Figaro Tutorial

Avi Pfeffer, Charles River Analytics

Contents

[Introduction 1](#_Toc317243568)

[What is Figaro? 1](#_Toc317243569)

[This tutorial 1](#_Toc317243570)

[Hello world! 1](#_Toc317243571)

[Figaro’s representation 1](#_Toc317243572)

[Elements 1](#_Toc317243573)

[Atomic elements 1](#_Toc317243574)

[Compound elements 1](#_Toc317243575)

[Chain 1](#_Toc317243576)

[Apply and Inject 1](#_Toc317243577)

[Creating models 1](#_Toc317243578)

[Basic models 1](#_Toc317243579)

[Conditions and constraints 1](#_Toc317243580)

[Classes, instances, and relationships 1](#_Toc317243581)

[Mutable fields 1](#_Toc317243582)

[Universes 1](#_Toc317243583)

[Names, element collections, and references 1](#_Toc317243584)

[Multi-valued References and Aggregates 1](#_Toc317243585)

[Open Universe Models 1](#_Toc317243586)

[Reasoning 1](#_Toc317243587)

[Computing ranges 1](#_Toc317243588)

[Exact inference using variable elimination 1](#_Toc317243589)

[Importance sampling 1](#_Toc317243590)

[Markov chain Monte Carlo 1](#_Toc317243591)

[Probability of evidence algorithm 1](#_Toc317243592)

[Computing the most likely values of elements 1](#_Toc317243593)

[Reasoning with dependent universes 1](#_Toc317243594)

[Abstractions 1](#_Toc317243595)

[Dynamic models and filtering 1](#_Toc317243596)

[Creating a new element class 1](#_Toc317243597)

[Creating an atomic class with inheritance 1](#_Toc317243598)

[Creating a compound class with inheritance 1](#_Toc317243599)

[Creating an atomic class without inheritance 1](#_Toc317243600)

[Creating a compound class without inheritance 1](#_Toc317243601)

[Making a class usable by variable elimination 1](#_Toc317243602)

[Creating a class with special Metropolis-Hastings behavior 1](#_Toc317243603)

[Creating a new algorithm 1](#_Toc317243604)

[One-time query answering algorithm 1](#_Toc317243605)

[Anytime algorithms 1](#_Toc317243606)

[Probability of evidence algorithm 1](#_Toc317243607)

[Allowing extension to new element classes 1](#_Toc317243608)

[Creating a new category of algorithm 1](#_Toc317243609)

[Conclusion 1](#_Toc317243610)

# Introduction

## What is Figaro?

Developing a new probabilistic model requires developing a representation for the model and a reasoning algorithm that can draw useful conclusions from evidence. These can both be challenging tasks. Furthermore, it can be hard to integrate a probabilistic model in a larger program.

Figaro is a probabilistic programming language that helps address both these issues. Figaro makes it possible to express probabilistic models using the power of programming languages, giving the modeler the expressive tools to create all sorts of models. Figaro comes with a number of built-in reasoning algorithms that can be applied automatically to new models. In addition, Figaro models are data structures in the Scala programming language, which is interoperable with Java, and can be constructed, manipulated, and used directly within any Scala or Java program.

Figaro is extremely expressive. It can represent a wide variety of models, including:

* directed and undirected models
* models in which conditions and constraints are expressed by arbitrary Scala functions
* models involving inter-related objects
* open universe models in which we don’t know what or how many objects exist
* models involving discrete and continuous elements
* models in which the elements are rich data structures such as trees

Figaro provides a rich library of constructs to build these models, and provides ways to extend this library to create your own model elements.

Figaro’s library of reasoning algorithms is also extensible. Current built-in algorithms include:

* Exact inference using variable elimination
* Importance sampling
* Metropolis-Hastings, with an expressive language to define proposal distributions
* Support computation
* Most probable explanation (MPE) using variable elimination

Figaro provides both regular and anytime versions of some of these algorithms. In addition to the built-in algorithms, Figaro provides a number of tools for creating your own reasoning algorithms.

Figaro can also represent dynamic models and provides a particle filtering algorithm for such models.

## This tutorial

This tutorial is a guide to using Figaro. Figaro is a probabilistic programming language, meaning that it can be used to create probabilistic models by writing programs in a programming language. In Figaro’s case, the underlying programming language is Scala. Scala combines object-oriented and functional programming styles and is interoperable with Java, so a Figaro program can be used within a Java program directly.

To be precise, Figaro is a Scala library. It defines rich data structures for probabilistic models and reasoning algorithms for reasoning with those models. Because these are Scala data structures, Figaro models can be created using the full power of Scala. These three things are the key to Figaro: the ability to represent an extremely large and interesting class of probabilistic models using these data structures; the ability to use a reasoning algorithm on these data structures to draw conclusions about the probabilistic model; and the ability to create and manipulate the data structures using Scala. Figaro is also extensible. It is easy to create new kinds of data structures in the library, and, while developing new algorithms is a more complex task, Figaro also provides the means to develop new algorithms for the library.

This tutorial assumes some basic knowledge of probabilistic modeling and inference to derive the maximum benefit from it. Also, while this tutorial is not an introduction to Scala, it will explain some Scala constructs as it goes along, so that the reader can make basic use of Figaro after reading the tutorial. However, to get the full benefit of Figaro, it is recommended that the reader learn some Scala. This could prove well worth the reader’s while, because Scala is a language that combines elegance and practicality in a useful way.

After presenting a “Hello world!” example, the tutorial will begin with a discussion of Figaro’s representation, i.e. the data structures that underlie the probabilistic models. Next, it will give examples using Scala of creating Figaro models. It will then describe how to use the built-in reasoning algorithms, followed by a brief discussion of probabilistic programming for dynamic models. The last two sections of the tutorial are geared towards users who want to extend Figaro, first describing how to create new modeling data structures and then describing how to create new algorithms.

# Hello world!

Make sure Scala version 2.8.0 or later is installed on your machine. Extract the file figaro.jar to some location. Change directory to that location and enter at the command prompt

scala –classpath “figaro.jar;$CLASSPATH”

This starts the Scala interactive console and makes sure all the Figaro classes are available. The interactive console reads one line of Scala code at a time and interprets it. It is useful for learning and trying new things. Ordinarily, you would use the compiler to compile a program into Java byte code and run it. To use the Scala compiler, use the scalac or fsc command, again making sure Figaro.jar is in the class path.

Once in the interactive console, at the Scala prompt, enter

import com.cra.figaro.\_

This loads the main Figaro directory. Then enter

import language.\_

This loads the subdirectory that allows you to create models using the core language. Now we’ll create a probabilistic model and give it a name:

val hw = Constant(“Hello world!”)

This line creates a field hw whose value is the probabilistic model that produces the string “Hello world!” with probability 1. To exercise the model, we need to create an instance of an algorithm. We’ll use an importance sampling algorithm. First we need to import the algorithm’s definition:

import algorithm.sampling.\_

Now we create the algorithm, telling it that the target model is hw:

val alg = Importance(1000, hw)

The 1000 tells the sampler to take 1000 samples. Before we can query the algorithm for an answer, we have to tell it to start running:

alg.start()

We can now ask for the probability of various strings. Enter

alg.probability(hw, “Hello world!”)

