A Brief Introduction to Figaro

April 16, 2013

(Language, features, and the tutorial, from which most of this slide deck is appreciatively "borrowed," by Avi Pfeffer. Slides and errors by Nathaniel Wesley Filardo.)

What is Figaro?

- Figaro is a probabilistic programming language.
  - Describes probabilistic models and inference thereon.
- Designed by Avi Pfeffer at Charles River Analytics.
  - Successor to his earlier language, IBAL.
- Open-source, available under a 4-BSD-like license, from `https://www.cra.com`
- Embedded Domain-Specific Language in Scala.

What is a domain-specific language?

A **domain-specific language** is a *syntax* which is geared towards the *semantics* of a problem domain.

Examples abound:

Problem Domain	DSL
Lexing	{POSIX,Perl,...} regular expressions
Parsing	YACC, ANTLR, ...
Database interaction	SQL, SPARQL, ...
Graph rendering	GraphViz, ...
Mathematical programming	AMPL, ...
Symbolic algebra	Mathematica, Maple, Maxima, ...

Embedded Domain-Specific Language?

Embedded (sometimes *Internal*) DSLs are a (relatively) recently popularized form of software development.

- Defined in a **host language**
  - Popular examples: Lisp, Forth, Prolog, Scala, Haskell.
  - Typically very expressive and syntactically flexible.
  - EDSL *inherits* features (e.g., calling convention, type system) and tools (e.g., compiler) from its host.
- Offer a library of domain-relevant functionality
  - and powerful operators.
- Often continue to try to hide or de-emphasize details of host.

What does a DSL for Probabilistic Modeling look like?

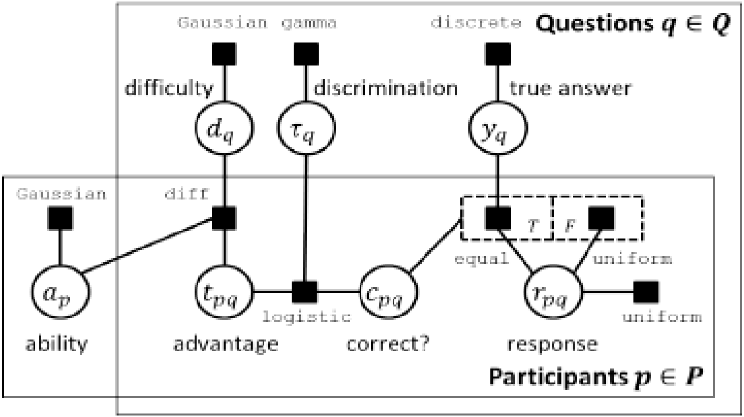
We might consider the canonical mathematical syntax for describing generative models a DSL. Its basic operation is to declare that a value is distributed according to some distribution, parameterized by other values:

$$\begin{aligned} a_p &\sim \mathrm{Normal}(\mu_p, \sigma_p^2) \\ d_q &\sim \mathrm{Normal}(\mu_q, \sigma_q^2) \\ t_{pq} &= a_p - d_q \end{aligned}$$

(These are parts of the model of Bachrach et al., *How To Grade a Test Without Knowing the Answers – A Bayesian Graphical Model for Adaptive Crowdsourcing and Aptitude Testing*, ICML 2012.)

What does a DSL for Probabilistic Modeling look like?

We could argue that plate notation for graphical models is a DSL. Bachrach et al's full model is shown in their paper as:



What's wrong with this picture?

Unfortunately, these forms of PRMs are

- Not directly executable
- Even when we make them executable (which takes time), all we get is the sampler!

What's wrong with this picture?

Unfortunately, these forms of PRMs are

- Not directly executable
- Even when we make them executable (which takes time), all we get is the sampler!

That is, it takes even more work to be able to...

- specify observed or constrained variables.
- run *inference* to recover information about un-observed variables.

So what does Figaro bring to the table?

