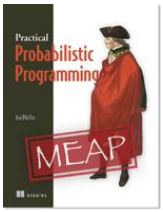
Practical Probabilistic Programming: Open universe situations with unknown number of objects

By Avi Pfeffer



An open universe situation is where you don't know how many objects there are. This article, excerpted from Practical Probabilistic Programming, focuses on number uncertainty, which is addressed by variable size arrays.

Imagine you are implementing a highway traffic surveillance system. Your goal is to monitor the flow of vehicles along the highway and predict when traffic jams might occur. You have video cameras placed at strategic points. To accomplish your goal, you will need to do two things. First, you will need to infer the vehicles passing at each point, including identifying when the same vehicle passes two different cameras. Second, you need to predict future vehicles that will arrive on the highway and the traffic jams that result.

Both of these tasks require reasoning about an unknown number of objects. When you are inferring the vehicles currently on the highway, you don't know how many vehicles there actually are, and your video images don't give you a perfect indication of that number. Sometimes, multiple vehicles might be merged in the image, and some vehicles might not be detected at all. And when you're predicting future traffic jams, you certainly don't know exactly how many vehicles there will be.

A situation where you don't know how many objects there are is known as an **open universe situation** (see below for an explanation of the name). An open universe situation is characterized by two properties:

- You don't know exactly how many objects there are, such as the number of vehicles. This is called **number uncertainty**.
- You are uncertain about the identity of objects. For example, you might be uncertain whether two vehicle images at different sites belong to the same vehicle. This is called **identity uncertainty**.

In this article, I'm going to focus on number uncertainty, which is addressed by variable size arrays.

Open universe modeling

Use of the term "open universe" in probabilistic programming is due to Stuart Russell and his group, who introduced it for their probabilistic programming language BLOG, which is primarily designed to represent and reason about open universe situations. The term originally comes from logic, where "closed world" or "closed universe" means that you assume that only objects that are explicitly mentioned or directly derivable from your model actually exist.

If you are familiar with the logic programming language Prolog, you will know that Prolog practices negation by failure; if it can't prove something to be true, it is assumed to be false. For example, if you are trying to prove that there exists a green vehicle in your image, and you are unable to prove it based on the vehicles you actually know about, then you assume that there is no such vehicle. You don't have to prove that there's no other green vehicle that you don't happen to know about. Negation by failure is a form of closed universe reasoning.

In contrast, full first-order logic is open universe. To prove that there is no green vehicle in your image, you have to prove that there cannot possibly be such a vehicle, even if you don't know about it. Modeling open universe situations is something probabilistic programming languages enable you to do that is hard in other probabilistic reasoning frameworks.

VARIABLE SIZE ARRAYS

Figaro's approach to open universe modeling is to use variable size arrays. A variable size array takes two arguments: an element whose value is the size of the array, and an element generator, which is similar to the element generator for a fixed size array. Figure shows an example of constructing a variable size array. The first argument to the `VariableSizeArray` constructor is an integer element representing the number of elements in the array. The second argument is the element generator, which is a function from an index into the array to an element definition. In Figure 1, each element in the array is defined by a `Beta`, so it is a double element.

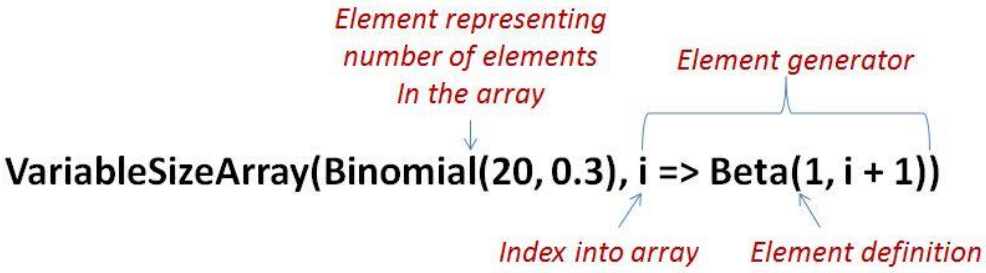


Figure 1: Structure of a variable size array construction

What actually happens when you construct a variable size array? Technically, a variable size array is not a single array at all, but rather represents a random variable whose value could be one of many arrays, each of a different length, depending on the value of the number argument. When you invoke `VariableSizeArray`, you are actually creating a `MakeArray` element to represent this random variable. All the operations on variable size arrays are actually applied to this `MakeArray` element. Under the hood, Figaro makes sure that `MakeArray` is implemented as efficiently as possible to ensure the maximum amount of data sharing between the arrays of different size. In particular, shorter arrays are prefixes of longer arrays and use the same elements. As a programmer, all this is taken care of for you and you don't have to worry about it, but I want to make sure you know what a MakeArray is because you might encounter this type sometimes.