Scala responds with something like

res3: Double = 1.0

This means that the answer is of type Double, has value 1.0, and is given the name res3. We can similarly ask

alg.probability(hw, “Goodbye!”)

Scala responds with something like

res4: Double = 0.0

# Figaro’s representation

This section describes the basic building blocks of Figaro models. We present the basic definitions of different kinds of model components. In the following section, we will show how to use these components to create a rich variety of models.

## Elements

All Figaro data structures are *elements*. Elements can be combined in various ways to produce more complex elements. The simplest elements are *atomic* elements that do not depend on other elements. An example of an atomic element is

Constant(6)

This defines the probabilistic model that produces the integer 6 with probability 1. Another atomic element is

Constant(“Hello”)

which produces the string “Hello” with probability 1. These two examples illustrate that every Figaro element has a *value type*, which in the first case is Int and in the second case String. The value type is the type of values produced by the probabilistic model defined by the element.

Scala is an object-oriented language, so all Figaro elements are instances of an Element class. The Element class is parameterized by its value type. In Scala’s notation, the first element is an instance of Element[Int] while the second is an instance of Element[String].

A constant is a particular type of element that is an instance of the Constant class, which is a subclass of Element. So, more specifically, the first element above is an instance of Constant[Int]. Figaro’s representation is defined by a class hierarchy under Element.

## Atomic elements

Constants are unusual atomic elements in that they are not random. All the other built-in atomic classes contain some aspect of randomness. We illustrate some of these classes by examples.

* Flip(0.7) is an Element[Boolean] that represents the probabilistic model that produces true with probability 0.7 and false with probability 0.3.
* Select(0.2 -> 1, 0.3 -> 2, 0.5 -> 3) is an Element[Int] that represents the probabilistic model that produces 1 with probability 0.2, 2 with probability 0.3, and 3 with probability 0.5. Select can select between elements of any type, so we may also have Select(0.4 -> “a”, 0.6 -> “b”), which is an Element[String].
* Uniform(0.0, 2.0) is an Element[Double] that represents the uniform probability distribution between 0 and 2.

While Flip and Select are in the language package that was imported earlier, Uniform is in the library.atomic.continuous package that needs to be imported using

import com.cra.figaro.library.atomic.continuous.\_

or alternatively

import com.cra.figaro.\_

import library.atomic.continuous.\_

Other built-in continuous atomic classes include Normal, Exponential, Gamma, Beta, and Dirichlet, also found in the library.atomic.continuous package, while discrete elements include discrete Uniform, Geometric, Binomial, and Poisson, to be found in the library.atomic.discrete package.

## Compound elements

In Flip(0.7), the argument to Flip is a Double. There is another version of Flip in which the argument is an Element[Double]. For example, we might have

Flip(Uniform(0.0, 1.0))

which represents the probabilistic model that produces true with a probability that is uniformly distributed between 0 and 1. This is a *compound* element that is built from another element. All the atomic elements described in the previous subsection have compound versions.

Another example of a compound element is a conditional. The element

If(Flip(0.7), Constant(1), Select(0.4 -> 2, 0.6 -> 3))

represents the Element[Int] in which with probability 0.7, Constant(1) is chosen, producing 1 with probability 1, while with probability 0.3, Select(0.4 -> 2, 0.6 -> 3) is chosen, producing 2 with probability 0.4 and 3 with probability 0.6. Overall, 1 is produced with probability 0.7 \* 1 = 0.7, 2 with probability 0.3 \* 0.4 = 0.12, and 3 with probability 0.3 \* 0.6 = 0.18. The first argument to If must be an Element[Boolean], while the other two arguments must have the same value type, which also becomes the value type of the If. If can be found in the library.compound package.

## Chain

Figaro provides a useful building block for building compound elements, called *chain*. Intuitively, a chain takes a probability distribution over a “parent” element and a conditional probability distribution over a “child” element given the parent to produce a distribution over the child.

A Chain has two type parameters, T and U, where T is the value type of the parent element and U s the value type of the child element. A Chain[T,U] takes two arguments: (1) an Element[T], representing the parent element, and (2) a function from a value of type T to an Element[U], representing the conditional distribution. Scala’s notation for this type of function is T => Element[U]. For each possible value of the parent element, this function specifies an element defining the distribution over the child. The Chain itself represents the probability distribution over the child that results from this chaining. Thinking in terms of a generative process, a Chain represents the probabilistic model in which first a value of type T is produced from the first argument, then the function in the second argument is applied to this value to generate a particular Element[U], and finally a particular value of type U is produced from this element. Therefore, a Chain[T,U]is an Element[U].

For example,

Chain(Flip(0.7), (b: Boolean) =>

if (b) Constant(1); else Select(0.4 -> 2, 0.6 -> 3))

represents exactly the same probabilistic model as

If(Flip(0.7), Constant(1), Select(0.4 -> 2, 0.6 -> 3))

Let’s understand this example from the inside out. First,

if (b) Constant(1); else Select(0.4 -> 2, 0.6 -> 3)

is a Scala expression. b is a Boolean variable. If b is true, the expression produces the element Constant(1), otherwise it produces the element Select(0.4 -> 2, 0.6 -> 3). Note that this is a Scala expression, not Figaro’s conditional data structure. Now,

(b: Boolean) =>

if (b) Constant(1); else Select(0.4 -> 2, 0.6 -> 3)

is Scala’s way of defining an anonymous function from an argument named b of type Boolean to a result defined by this if expression. This function is the second argument to the chain. The first argument is the element Flip(0.7). The chain represents the probabilistic model in which first a Boolean is produced, where true is produced with probability 0.7, then the function is applied to obtain either Constant(1) or Select(0.4 -> 2, 0.6 -> 3), and finally the resulting element is used to produce an integer.

This is exactly the same model as that represented by the conditional element in the previous subsection. It is easy to see that any conditional can be represented by a chain in a similar way. Chaining is in fact an extremely powerful concept and we will see a number of examples of it in this tutorial. It is sufficient to represent all compound elements. All the compound elements in the previous section can be represented using a chain, and many of them are actually implemented that way.

## Apply and Inject

Another useful tool for building elements is Apply. Apply serves to lift Scala functions that operate on values to Figaro elements. For example,

(i: Int) => i + 5

is the Scala function that adds 5 to its integer argument.

Apply(Select(0.2 -> 1, 0.8 -> 2), (i: Int) => i + 5)

is the Figaro element representing the probabilistic model in which first either 1 or 2 is produced with the corresponding probability, and then 5 is added to the result. In the resulting probabilistic model, 6 is produced with probability 0.2 and 7 is produced with probability 0.8.

There are versions of Apply defined for functions of up to 5 arguments.

Often, one needs to apply a function to a whole sequence of arguments. Inject is provided for this. Inject takes a sequence of elements with value type T and produces an element whose value type is sequences of values of type T. In Scala notation, Inject takes a variable number of arguments of type Element[T] and produces an Element[Seq[T]]. Seq is Scala’s general sequence class, which includes lists and arrays, for example. The name “inject” refers to the fact that the sequence of arguments is injected inside the element.