Figaro, as with other PPLs, lets the programmer

- capture a PRM in a syntax that is not so far removed from the mathematical notation (IMHO),
- specify knowns about the variables in the model,
- and run inference.

Because it is embedded in Scala, the full power of that language can be brought to bear on any of these parts of the task.

So what does Figaro bring to the table?

Figaro, as with other PPLs, lets the programmer

- capture a PRM in a syntax that is not so far removed from the mathematical notation (IMHO),
- specify knowns about the variables in the model,
- and run inference.

Because it is embedded in Scala, the full power of that language can be brought to bear on any of these parts of the task. Anecdotally, I was able to take the model from before (Bachrach et al.) and cobble together the program in 2.5 hours and 65 lines.

- I am not a domain expert.
- I don't really even know Scala all that well.
- The resulting program is unlikely to win any speed contests.
- And it's a little too big to fit in slides.

Defining A Model

Figaro represents models as collections of **elements**.

- AKA **random variable**.
- Elements are parameterized by their output type: an `Element[T]` is a `T`-valued random variable.
- Elements come in two flavors: **atomic** and **compound**.
  - Atomic elements are self-contained.
  - Compound elements wire other elements together.

Defining A Model: Atomic Elements

- Constants (the least random “random variable”).

```
Constant(4)
```

- Discrete distributions:

```
Flip(0.75)
```

```
Select(1 -> "a", 1 -> "b", 2 -> "c")
```

- Continuous distributions:

```
Uniform(0.0,1.0)
```

```
Normal(0.0,4.0)
```

- Many more than listed on this slide
  - Also quite easy to add more

Defining A Model: Compound Elements

Three base compounds:

1. A function applied to a RV is itself a RV:

$$\underbrace{\texttt{Apply(}\underbrace{\texttt{Uniform(0.0,1.0)}}_{\texttt{Element[Double]}}\texttt{, }\underbrace{\texttt{(d:Double) => d > 0.3}}_{\texttt{(Double) => Bool}}\texttt{)}}_{\texttt{Element[Bool]}}$$

Defining A Model: Compound Elements

Three base compounds:

1. A function applied to a RV is itself a RV:

```
Apply(Uniform(0.0,1.0), (d:Double) => d > 0.3)
```

2. Several `T`-valued RVs can be grouped to a list-of-`T`-valued RV:

$$\underbrace{\texttt{Inject(}\underbrace{\texttt{Uniform(0.0,1.0)}}_{\texttt{Element[Double]}}\texttt{, }\underbrace{\texttt{Uniform(1.0,2.0)}}_{\texttt{Element[Double]}}\texttt{)}}_{\texttt{Element[Seq[Double]]}}$$

Defining A Model: Compound Elements

Three base compounds:

1. A function applied to a RV is itself a RV:

   ```
   Apply(Uniform(0.0,1.0), (d:Double) => d > 0.3)
   ```

2. Several `T`-valued RVs can be grouped to a list-of-`T`-valued RV:

   ```
   Inject(Uniform(0.0,1.0), Uniform(1.0,2.0))
   ```

   These two combine to let us define a sum of random variables:

   ```
   val es = Inject(Uniform(0.0,1.0), Normal(0.0,1.0))
   val esum = Apply(es, (x:Seq[Double]) => (0 /: x) (_ + _))
   ```

Defining A Model: Compound Elements

Three base compounds:

1. A function applied to a RV is itself a RV:

   ```
   Apply(Uniform(0.0,1.0), (d:Double) => d > 0.3)
   ```

2. Several `T`-valued RVs can be grouped to a list-of-`T`-valued RV:

   ```
   Inject(Uniform(0.0,1.0), Uniform(1.0,2.0))
   ```

3. Random variables can be chained:

   $$\underbrace{\texttt{Flip(}\underbrace{\texttt{Uniform(0.0,1.0)}}_{\texttt{Element[Double]}}\texttt{)}}_{\texttt{Element[Bool]}}$$

Defining A Model: Compound Elements

Three base compounds:

