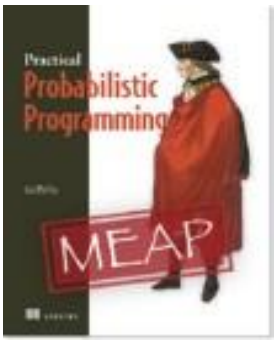
Practical Probabilistic Programming: Your First Model

By Avi Pfeffer



In this article, excerpted from Practical Probabilistic Programming by Avi Pfeffer, we'll build the simplest Figaro model possible.

In this article, we're going to start by building the simplest possible Figaro model. This will be a model consisting of just a single atomic element. Before we can build a model, however, we have to import the necessary Figaro constructs.

```
import com.cra.figaro.language._
```

This imports all the classes in the `com.cra.figaro.language` package, which contains the most basic Figaro constructs. One of the classes is called `Flip`. We can build a simple model using `Flip`.

```
val sunnyToday = Flip(0.2)
```

Figure 1 explains what this line of code means. It is important to be clear about what is Scala and what is Figaro. In this line of code, we have created a Scala variable named `sunnyToday` and assigned it the value `Flip(0.2)`. The Scala value `Flip(0.2)` is a Figaro element that represents the random process that results in a value of `true` with probability 0.2 and `false` with probability 0.8. An element is a data structure representing a process that randomly produces a value. A random process can result in a number of outcomes. Each possible outcome is known as a *value* of the process. So `Flip(0.2)` is an element whose possible values are the Booleans `true` and `false`. So, to summarize, we have a Scala variable with a Scala value. That Scala value is a Figaro element and it has a number of possible values representing different outcomes of the process.

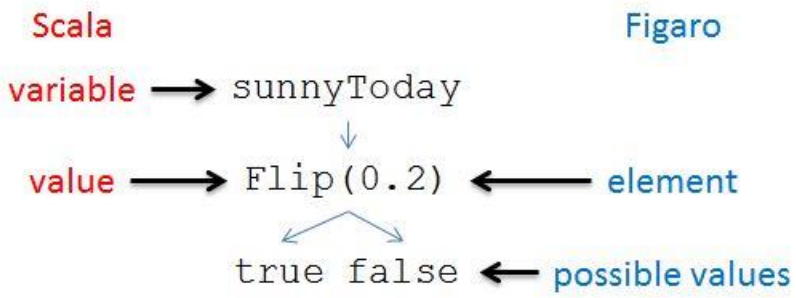


Figure **Error! No text of specified style in document.**1: The relationship between Scala variables and values and Figaro elements and possible values

In Scala, a type can be parameterized by another type that describes its contents. You might be familiar with this concept from Java generics, where you can have, for example, a list of integers or a list of strings. All Figaro elements are instances of the `Element` class. The Element class is parameterized by the type of values an element can take.

This type is known as the *value type* of the element. Since `Flip(0.2)` can take Boolean values, the value type of `Flip(0.2)` is `Boolean`. The notation for this is that `Flip(0.2)` is an instance of `Element[Boolean]`.

KEY DEFINITIONS

Element: A Figaro data structure representing a random process.

Value: *A possible outcome of the random process.*

Value type: *The Scala type representing the possible values of the element.*

There's been a lot to say about a very simple model. Fortunately, what you have just learned applies to all Figaro models. Figaro models are created by taking simple Figaro elements—the building blocks—and combining them to create more complex elements and sets of related elements. So, the definitions you have just learned of elements, values, and value types, are the most important definitions in Figaro.

Before going on to build more complex models, let's have a look at how you can reason with this model.

Running Inference and Answering a Query

We've built a simple model. Let's run inference so we can query the probability that sunnyToday is true. First we need to import the inference algorithm we will use:

```
import com.cra.figaro.algorithm.factored.VariableElimination
```

This imports the algorithm known as variable elimination, which is an exact inference algorithm, meaning it computes exactly the probabilities implied by your model and the evidence. Probabilistic inference is complex, so exact algorithms sometimes take too long or run out of memory, so Figaro provides a number of approximate algorithms that usually compute answers in the ballpark of the exact answers. In this chapter, we'll use simple models, so variable elimination will for the most part work.

Now, Figaro provides a single command to specify a query, run an algorithm, and get an answer. We can write the following code:

```
println(VariableElimination.probability(sunnyToday, true))
```

This command prints 0.2. Our model consists only of the element `Flip(0.2)`, which comes out true with probability 0.2. Variable elimination correctly computes that the probability `sunnyToday` is true is 0.2.

To elaborate, the command we have just seen accomplishes several things. It first creates an instance of the variable elimination algorithm. It tells this instance that the query target is `sunnyToday`. Then it runs the algorithm and returns the probability that the value of `sunnyToday` is true. The command also takes care of cleaning up after itself, releasing any resources used by the algorithm.