In Scala, a variable number of arguments are simply listed as arguments. For example, suppose we have the elements Constant(1)and Select(0.2 -> 2, 0.8 -> 3)). Then

Inject(Constant(1), Select(0.2 -> 2, 0.8 -> 3))

represents the probabilistic model that produces the sequence (1,2) with probability 0.2 and (1,3) with probability 0.8.

If you already have a sequence, it can be turned into a variable argument list using the notation :\_\*. For example, you can use

Inject(List(Constant(1), Select(0.2 -> 2, 0.8 -> 3)):\_\*)

Using Inject and Apply, we can apply a function to a sequence of arguments. First, consider the Scala function

(xs: Seq[Int]) => xs.map((x: Int) => x + 5)

map is a Scala method that can be applied to sequences. It takes as argument a function on elements of the sequence, applies the function to every member of the sequence, and returns the resulting sequence. So, the above function takes a sequence of integers and adds 5 to every element in the sequence. Now we can write

Apply(Inject(Constant(1), Select(0.2 -> 2, 0.8 -> 3)),

(xs: Seq[Int]) => xs.map((x: Int) => x + 5))

This represents the probabilistic model that produces the sequence (2,7) with probability 0.2 and (2,8) with probability 0.8.

Finally, there are a variety of operators and functions that are defined using Apply. For example

* ^^ creates tuples. For example, ^^(x,y) where x and y are elements, creates an element of pairs. ^^ is defined for up to five arguments. The arguments can have different value types.
* If x is an element whose value type is a tuple, x.\_1 is an element that corresponds to extracting the first component of x. Similarly for \_2, \_3, \_4, and \_5.
* x === y, where x and y have the same value type, is the element that produces true whenever they are equal. Similarly for !=.
* A standard set of Boolean and arithmetic operators is provided.

# Creating models

The previous section described the basic building blocks of Figaro models. Out of these building blocks, a wide variety of models can be created. This section describes how to build a range of models.

## Basic models

One of the first things you can do with an element is to give it a name:

val burglary = Flip(0.01)

A val represents a field with a name (in this case burglary) that takes on an immutable value (in this case the element Flip(0.01)). A field is not a variable; its value cannot be changed. You can use the value of a field by referring to its name:

val alarm = If(burglary, Flip(0.9), Flip(0.1))

Recall that an element defines a process that probabilistically produces a value. If an element is referred to multiple times, it must produce the same value everywhere it appears. Consider:

val x = Flip(0.5)

val y = x === x

Although we don’t know the value, x must produce the same value on both sides of the equality test. Therefore, y produces the value true with probability 1. In contrast, in

val y = Flip(0.5) === Flip(0.5)

the left and right hand sides are distinct elements, so they need not produce the same value. Therefore, y will produce true with probability 0.5.

With the tools we have defined so far, we can easily create a Bayesian network. In the following code, CPD is a library element (based on Chain) that makes it easy to define conditional probability distributions:

import com.cra.figaro.language.\_

import com.cra.figaro.library.compound.CPD

val burglary = Flip(0.01)

val earthquake = Flip(0.0001)

val alarm = CPD(burglary, earthquake,

(false, false) -> Flip(0.001),

(false, true) -> Flip(0.1),

(true, false) -> Flip(0.9),

(true, true) -> Flip(0.99))

val johnCalls = CPD(alarm,

false -> Flip(0.01),

true -> Flip(0.7))

With CPD, every single combination of values of the parents needs to be listed. RichCPD provides a more flexible format that allows for specification of structures such as context specific independence. Each clause in a RichCPD consists of a tuple of cases, one for each parent. A case can be OneOf a set of values, NoneOf a set of values (meaning that it matches all values except for the ones listed), or \*, meaning that it accepts all values. For example:

import com.cra.figaro.library.compound.RichCPD

val x1 = Select(0.1 -> 1, 0.2 -> 2, 0.3 -> 3, 0.4 -> 4)

val x2 = Flip(0.6)

val x3 = Constant(5)

val x4 = Flip(0.8)

val y = RichCPD(x1, x2, x3, x4,

(OneOf(1, 2), \*, OneOf(5), \*) -> Flip(0.1),

(NoneOf(4), OneOf(false), \*, \*) -> Flip(0.7),

(\*, \*, NoneOf(6, 7), OneOf(true)) -> Flip(0.9),

(\*, \*, \*, OneOf(false)) -> Constant(true))

A particular combination of values of the parents is matched against each row in turn, and the first match is chosen. For example, the combination (1, false, 5, true) matches the first three rows, so the first result (Flip(0.1)) is chosen.

## Conditions and constraints

So far, we have defined probabilistic models, but we have not provided a way to post evidence about them. Evidence can be specified using either conditions or constraints.

A *condition* represents hard evidence about an element. Only values that satisfy the condition are possible. Every element has a condition, which is a function from a value of the element to a Boolean. If the element is of type Element[T], the condition is of type T => Boolean. The default condition of an element returns true for all values. The condition can be changed using setCondition:

val x1 = Select(0.1 -> 1, 0.2 -> 2, 0.3 -> 3, 0.4 -> 4)

x1.setCondition((i: Int) => i == 1 || i == 4)

which says that x1 must have value 1 or 4. We can add a condition on top of existing conditions using the addCondition method. For example, the following code says that not only must x1 equal 1 or 4, it must also be odd:

x1.addCondition((i: Int) => i % 2 == 1)

The observe method provides an easy way to specify a condition that only allows a single value. For example, to specify that x1 must have the value 2, we can use

x1.observe(2)

A *constraint* provides a way to specify a potential over an element. It is a function from a value of the element to a Double, so if the element has type Element[T], the constraint is of type T => Double.

Constraints serve multiple purposes in Figaro. One is to specify soft evidence on an element. For example, if in the above Bayesian network we think we heard John call but we’re not sure, we might introduce the constraint

johnCalls.setConstraint((b: Boolean) => if (b) 1.0; else 0.1)

This line will have the effect of making John calling 10 times more likely than not, all else being equal.

Another purpose of constraints is to define some probabilistic relationships conveniently that are more difficult to express without them. Consider the following example, in which we are modeling the process of firms bidding for a contract and one of them being selected as the winner.

import com.cra.figaro.\_

import language.\_

import library.atomic.\_

import library.compound.If

class Firm {

val efficient = Flip(0.3)

val bid = If(efficient, continuous.Uniform(5,15),

continuous.Uniform(10,20))

}

val firms = Array.fill(20)(new Firm)

val winner = discrete.Uniform(firms:\_\*)

val winningBid = Chain(winner, (f: Firm) => f.bid)

winningBid.setConstraint((d: Double) => 20 – d)

This example shows some new Scala features. First, we have a class definition (the Firm class). A class creates a type that can be instantiated to create instances. The Firm class two fields, efficient and bid. Note that bid makes use of continuous.Uniform. This is the continuous uniform element defined in the library.atomic.continuous package, but we did not import the members of this package, only the members of the library.atomic package. The reason we did things this way is that later in the example, we use the discrete uniform, and we want to be explicit about which uniform element we mean at each point.