1. A function applied to a RV is itself a RV:

```
Apply(Uniform(0.0,1.0), (d:Double) => d > 0.3)
```

2. Several `T`-valued RVs can be grouped to a list-of-`T`-valued RV:

```
Inject(Uniform(0.0,1.0), Uniform(1.0,2.0))
```

3. Random variables can be chained; under the covers:

$$\underbrace{\texttt{Chain(}\underbrace{\texttt{Uniform(0.0,1.0)}}_{\texttt{Element[Double]}}\texttt{, }\underbrace{\texttt{(d:Double) => Flip(d)}}_{\texttt{(Double) => Element[Bool]}}\texttt{)}}_{\texttt{Element[Bool]}}$$

Defining A Model: Compound Elements

Three base compounds:[1]

1. A function applied to a RV is itself a RV:

```
Apply(Uniform(0.0,1.0), (d:Double) => d > 0.3)
```

2. Several `T`-valued RVs can be grouped to a list-of-`T`-valued RV:

```
Inject(Uniform(0.0,1.0), Uniform(1.0,2.0))
```

3. Random variables can be chained; under the covers:

$$\underbrace{\texttt{Chain(}\underbrace{\texttt{Uniform(0.0,1.0)}}_{\texttt{Element[Double]}}\texttt{, }\underbrace{\texttt{(d:Double) => Flip(d)}}_{\texttt{(Double) => Element[Bool]}}\texttt{)}}_{\texttt{Element[Bool]}}$$

[1] The categorically inclined will recognize these as the `Element` functor on morphisms, a folded strength, and a monadic bind. The not-so-inclined should take all that to mean that these are the right things to have.

Many more things in the library built up from these:

- Conditionals:

$$\underbrace{\texttt{If(}\underbrace{\texttt{Flip(0.3)}}_{\texttt{Element[Bool]}}\texttt{, }\underbrace{\texttt{Constant("Yes")}}_{\texttt{Element[String]}}\texttt{, }\underbrace{\texttt{Constant("No")}}_{\texttt{Element[String]}}\texttt{)}}_{\texttt{Element[String]}}$$

Defining A Model: Compound Elements

Many more things in the library built up from these:

- Conditionals:

```
If(Flip(0.3), Constant("Yes"), Constant("No"))
```

- Tuples:

$$\underbrace{\texttt{\^{}\^{}(}\underbrace{\texttt{Flip(0.4)}}_{\texttt{Element[Bool]}}\texttt{, }\underbrace{\texttt{Normal(0.0,1.0)}}_{\texttt{Element[Double]}}\texttt{)}}_{\texttt{Element[(Bool,Double)]}}$$

Defining A Model: Compound Elements

Many more things in the library built up from these:

- Conditionals:

```
If(Flip(0.3), Constant("Yes"), Constant("No"))
```

- Tuples:

```
^^(Flip(0.4), Normal(0.0,1.0))
```

- Convenient sugars for specifying conditional probability tables.

Defining A Model: Elements are Objects

When we do inference over a model (later), we assign values to elements. *Each element in a Figaro model will have the same value throughout.*

Contrast:

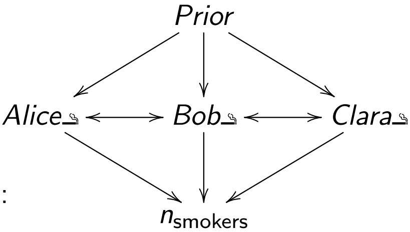
1. Two separate flips:

```
val x = Flip(0.75)
val y = Flip(0.75)
If(Eq(x,y),Constant("Eq"),Constant("Neq"))
```