Building Up Models and Making Observations

Now let's start building up a more interesting model. We're going to need a Figaro construct called `If`, so let's import it. We also need a construct named `Select`, but that was already imported along with the rest of `com.cra.figaro.language`.

```
import com.cra.figaro.library.compound.If
```

Let's use `If` and `Select` to build a more complex element.

```
val greetingToday = If(sunnyToday,
     Select(0.6 -> "Hello world!", 0.4 -> "Howdy, universe!"),
     Select(0.2 -> "Hello world!", 0.8 -> "Oh no, not again"))
```

The way to think about this is that an element represents a random process. In this case, the element named `greetingToday` represents the process that first checks the value of `sunnyToday`. If this value is `true`, it selects "Hello world!" with probability 0.6 and "Howdy, universe!" with probability 0.4. If the value of `sunnyToday` is `false`, it selects "Hello world!" with probability 0.2 and "Oh no, not again" with probability 0.8. `greetingToday` is a compound element, because it is built out of three elements. Since the possible values of `greetingToday` are strings, `greetingToday` is an `Element[String]`.

Now, let's say that you've seen the greeting today and it was "Hello world!". You can specify this evidence using an observation as follows:

```
greetingToday.observe("Hello world!")
```

Now, we can find out the probability that today is sunny, given that the greeting is "Hello world!".

```
println(VariableElimination.probability(sunnyToday, true))
```

This prints out 0.4285714285714285. Note that this is significantly higher than the previous answer (0.2). This is because "Hello world!" is more likely if today is sunny than otherwise, so the evidence provides support for today being sunny. This inference is a simple example of Bayes' rule in action.

We're going to be extending this model and running more queries with different evidence, so we'll want to remove the observation on the variable `greetingToday`. We can do this with the following command:

```
greetingToday.unobserve()
```

Now, if we ask our query:

```
println(VariableElimination.probability(sunnyToday, true))
```

we get the same answer, 0.2, as before we specified the evidence.

To finish off this section, let's elaborate the model further:

```
val sunnyTomorrow = If(sunnyToday, Flip(0.8), Flip(0.05))
val greetingTomorrow = If(sunnyTomorrow,
     Select(0.6 -> "Hello world!", 0.4 -> "Howdy, universe!"),
     Select(0.2 -> "Hello world!", 0.8 -> "Oh no, not again"))
```

We'll compute the probability tomorrow's greeting is "Hello world!", both without and with evidence about today's greeting.

```
println(VariableElimination.probability(greetingTomorrow, "Hello world!"))
// prints 0.27999999999999997

greetingToday.observe("Hello world!")
println(VariableElimination.probability(greetingTomorrow, "Hello world!"))
// prints 0.3485714285714286
```

We see that after observing today's greeting is "Hello world!", the probability tomorrow's greeting is "Hello world!" goes up. Why? Because today's greeting being "Hello world!" makes it more likely that today is sunny, which in turn makes it more likely that tomorrow is sunny, which finally makes it more likely that tomorrow's greeting will be "Hello world!". This is an example of inferring the past to better predict the future, and Figaro takes care of all these calculations.

Understanding How the Model is Built

Now that we've seen all the steps in creating a model, specifying evidence and queries, running evidence and getting answers, let's take a closer look at the Hello World model to understand how it is built up from the building blocks (atomic elements) and connectors (compound elements).

Figure 2 is a graphical depiction of this model. The figure first reproduces the model definition, with each Scala variable color coded. The second half of the figure shows a graph in which each node in this graph represents an element in the model. The colored blocks show which elements are associated with which Scala variable. Some elements are values of Scala variables themselves. For example, the value of the `sunnyToday` Scala variable is the `Flip(0.2)` element. If an element is the value of a Scala variable, the graph shows both the name of the variable and the element. The model also contains some elements that are not the values of specific Scala variables, but still appear in the model. For example, since the definition of `sunnyTomorrow` is `If(sunnyToday, Flip(0.8), Flip(0.05))`, the elements `Flip(0.8)` and `Flip(0.05)` are also part of the model, so they are shown as nodes in the graph.