Once we have defined the Firm class, we create an array named firms consisting of 20 instances of Firm. Array.fill(20)(new Firm) creates an array filled with the result of 20 different invocations of new Firm, each of which creates a separate instance of Firm. We then define the winner to be one of the firms, chosen uniformly. Note the notation firms:\_\*. The element discrete.Uniform takes as arguments an explicit sequence of values of variable length, for example, discrete.Uniform(1,2,5) or discrete.Uniform(“x”). Since firms is a single field representing an array, we must convert it into a sequence of arguments, which is accomplished using the :\_\* notation. The field winner represents an Element[Firm]; it is intended to mean the winning bidder, although so far we have done nothing to relate the winner to its bid.

The next line is interesting. It allows us to identify the bid of the winning bidder as an element with a name, even though we don’t know who the winner is. We can do this because even though we don’t know who the winner is, we can refer to the winner field, and because the value of winner, whatever it is, is a Firm that has a bid field, which is an element that can be referred to. It is important to realize that this Chain does not create a new element but rather refers to the element f.bid that was created previously.

Finally, we introduce the constraint, which says that a winning bid of d has weight 20 – d. This means that a winning bid of 5 is 15 times more likely than a winning bid of 19. The effect is to make the winning bid more likely to be low. Note that in this model, the winning bid is not necessarily the lowest bid. For various reasons, the lowest bidder might not win the contract, perhaps because they offer a poor quality service or they don’t have the right connections. Using a constraint, the model is specified very simply using a discrete uniform selection and a simple constraint.

Constraints are also useful for expressing undirected models such as relational Markov networks or Markov logic networks. To illustrate, we will use a version of the friends and smokers example. This example involves a number of people and their smoking habits. People have some propensity to smoke, and people are likely to have the same smoking habit as their friends.

import com.cra.figaro.language.Flip

import com.cra.figaro.library.compound.^^

class Person {

val smokes = Flip(0.6)

}

val alice, bob, clara = new Person

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

clara.smokes.observe(true)

def smokingInfluence(pair: (Boolean, Boolean)) =

if (pair.\_1 == pair.\_2) 3.0; else 1.0

for { (p1, p2) <- friends } {

^^(p1.smokes, p2.smokes).setConstraint(smokingInfluence)

}

First, we create a Person class with a smokes field. We create three different people and a network of friends, represented by a list of pairs of people. We also observe that one of the people smokes.

Now we create the constraint function smokingInfluence. This function takes a pair of Booleans, and returns 3.0 if they are the same, 1.0 if different. The intended meaning of this function is to compare the smoking habit of two friends, and say that having the same smoking habit is three times as likely as a different smoking habit, all else being equal.

Finally, we apply the constraint to all the pairs of friends. The code uses a Scala feature called a “for comprehension”. The notation for { (p1, p2) <- friends } { “do something” } iterates through all pairs of people in the friends list and executes “do something” for each pair. In this case, “do something” is “add the constraint on their smoking habits to the pair of friends”. The notation ^^(p1.smokes, p2.smokes) takes each pair of friends and creates the pair element consisting of their smoking habits. We then assign the smokingInfluence constraint to this pair.

## Classes, instances, and relationships

The object-oriented nature of Scala makes Figaro ideal for representing probabilistic models involving objects and relationships such as probabilistic relational models (PRMs). In the following example, we will see how to define general classes of object, and create instances of a class by using a subclass of the class specially designed for the instance.

In this example, we are given two possible sources and a sample that came from one of the sources, and want to determine which source the sample came from based on the strength of the match with each source.

class Source(val name: String)

abstract class Sample {

val fromSource : Element[Source]

}

class Pair(val source: Source, val sample: Sample) {

val isTheRightSource = Apply(sample.fromSource,

(s: Source) => s == source)

val distance = If(isTheRightSource,

Normal(0.0, 1.0),

Uniform(0.0, 10.0))

}

val source1 = new Source("Source 1")

val source2 = new Source("Source 2")

val sample1 = new Sample {

val fromSource = Select(0.5 -> source1, 0.5 -> source2)

}

val pair1 = new Pair(source1, sample1)

val pair2 = new Pair(source2, sample1)

pair1.distance.setCondition((d:Double) => (d > 0.15 && d < 0.25))

pair2.distance.setCondition((d:Double) => (d > 1.45 && d < 1.55))

We begin by creating classes representing sources and samples, where each sample comes from a source. Note that Sample is an abstract class, because in this class we do not say anything about what source the sample came from. We then create the Pair class representing a pair of a source and a sample. Pair has two fields: isTheRightSource, which produces true if the sample is from the source in the pair, and distance, which measures the closeness of the match between the sample and the source (lower distance means better match). The distance will tend to be smaller if the sample is from the right source but will not always be so.

Now it’s time to create some instances. Note that the Source class takes an argument which is the name of the source. When we create instances source1 and source2 of this class, we supply the name argument. Next, we create an instance of Sample. Since Sample is abstract, we need to supply a definition of fromSource. We can do that right inline here, specifying that sample1 could come either from source1 or source2, each with probability 0.5. Finally, we create pairs pairing both of the sources to sample1 and create conditions about the distances. The conditions are ranges rather than exact observations because Figaro’s algorithms currently can’t handle exact observations on continuous elements.

Using similar techniques, we can create a PRM. The following example shows the classical actors and movies PRM. There are three classes: actors, movies, and appearances relating actors to movies. Whether an actor receives an award for an appearance depends on the fame of the actor and the quality of the movie. The Figaro code for this example is as follows:

class Actor {

val famous = Flip(0.1)

}

class Movie {

val quality = Select(0.3 -> 'low, 0.5 -> 'medium, 0.2 -> 'high)

}

class Appearance(actor: Actor, movie: Movie) {

val award = CPD(movie.quality, actor.famous,

('low, false) -> Flip(0.001),

('low, true) -> Flip(0.01),

('medium, false) -> Flip(0.01),

('medium, true) -> Flip(0.05),

('high, false) -> Flip(0.05),

('high, true) -> Flip(0.2))

}

val actor1 = new Actor

val actor2 = new Actor

val actor3 = new Actor

val movie1 = new Movie

val movie2 = new Movie

val appearance1 = new Appearance(actor1, movie1)

val appearance2 = new Appearance(actor2, movie2)

val appearance3 = new Appearance(actor3, movie2)

actor3.famous.observe(true)

movie2.quality.observe('high)

// Ensure that exactly one appearance gets an award.

def uniqueAwardCondition(awards: Seq[Boolean]) =

awards.count((b: Boolean) => b) == 1

val allAwards: Element[Seq[Boolean]] =

Inject(appearances.map(\_.award):\_\*)

allAwards.setCondition(uniqueAwardCondition)

The code is self-explanatory except for the last few lines, which enforce the condition that an award is given to exactly one appearance. The function uniqueAwardCondition takes a sequence of award Booleans and returns true if exactly one Boolean in the list is true. The count method counts the number of elements in the sequence that satisfy the predicate contained in its argument. In this case the predicate is (b: Boolean) => b which is true precisely when the element of the sequence is true. So awards.count((b: Boolean) => b)counts the number of elements in the sequence that are true.