2. A single flip:

```
val x = Flip(0.75)
If(Eq(x,x),Constant("Eq"),Constant("Neq"))
```

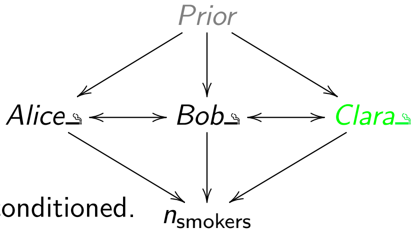
So let's define a model



A small smokers-and-friends example:

```
class Person { val smokes = Flip(smokePrior); }

val smokePrior = Uniform(0.0,1.0);

val alice, bob, clara = new Person
val allPeople = List(alice,bob,clara)

val nSmokers = Apply(Inject(allPeople map (_.smokes):_*),
                     (_:Seq[Boolean]).count((b:Boolean) => b))

val friends = List((alice,bob),(bob,clara))
```

Specifying Hard Evidence



Elements within the model may be conditioned.

- The simplest kind of conditioning is a **observation**, which fixes the value of an element:

```
clara.smokes.observe(true)
```

- More general conditioning is possible, too:

```
smokePrior.condition = (d:Double) => (d < 0.75)
```

- Also possible to add conditions dynamically:

```
smokePrior.addCondition((d:Double) => d > 0.2)
```

Specifying Soft Evidence: Constraints

It is common in PRMs to attach unary factors to RVs. These capture *indirect observation* assertions of the form

> *All other things being equal, having observed $f(v)$ it is k times more likely that v is such that $g(v)$ holds than not.*

For example,

> *Absent any other evidence about nwf's political views, he has been seen walking barefoot in the woods, which doubles the odds that he is a hippie.*

Specifying Soft Evidence: Constraints

It is common in PRMs to attach unary factors to RVs. These capture *indirect observation* assertions of the form

> *All other things being equal, having observed $f(v)$ it is k times more likely that v is such that $g(v)$ holds than not.*

For example,

> *Absent any other evidence about nwf's political views, he has been seen walking barefoot in the woods, which doubles the odds that he is a hippie.*

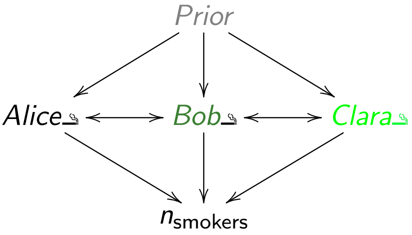
Figaro has a convenient short-cut for this, called **constraints**:

```
bob.smokes.constraint = (b:Boolean) => if (b) 2.0; else 1.0
```

And of course we can add dynamically,

```
bob.smokes.addConstraint((b:Boolean) => if (b) 4.0; else 3.0
```

Specifying Soft Evidence: Constraints



In fact, we often will create elements explicitly for the purpose of constraining them. We might believe that friends are three times more likely to share smoking habits than not:

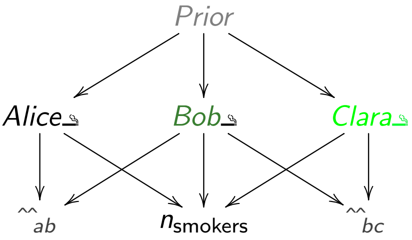
- Need a function that expresses our three-to-one odds:

```
def smokingInfluence(i:Boolean,j:Boolean) =
  if (i == j) 3.0; else 1.0
```

- And need to traverse the list of friends, creating and constraining pair elements.

```
for { (p1, p2) <- friends } {
  ^^(p1.smokes, p2.smokes).constraint =
      Function.tupled(smokingInfluence _)
}
```

Specifying Soft Evidence: Constraints



In fact, we often will create elements explicitly for the purpose of constraining them. We might believe that friends are three times more likely to share smoking habits than not:

- Need a function that expresses our three-to-one odds:

```
def smokingInfluence(i:Boolean,j:Boolean) =
  if (i == j) 3.0; else 1.0
```

- And need to traverse the list of friends, creating and constraining pair elements.

```
for { (p1, p2) <- friends } {
  ^^(p1.smokes, p2.smokes).constraint =
       Function.tupled(smokingInfluence _)
}
```