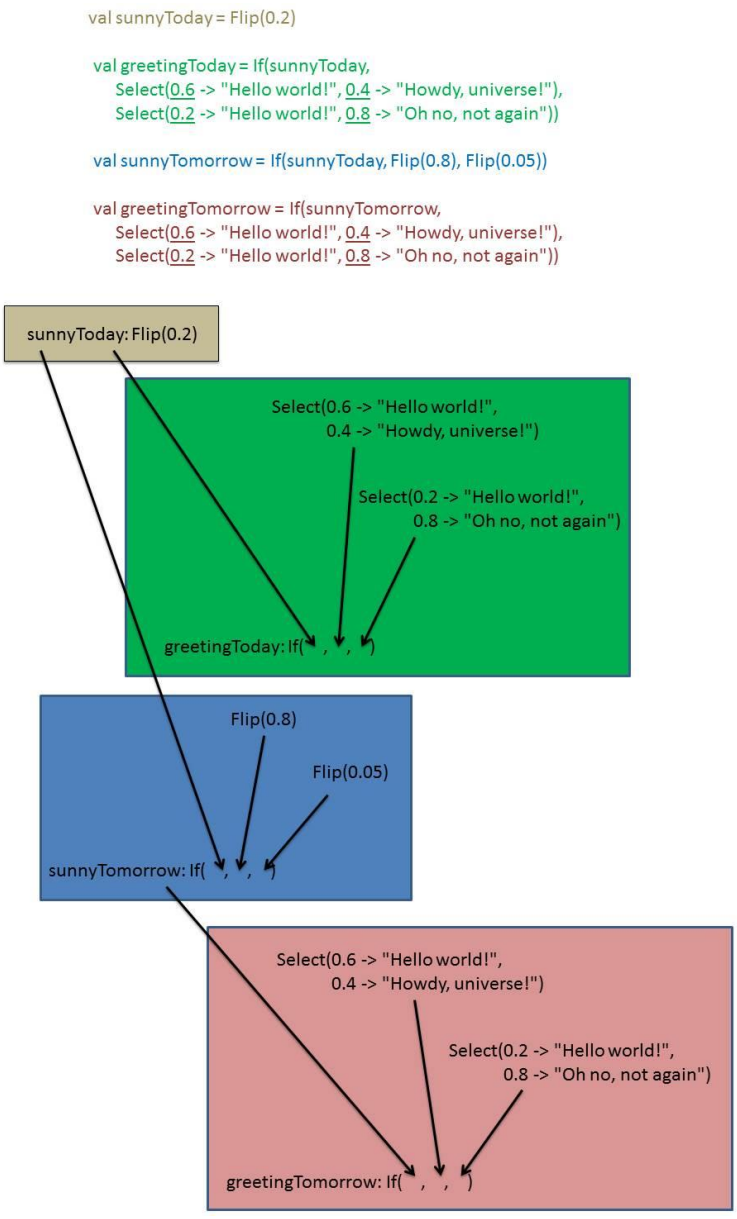


Figure 2: Structure of the Hello World model as a graph. Each node in the graph is an element. Edges in the graph show when one element is used by another element.

The graph contains edges between elements. For example, there's an edge from `Flip(0.8)` to the `If` element that is the value of `sunnyTomorrow`. This means that the element `Flip(0.8)` is used by the `If`. In general, there is an edge from one element to another if the second element uses the first element in its definition. Since only compound elements are built out of other elements, only compound elements can be the destination of an edge.

Repeated Elements: When Are They the Same and When Are They Different?

One point to notice is that `Select(0.6 -> "Hello world!", 0.4 -> "Howdy, universe!")` appears twice in the graph, and the same for `Select(0.2 -> "Hello world!", 0.8 -> "Oh no, not again")`. This is because the definition itself appears twice in the code, once for `greetingToday` and once for `greetingTomorrow`. These are two distinct elements even though their definitions are the same. What this means is that they can take different values in the same execution of the random process defined by this Figaro model. For example, the first instance of the element could have the value "`Hello world!`" while the second could have the value "`Howdy, universe!`". This makes sense, because one is being used to define `greetingToday` and the other is used to define `greetingTomorrow`. It's quite possible that today's greeting and tomorrow's greeting will be different.

On the other hand, notice that the Scala variable `sunnyToday` also appears twice in the code, once in the definition of `greetingToday` and once for `sunnyTomorrow`. But the element that is the value of `sunnyToday` only appears once in the graph. Why? Because `sunnyToday` is a Scala variable, not the definition of a Figaro element. When a Scala variable appears more than once in a piece of code, it is the same variable, and so the same value is used. In our model, this makes sense; it's the same day's weather that's being used in the definitions of `greetingToday` and `sunnyTomorrow`, so it will have the same value in any random execution of the model.

If this seems mysterious, this distinction is not really about Figaro. Consider the following ordinary code:

```
def f() = System.currentTimeMillis() //#A
val x = f()
Thread.sleep(1000) //#B
val y = f()
println(x == y) //#C
val z = y
val w = y
println(z == w) //#D
```

#A Gets the current time in milliseconds
#B Waits one second before continuing
#C Prints false
#D Prints true

Why does the first `println` print false while the second one prints true? Because `x` and `y` are the result of two different applications of the function `f`. Even though they both have the same definition, the two applications of `f` are one second apart, so they produce different results. So `x == y` is false. On the other hand, both `z` and `w` are defined to be the value of the Scala variable y. This variable has the same value in both places, so `z == w` is true.

Ok, so now you've got it, you have a general idea of all the main concepts of Figaro and how they fit together.