We then define the allAwards element to be the element over sequences of Booleans consisting of the award field of all the appearances. Here we have a new notation: appearances.map(\_.award). We have already seen the map method, which applies a function to every element of a sequence and returns a new sequence consisting of the results. In this case, the argument to map is the function \_.award. This is shorthand for a function of one argument in which the argument appears once in the body and in which the type of the argument is known. Here, the type of the argument is clearly an appearance. We could have used appearance => appearance.award. The notation \_.award is short for this.

## Mutable fields

Up to this point, all our Figaro programs have been purely functional. All elements have been defined by a val, and they have been immutable. In principle, all programs can be written in a purely functional style. However, this can make it quite inconvenient to represent situations in which different entities refer to each other. Scala supports both functional and non-functional styles of programming, allowing us to gain the benefits of both.

For example, let’s expand the actors and movies example so that actors have a skill, and the quality of a movie depends on the skill of the actors in it. In turn, the fame of an actor depends on the quality of the movies in which he or she has appeared. We have created a mutual dependence of actors on movies which is hard to represent in a purely functional style. We can capture it in Figaro using the following code:

class Actor {

var movies: List[Movie] = List()

lazy val skillful = Flip(0.1)

lazy val famous =

Flip(Apply(Inject(movies.map(\_.quality):\_\*), probFamous \_))

private def probFamous(qualities: Seq[Symbol]) =

if (qualities.count(\_ == 'high) >= 2) 0.8; else 0.1

}

class Movie {

var actors: List[Actor] = List()

lazy val actorsAllGood =

Apply(Inject(actors.map(\_.skillful):\_\*), (s: Seq[Boolean]) =>

!(s.contains(false)))

lazy val quality =

If(actorsAllGood,

Select(0.2: ‘low, 0.3: ‘medium, 0.5: ‘high),

Select(0.5: ‘low, 0.3: ‘medium, 0.2: ‘high))

}

class Appearance(actor: Actor, movie: Movie) {

actor.movies ::= movie

movie.actors ::= actor

lazy val award = CPD(movie.quality, actor.famous,

('low, false) -> Flip(0.001),

('low, true) -> Flip(0.01),

('medium, false) -> Flip(0.01),

('medium, true) -> Flip(0.05),

('high, false) -> Flip(0.05),

('high, true) -> Flip(0.2))

}

val actor1 = new Actor

val actor2 = new Actor

val actor3 = new Actor

val movie1 = new Movie

val movie2 = new Movie

val appearance1 = new Appearance(actor1, movie1)

val appearance2 = new Appearance(actor2, movie2)

val appearance3 = new Appearance(actor3, movie2)

actor3.famous.observe(true)

movie2.quality.observe('high)

// Ensure that exactly one appearance gets an award.

def uniqueAwardCondition(awards: Seq[Boolean]) =

awards.count((b: Boolean) => b) == 1

val allAwards: Element[Seq[Boolean]] =

Inject(appearances.map(\_.award):\_\*)

allAwards.setCondition(uniqueAwardCondition)

First, note that the Actor class has a movies field, whose purpose is to indicate the list of movies the actor has appeared in. Likewise, the Movie class has an actors field to represent the actors who appear in it. If these fields were immutable, we would need to create all the movies an actor appears in before we create the actor, and we would need to create all the actors appearing in a movie before the movie, which is impossible. Therefore, we use mutable variables, which are indicated in Scala by the var keyword.

The initial value of both movies and actors is an empty list. We add elements to them later. In fact, whenever we create an appearance, we make sure to add the movie to the actor’s list of movies and vice versa. This is achieved by the first two lines of the Appearance class. The notation

actor.movies ::= movie

is short for

actor.movies = movie :: actor.movies,

which prepends movie to the actor.movies list. The ::= notation is a variant of the familiar += notation common in many languages.

The Actor class has skillful and famous fields. Rather than an ordinary val, each of these fields is defined to be lazy val, which means that their contents are not determined until they are required by some other computation. This is necessary for us because their contents can depend on the list of movies the actor appears in. For example, whether the actor is famous depends on whether at least two movies have high quality, as defined by probFamous. (Comment on the notation: the underscore after probFamous is required here to tell Scala that what is desired is the probFamous function itself, not its application to arguments.) If famous was an ordinary val, its value (an Element[Boolean]) would be computed at the point it is defined, so it would use an empty list of movies. Because we want to use the correct list of movies in defining it, we postpone evaluating it until the movies list has been filled. For actor3, this will happen when we make the observation actor3.famous.observe(true), which we make sure to delay until after all the appearances have been created. For other actors, the famous field will be evaluated even later, during inference.

Do not hesitate to use mutation if it will help you organize your program in a logical way. In one application, I have found it convenient to use a hash table that maps concepts to their associated elements. This allowed me to create the element associated with a concept as the concept was introduced. If I later had to refer to the same concept again, I could easily access its element.

## Universes

A central concept in Figaro is a *universe*. A universe is simply a collection of elements. Reasoning algorithms operate on a universe (or, as we shall see for dependent universe reasoning, on multiple connected universes). Most of the time while using Figaro, you will not need to create a new universe and can rely on the default universe, which is just called universe. It can be accessed using

import com.cra.figaro.language.\_

Universe.universe

If you do need a different universe, you can call Universe.createNew(). This creates a new universe and sets the default universe to it. If you are going to need the old default universe, you will need a way to refer to it. You could use

val u1 = Universe.universe

val u2 = Universe.createNew()

u1 will now refer to the old default universe while u2 refers to the new one. Every element belongs to exactly one universe. Ordinarily, when an element is created, it is assigned to the current default universe. As we will see below when we talk about element collections, it is possible to assign a particular element to a different universe from the current default.

Elements can be activated or deactivated. Elements that are inactive are not operated on by reasoning algorithms. Elements are active when created. To deactivate an element e use e.deactivate(); to reactivate it, use e.activate().

You can get a list of all active elements in universe u using u.activeElements. There are many more methods of a universe that are useful for writing reasoning algorithms.

## Names, element collections, and references

Suppose we want to create a PRM in which we are uncertain about the value of an attribute whose value is itself an instance of another class (which is called reference uncertainty). For example, suppose we have the following classes and instances:

abstract class Engine { val power : Element[Symbol] }

class I4 extends Engine {

val power = Select(0.8 -> ‘low, 0.2 -> ‘high)

}

class V8 extends Engine {

val power = Select(0.8 -> ‘low, 0.2 -> ‘high)

}

object MyEngine extends Engine {

Val power = Constant(‘high)

}

class Car {

val engine: Element[Engine] =

Select(0.2 -> new I4, 0.3 -> new V8, 0.5 -> MyEngine)

val speed = CPD(

?,

‘low -> Constant(65)

‘high -> Select(0.5 -> 80, 0.5 -> 90)

}

We want the speed of the car to depend on the power of its engine, but we have uncertainty over what the engine actually is. What should we put in place of the question mark? The obvious choice is engine.power, but this does not work because engine is an Element[Engine], not an instance of Engine.