Reasoning

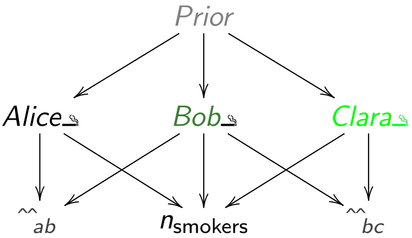
Figaro can, out of the box, try to do several kinds of probabilistic inference:

- Range (i.e., support) computation
- Exact inference w/ variable elimination
- Probability of evidence
- Most Probable Explanation
- Importance/rejection sampling (one-time or any-time)
- MCMC (one-time or any-time)
  - With built-in or user-specified proposers
- Particle Filtering (on dynamic models)

And, as with Elements, it aims to make it easy to add your own.

Reasoning: Importance Sampling

Importance (+rejection) sampling is typical of Figaro's algorithms:



- Create the inference engine, passing the elements we want to know about and any algorithm parameters:

```
val ia = Importance(alice.smokes,nSmokers,1000)
ia.start()
```

- Ask questions:

```
val pm  = ia.probability(nSmokers, (_:Int) >= 2) //0.9343...
val eas = ia.expectation(alice.smokes,            //3.7056...
           (b:Boolean) => if (b) 5.0; else 0.0)
val das = ia.distribution(alice.smokes)
    // stream of (0.2503...,false), (0.7496...,true)
```

In Summary

Figaro aims to be a Scala EDSL for the whole pipeline of probabilistic modeling: describing the model, stating observations and priors, and performing inference.

Questions?

Reasoning: Metropolis-Hastings MCMC

There are two (more) things needed for MHMCMC:

- A **proposal scheme**.
- The ability to compute the ratio of the model's constraints before and after a change.

Reasoning: Metropolis-Hastings MCMC

There are two (more) things needed for MHMCMC:

- A **proposal scheme**.
- The ability to compute the ratio of the model's constraints before and after a change.
  - All built-in `Elements` know how to do this.
  - Extensions need to be taught; not hard.

Reasoning: Metropolis-Hastings MCMC

There are two (more) things needed for MHMCMC:

- A **proposal scheme**.
- The ability to compute the ratio of the model's constraints before and after a change.
  - All built-in `Elements` know how to do this.
  - Extensions need to be taught; not hard.

There's even a default proposal scheme:

- Selects an element from the model at random each step.
- This may or may not mix well, depending on your model. How do we do better?

Reasoning: Building a Proposal Scheme

Proposal schemes are themselves built up from modular pieces!

- The base case is a `ProposalScheme`, which takes a list of elements and selects each in turn:

  ```
  val psA = ProposalScheme(alice.smokes)
  val psB = ProposalScheme(bob.smokes)
  ```

- A `DisjointScheme` takes a weighted list of `ProposalSchemes` and selects among them at random:

  ```
  val psAorB = DisjointScheme(0.75 -> psA, 0.25 -> psB)
  ```

- An `UntypedScheme` proposes an element this step and then (optionally) behaves like another scheme in subsequent steps.

  ```
  val psAthenB = UntypedScheme(alice.smokes, Some(psB))
  ```

- The `TypedScheme` proposes an element and can continue as a different scheme in light of that element's value.

Under the hood: Elements

An `Element[T]` is really a *deterministic* producer of `T` values, with a relatively simple API:

- An abstract type `Randomness`.
- A non-deterministic randomness-creation function `generateRandomness : () => Randomness`.
- A *deterministic* `generateValue : Randomness => T`.
- A density function `density : T => Double`.

For MCMC, there is an additional hook:

- `nextRandomness : Randomness => (Randomness, Double)` which not only proposes a new `Randomness` for the object but also returns the proposal probability ratio $\frac{P(r' \to r)P(r')}{P(r \to r')P(r)}$.
  - Useful so that elements can help the chain mix quickly.
  - e.g., `SwitchingFlip`, a version of `Flip` that always changes its mind.