way.

Inference methods are then specified with types that choose specific constraint functions for `c`, thereby giving an upper bound on the distributions. For instance, forward sampling methods have input type `Model (SampleableIn m) a`, where `SampleableIn m d` expresses that distribution type `d` can be sampled from in monad `m`. More efficient inference methods, like hill climbing algorithms, might require distributions to be `Continuous`, or isomorphic to the real line.

References

B. Carpenter, A. Gelman, M. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. A. Brubaker, J. Guo, P. Li, and A. Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 2016.

E. Kmett. Free monads for less. `http://comonad.com/reader/2011/free-monads-for-less/`, 2011.

A. Ścibior, Z. Ghahramani, and A. D. Gordon. Practical probabilistic programming with monads. In *Haskell*, 2015.

F. Wood, J. W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *AISTATS*, 2014.