To get around this problem, Figaro provides *names* and *element collections*. Every element has a name and belongs to an element collection. By default, the name is the empty string and the element collection is the default universe at the time the element is created, which works because universes are element collections. So, most of the time, as in the tutorial to this point, you don’t have to worry about the name and element collection of an element. To assign a name and element collection to an element explicitly, you provide an extra pair of arguments when creating it.

We can give the engine a name and make it belong to the car as an element collection as follows:

class Car extends ElementCollection {

val engine: Element[Engine] =

Select(0.2 -> new I4, 0.3 -> new V6, 0.5 -> MyEngine)(

“engine”, this

)

In the first line we make the Car class inherit from ElementCollection, so that every instance of Car is an element collection. In the fourth line, we assign engine the name “engine” and add it to the instance of MyCar being created, which is referred to by this within the MyCar class. We similarly make instances of Engine element collections and assign power the name “power” within I4, V8, and MyEngine.

An element collection, like a universe, is simply a set of elements. The difference is that a universe is also a set of elements on which a reasoning algorithm operates. An element collection provides the ability to refer to an element by name. For example, if car is an instance of Car, we can use car.get[Engine](“engine”) to get at the element named “engine”. The get method takes a type parameter, which is the value type of the element being referred to. The notation [Engine] specifies this type parameter, and serves to make sure that the expression car.get[Engine](“engine”) has type Element[Engine].

The key ability of element collections that allows them to solve our puzzle is their ability to get at elements embedded in the value of an element. It uses *references* to do this. A reference is a series of names separated by dots. For example, “engine.power” is a reference. When we call car.get[Symbol](“engine.power”), it refers to the element named “power” within the *value* of the element named “engine” within the car. The value of this expression is a ReferenceElement that captures the uncertainty about which power element is actually being referred to. In a particular state of the world, i.e., an assignment of values to all elements, it is possible to determine the value of engine and therefore which power element is being referred to. So a ReferenceElement is a deterministic element that defines a way to get its value in any possible world.

So, finally, the answer to our puzzle is that in place of the question mark, we put get[Symbol](“engine.power”). This applies the get method to the instance of Car being created. Here is the full example:

abstract class Engine extends ElementCollection {

val power : Element[Symbol]

}

class I4 extends Engine {

val power = Select(0.8 -> ‘low, 0.2 -> ‘high)(“power”, this)

}

class V8 extends Engine {

val power = Select(0.8 -> ‘low, 0.2 -> ‘high)(“power”, this)

}

Object MyEngine extends Engine {

Val power = Constant(‘high)(“power”, this)

}

class Car extends ElementCollection {

val engine: Element[Engine] =

Select(0.2 -> new I4, 0.3 -> new V8, 0.5 -> MyEngine)(

“engine”, this

)

val speed = CPD(

get[Symbol](“engine.power”),

‘low -> Constant(65)

‘high -> Select(0.5 -> 80, 0.5 -> 90)

)

}

## Multi-valued References and Aggregates

The previous subsection described how to refer to elements using references that identify a single element. A feature of PRMs is the ability to define multi-valued relationships, where an entity is related to multiple entities via an attribute. In Figaro, we use multi-valued references and aggregates to capture these kinds of situations. For example:

class Component extends ElementCollection {

val f = Select(0.2 -> 2, 0.3 -> 3, 0.5 -> 5)("f", this)

}

val specialComponent1 = new Component

val specialComponent2 = new Component

def makeComponent() =

Select(0.1 -> specialComponent1,

0.2 -> specialComponent2,

0.7 -> new Component)

class Container extends ElementCollection {

val components =

Select(0.5 -> List(makeComponent()),

0.5 -> List(makeComponent(), makeComponent()))

(“components”, this)

def sum(xs: List[Int]) = (0 /: xs) (\_ + \_)

val totalComponents = getAggregate(sum)("components.f")

}

First, we create a Component class with an element named “f”. We then define two specific instances of Component. Next, we define a makeComponent function that either produces one of the specific instances or a new instance of Component that is distinct from all other instances. We then create a Container class that contains components. Now, the contained components are a list that has either one or two elements, each produced by makeComponent. We then create a totalComponents element that aggregates the values of all elements referred to by “components.f”; that is, the values of the elements named “f” in all the values of the element named “components”.

Multi-valued references have “set semantics”. If the same element appears more than once as the target of the reference, it only contributes one value to the aggregate. So, if the components list has two components, both of which are specialComponent1, whose value is 2, the value of the aggregate will be 2, not 4. On the other hand, if two different target elements both have the same value, both values contribute to the aggregate. For example, if the components are specialComponent1 and specialComponent2, and both have value 2, the value of the aggregate is 4.

A comment on the code. The definition of the sum function might look mysterious. This function takes a list of integers and returns their sum. This is a standard Scala idiom that unfortunately is a bit obscure if you’re not familiar with it. It is used to “fold” a function through a list. We begin with 0 and then repeatedly add the current result to the next element of the list until the list is exhausted. The notation (\_ + \_) is shorthand for the function that takes two arguments and adds them. The notation (0 /: xs) means that this function should be folded through xs, starting from 0.

## Open Universe Models

We close this section by showing how Figaro can be used to represent “open universe” situations. An open universe situation is one in which we don’t know what objects are there, how many there are, which objects are the same as which other objects, and so on. In our example situation, there are an unknown number of sources that is geometrically distributed. Each source is uniformly distributed between 0 and 1. There is some number of observed samples, each drawn from a single unknown source. This is the classic data association problem in which we want to determine which sample comes from which source, and in particular which two samples actually come from the same source. The Figaro code for the example is as follows:

def source(): Element[Double] = Uniform(0.0, 1.0)

val numSources = Geometric(0.9)

val sources = MakeList(numSources, source \_)

class Sample {

val sourceNum = IntSelector(numSources)

val source =

Apply(sources, sourceNum, (s: Seq[Double], i: Int) => s(i))

val position =

Chain(source, (x: Double) => Normal(x, 1.0))

}

val sample1 = new Sample

val sample2 = new Sample

val equal = sample1.source === sample2.source

sample1.position.addCondition((y: Double) => y >= 0.7 && y < 0.8)

sample2.position.addCondition((y: Double) => y >= 0.7 && y < 0.8)

Most of this should be self-explanatory at this point. There are a couple of interesting new element classes being used. MakeList takes an element over integers and a function that generates elements over a certain type (in this case doubles). It returns an element over lists of the appropriate type (in this case lists of doubles) whose length is distributed according to the first argument and in which each element is generated according to the second argument. In our example, sources is a list of sources whose length is geometrically distributed and in which each source is generated according to the source model. A notable aspect of the MakeList class, which is important for reasoning algorithms, is that the elements generating the values in the list are stored as an infinite lazy stream. Depending on the value of the first argument, the value of the MakeList is a finite prefix of the values of elements in the stream. As a result of this design, we don’t create a completely fresh list each time the length of the list changes.