OPERATIONS ON VARIABLE SIZE ARRAYS

What can you do with a variable size array? Pretty much everything you can do with a fixed size array, but there are some important differences. In the following examples, `vsa` will represent a variable size array whose elements are string elements.

- **Getting the element at an index:** In principle, `vsa(5)` will get the element at index 5 (i.e., the sixth element in the array). However, this can be dangerous. You should be sure the array will always have at least six elements to call this method. Otherwise, you are liable to get an `IndexOutOfRangeException`.
- **Safely getting an optional element at an index:** For this reason, Figaro provides a safer way to get at the element at an index. `vsa.get(5)` returns an `Element[Option[String]]`. `Option[String]` is a Scala type that could be `Some[String]`, if there is actually a string, or `None` if there is no string. `vsa.get(5)` represents the following random process:
    - Choose the number of elements according to the number argument.
    - Get the fixed size array of the appropriate size.
    - If the array has a least six elements, let `s` be the value of the element at index 5. Return `Some(s)`.
    - Otherwise, return `None`.

  We see that in either case, the random process produces an `Option[String]`. So the process is represented by an `Element[Option[String]]`. There are many ways to use an `Element[Option[String]]`. You could pass it, for an example, to an `Apply` element that does something interesting if the argument is `Some(s)` and produces a default value if the argument is `None`.

- **Map the variable size array through a function on values:** This is similar to fixed size arrays, except that now Figaro reaches inside the variable size array to the fixed size arrays it points to, and then reaches inside the fixed size arrays to the actual values. For example, `vsa.map(_.length)` produces a new variable size array, in which each string is replaced by its length.
- **Chain the variable size array through a function that maps values to elements:** Again, this is similar to fixed size array, and again Figaro reaches inside the variable size array to the fixed size arrays and chains each of the elements in the fixed size arrays through the function. For example, `vsa.map((s: String) => discrete.Uniform(s:_*))` creates a variable size array in which each element contains a random character from the corresponding string. In this example, `s:_*` turns the string into a sequence of characters, and `discrete.Uniform(s:_*)` selects a random member from the sequence.
- **Folds and aggregates:** All the folds and aggregates that are offered for fixed size arrays are also available for various size arrays. For example, `vsa.count(_.length > 2)` returns an Element[Int] representing the number of strings in the array whose length is greater than two. You can understand this as randomly choosing a fixed size array according to the number argument, and then counting the number of strings in that array whose length is greater than two. Again, consult the Scaladoc to see all the folds and aggregates that are defined.

EXAMPLE: PREDICTING SALES OF AN UNKNOWN NUMBER OF NEW PRODUCTS

In this example, we're going to imagine that we're planning the R&D investment of our company in the coming year. Higher R&D investment leads to more new products being developed, which leads to higher sales. However, at the time of making the investment, we do not know exactly how many new products will be developed for a given level of investment.

Therefore, we use a variable size array to represent the new products. The model is defined as follows:

1. It takes as argument `rNDLevel`, which is a double representing the level of R&D investment.
2. The `numNewProducts` element, which represents the number of new products that are developed as a result of R&D, is an integer element defined by `Geometric(rNDLevel)`. A geometric distribution characterizes a process that goes on for a number of steps. After each step, the process can terminate or it can go on to the next step. The probability of going on to the next step is provided by the parameter of the geometric distribution (in this case, `rNDLevel`). The value of the geometric process is the number of steps before termination. The probabilities of the number of steps decrease geometrically by a factor of `rNDLevel` each time. The higher the `rNDLevel`, the longer the process is likely to go on, and the more new products will be developed.
3. We create a variable size array called `productQuality`, representing the quality of each of the new products. The number argument of this variable size array is `numNewProducts`. The element generator is the function that maps an index `i` to `Beta(1, i + 1)`. The expected value of `Beta(1, i + 1)` is `1 / (i + 2)`, so the product quality tends to decrease as more new products are developed, representing a diminishing return on R&D investment.
4. Next, we turn the product quality into a prediction of the sales of each product. This takes place in two steps. First, we generate the raw sales using a Normal distribution centered on the product quality. However, a Normal distribution can have negative values, and negative sales are impossible, so in the second step, we truncate the Normal distribution and give it a lower bound of zero. These two steps are accomplished using the chain and map methods of `VariableSizeArray`.
5. Finally, we get the total sales by folding the sum function through the `productSales` variable size array.

The full code for the model is shown below.

```
val numNewProducts = Geometric(rNDLevel)
val productQuality =
  VariableSizeArray(numNewProducts, i => Beta(1, i + 1))
val productSalesRaw = productQuality.chain(Normal(_, 0.5))
val productSales = productSalesRaw.map(_.max(0))
val totalSales = productSales.foldLeft(0.0)(_ + _)
```

You've just seen an example where probabilistic programming lets you implement quite a rich process in just a few lines of code.