The second new element class is IntSelector which takes an element over integers and returns an element that produces uniformly a number between 0 and the value of its argument (exclusive). This element can be used to generate a random index into a list produced by MakeList. IntSelector also has an interesting implementation that has benefits for reasoning algorithms (especially Metropolis-Hastings). The Randomness is an infinite stream of uniformly distributed doubles between 0 and 1. Given a particular value of the integer argument, the selected index is the one with the highest randomness value in the finite portion of the stream defined by the argument.

# Reasoning

Figaro contains a number of reasoning algorithms that allow you to do useful things with probabilistic models. First, I describe an algorithm that simply computes the range of possible values of all elements in a universe. Then, I describe three algorithms for computing the conditional probability of query elements given conditions and constraints on elements. These are variable elimination, importance sampling, and Markov chain Monte Carlo. Next, I describe algorithms for performing other kinds of reasoning. One is an importance sampling algorithm for computing the probability of evidence in a universe. The other is a variable elimination algorithm for computing the most likely values of elements given the evidence. Finally, I describe two additional features of the reasoning: the ability to reason across multiple universes, and a way to use abstractions in reasoning algorithms.

## Computing ranges

It is possible to compute the set of possible values of in a universe, as long as expanding the probabilistic model of the universe does not (1) result in generating an infinite number of elements, and (2) result in results in an infinite number of values for an element (3) involves an element class for which getting the range has not been implemented.

To explain (1), computing the possible values of a chain requires computing the possible values of the arguments and, for each value, generating the appropriate element and computing all its possible values. If the generated element also contains a chain, it will require recursively generating new elements for all possible values of the contained chain’s arguments. This could potentially lead to an infinite recursion, in which case computing ranges will not terminate.

For (2), most built in element classes have a finite number of possible values. Exceptions are the atomic continuous classes like Uniform and Normal.

To compute the values of elements in universe u, you first create a Values object using

import com.cra.figaro.algorithm.\_

val values = Values(u)

You can also create a Values object for the current universe simply with

val values = Values()

values can then be used to get the possible values of any object. For example,

val e1 = Flip(0.7)

val e2 = If(e1, Select(.2 -> 1, .8 -> 2), Select(.4 -> 2, .6 -> 3)

val values = Values()

values(e2)

returns a Set[Int] equal to { 1, 2, 3 }.

If you are only interested in getting the range of the single element e2, you can use the shorthand Values()(e2). However, if you want the range of multiple elements, you are better off creating a Values object and applying it repeatedly to get the range of the different elements. The reason is that within a Values object, computing the range of an element is memorized, meaning that the range is only computed once for each object and then stored for future use.

## Exact inference using variable elimination

Figaro provides the ability to perform exact inference using variable elimination. The algorithm works in three steps:

1. Expand the universe to include all elements generated in any possible world.
2. Convert each element into a factor.
3. Apply variable elimination to all the factors.

Step 1, like for range computation, requires that the expansion terminate in a finite amount of time. Step 2 requires that each element be of a class that can be converted into a set of factors. Every built-in class can be converted into a set of factors except for atomic continuous classes with infinite range, although see later in the section on abstractions how to make variable elimination work for continuous classes. Also see later in the section on creating a new element class how to specify a way to convert a new class into a set of factors.

To use variable elimination, you need to specify a set of query elements whose conditional probability you want to compute given the evidence. For example,

import com.cra.figaro.algorithm.factored.\_

val e1 = Select(0.25 -> 0.3, 0.25 -> 0.5, 0.25 -> 0.7, 0.25 -> 0.9)

val e2 = Flip(e1)

val e3 = If(e2, Select(0.3 -> 1, 0.7 -> 2), Constant(2))

e3.setCondition((i: Int) => i == 2)

val ve = VariableElimination(e2)

This will create a VariableElimination object that will apply variable elimination to the universe containing e1, e2, and e3, leaving query variable e2 uneliminated. However, it won’t perform the variable elimination immediately. To tell it to perform variable elimination, you have to say

ve.start()

When this call terminates, you can use ve to answer queries using three methods:

ve.distribution(e2) will return a stream containing possible values of e2 with their associated probabilities.

ve.probability(e2, (b: Boolean) => b) will return the probability that the value of e2 satisfies the given predicate, which in this case is (b: Boolean) => b. This is the function that takes a Boolean argument and returns true precisely if its argument is true. So, in other words, the expression computes the probability that e2 has value true. The probability method also provides a shorthand the species a value as the second argument instead of a predicate and returns the probability the element takes that specific value. So, for the previous example, we could have written ve.probability(e2, true).

ve.expectation(e2, (b: Boolean) => if (b) 3.0; else 1.5) returns the expectation of the given function applied to e2.

Once you are done with the results of variable elimination, you can call ve.kill(). This has the effect of freeing up memory used for the results.

These methods start, kill, distribution, probability, and expectation are a uniform interface to all reasoning algorithms that compute the conditional probability of query variables given evidence. We will see below how this interface is extended for anytime algorithms.

## Importance sampling

The algorithm called importance sampling is actually a combination of importance and rejection sampling. It uses a simple forward sampling approach. When it encounters a condition, it checks to see if the condition is satisfied and rejects if it is not. When it encounters a constraint, it multiplies the weight of the sample by the value of the constraint.

Unlike variable elimination, this algorithm can be applied to models whose expansion produces an infinite number of elements, provided any particular possible world only requires a finite number of elements to be generated. Also, this algorithm works for atomic continuous models. In addition, as an approximate algorithm, it can produce reasonably accurate answers much more quickly than the exact variable elimination.

The interface to importance sampling is very similar to that to variable elimination. For example,

import com.cra.figaro.algorithm.sampling.\_

val e1 = Select(0.25 -> 0.3, 0.25 -> 0.5, 0.25 -> 0.7, 0.25 -> 0.9)

val e2 = Flip(e1)

val e3 = If(e2, Select(0.3 -> 1, 0.7 -> 2), Constant(2))

e3.setCondition((i: Int) => i == 2)

val imp = Importance(e2, 10000)

The second argument to Importance is an indication of how many samples the algorithm should take. After calling imp.start(), you can use the methods distribution, probability, and expectation to answer queries.

Figaro also provides an anytime importance sampling algorithm that runs in a separate thread and continues to accumulate samples until it is stopped. Two additional methods are provided in the interface. imp.stop() stops it from accumulating samples, while imp.restart() starts it going again, carrying on from where it left off before. In addition, the kill method has the additional effect of killing the thread, so it is essential that it be called when you are finished with the Importance object. To create an anytime importance algorithm, simply omit the number of samples argument to Importance. A typical way of using anytime importance sampling, allowing it to run for one second, is as follows:

val imp = Importance(e2)

imp.start()

Thread.sleep(1000)

imp.stop()

println(imp.probability(e2, (b: Boolean) => b))

imp.kill()

## Markov chain Monte Carlo

Figaro provides a Metropolis-Hastings Markov chain Monte Carlo algorithm. Metropolis-Hastings uses a proposal distribution to propose a new state at each step of the algorithm, and either accepts or rejects the proposal. In Figaro, a proposal involves proposing new randomnesses for any number of elements. After proposing these new randomnesses, any element that depends on those randomnesses must have its value updated. Recall that the value of an element is a deterministic function of its randomness and the values of its arguments, so this update process is a deterministic result of the randomness proposal.

Proposing the randomness of an element involves calling the nextRandomness method of the element, which takes the current value of the randomness as the argument. nextRandomness has been implemented for all the built-in model classes, so you will not need to worry about it unless you define your own class. See the section on creating a new element class for details.

Computing the acceptance probability requires computing the ratio of the element’s constraint of the new value divided by the constraint of the old value. Ordinarily, this is achieved by applying the constraint to the new and old value separately and taking the ratio. However, sometimes we want to define a constraint on a large data structure, and applying the constraint to either the new or old value will produce overflow or underflow, so the ratio won’t be well defined. It might be the case that the ratio is well defined even though the constraints are large, since only a small part of the data structure changes in a single Metropolis-Hastings situation. For example, we might want to define a constraint on an ordering, penalizing the number of items out of order. The total number of items out of order might be large, but if a single iteration consists of swapping two elements, the number that change might be small. For this reason, an element contains a score method that takes the old value and the new value and produces the ratio of the constraint of the new value to the old value.

Figaro allows the user to specify which elements get proposed using a *proposal scheme*. Figaro also provides a default proposal scheme that simply chooses a non-deterministic element in the universe uniformly at random and proposes a new randomness for it. To create an anytime Metropolis-Hastings algorithm using the default proposal scheme, use

import com.cra.figaro.algorithm.sampling.\_

val e1 = Select(0.25 -> 0.3, 0.25 -> 0.5, 0.25 -> 0.7, 0.25 -> 0.9)

val e2 = Flip(e1)

val e3 = If(e2, Select(0.3 -> 1, 0.7 -> 2), Constant(2))

e3.setCondition((i: Int) => i == 2)

val mh = MetropolisHastings(ProposalScheme.default, e2)

Metropolis-Hastings takes two additional optional arguments. The first represents the burn-in, which is the number of proposal steps the algorithm goes through before collecting samples, while the second is the number of proposal steps between samples. The default burn-in is 0, while the default interval is 1. These arguments appear before the query elements.

To use a one-time (i.e., non-anytime) Metropolis-Hastings algorithm, simply provide the number of samples as the first argument. The Metropolis-Hastings algorithm uses the same interface as importance sampling.

#### Defining a proposal scheme

A proposal scheme is an instance of the ProposalScheme class. A number of constructs are provided to help define proposal schemes. We will illustrate some of them using the first movie example from the section titled “Classes, instances, and relationships”. The default proposal scheme does not work well for this example because it is unlikely to maintain the condition that exactly one appearance is awarded. A better proposal scheme will maintain this condition by always replacing one awarded appearance with another.

The SwitchingFlip class is defined to facilitate this. SwitchingFlip is just like a regular Flip except that its nextRandomness method always returns the opposite of its argument. The award attribute of Appearance is defined to be a SwitchingFlip. The value of SwitchingFlip is that now we can change which appearance gets awarded by proposing the award attribute of the appearance that is currently awarded and one other appearance. This idea is implemented in the function switchAwards, which returns a proposal scheme depending on the current state of awards.

def switchAwards(): ProposalScheme = {

val (awarded, unawarded) = appearances.partition(\_.award.value)

awarded.length match {

case 1 =>

val other = unawarded(random.nextInt(numAppearances - 1))

ProposalScheme(awarded(0).award, other.award)

case 0 =>

ProposalScheme(

appearances(random.nextInt(numAppearances)

).award)

case \_ =>

ProposalScheme(awarded(random.nextInt(awarded.length)).award)

}

}

switchAwards first makes lists of the awarded and unawarded appearances. Then, if exactly one appearance is awarded, it chooses one unawarded element and returns ProposalScheme(awarded(0).award, other.award). This scheme first proposes the award attribute of the only awarded appearance and then proposes the award attribute of the chosen unawarded appearance. In general, ProposalScheme with a sequence of elements as arguments proposes each of them in turn. Moving on, if zero appearances are currently awarded, it proposes a single randomly chosen appearance’s award to bring the number of awarded appearances to one. If more than one appearance is currently awarded, it proposes one of the awarded appearance’s awards to reduce the number of awarded appearances.

In this example, we will also sometimes want to propose the fame of actors or the quality of movies. To achieve this, we use a DisjointScheme, which returns various proposal schemes with different probabilities. This is implemented in the following chooseScheme function:

private def chooseScheme(): ProposalScheme = {

DisjointScheme(

(0.5, () => switchAwards()),

(0.25, () =>

ProposalScheme(actors(random.nextInt(numActors)).famous)),

(0.25, () =>

ProposalScheme(movies(random.nextInt(numMovies)).quality))

)

}

In general, the proposal scheme argument of MetropolisHastings is actually a function of zero arguments that returns a ProposalScheme. The ProposalScheme.default is just that. Since chooseScheme is the same, it can be passed directly to MetropolisHastings. So we can call

val alg =

MetropolisHastings(10000, chooseScheme, 1000, appearance1.award,

appearance2.award, appearance3.award)

In some cases, it might be useful to have the decision as to which later elements to propose depend on the proposed values of earlier elements. TypedScheme is provided for this purpose. It has a type parameter T which is the value type of the first element to be proposed. The first argument to TypedScheme is a function of zero arguments that returns an Element[T]. The second argument is a function from a value of type T to an Option[ProposalScheme]. An Option[ProposalScheme], as its name implies, is an optional proposal scheme. It can take the value None, meaning that there is no proposal scheme, or the value Some(ps), where ps is a proposal scheme. This allows the proposed value of the first element to determine, first of all, whether there will be any more proposals, and if there will be more proposals, what the subsequent proposal scheme will be.

#### Caching and non-caching Chain

In designing a Metropolis-Hastings algorithm using chains, there are two design options. In one option, when the chain’s function is invoked on an argument to produce a result element, the element is cached, and retrieved every time the function is applied to the same argument. In the second option, the result is not cached, but is instead generated fresh every time. Both options have advantages.

The standard advantage of caching is that it can save significant time if the function is executed repeatedly. In Metropolis-Hastings, there is an important additional advantage. After an element is created, it may go through a sequence of proposals and eventually reach a region of high probability. Caching allows this work to be saved and reused every time the parent of the chain returns to the same value. Without caching, when the parent changes from the original value, we free up the memory taken by the result element. If at a later stage the parent returns to the same original value, we need to begin the search process from scratch.

However, the standard disadvantage of caching is that it uses more memory. In particular, a different element is stored for every value of the parent that has been seen, and is never released. If the parent can have a large or infinite number of possible values, this can lead to exhausting the memory of the machine.

A good rule of thumb is that caching is usually better for elements with discrete parents with relatively few values, and not caching is better for elements with continuous parents. Since the best option is different for different cases, Figaro allows you to specify which option explicitly you want. If you want a caching chain, use the CachingChain class. For a non-caching chain, use the NonCachingChain class. The syntax for both is the same as for Chain. Chain on its own defaults to non-caching, but if you are really sure you want a non-caching chain, we recommend using NonCachingChain explicitly.

#### Debugging Metropolis-Hastings

Designing good proposal schemes is more of an art than a science and can be quite challenging. It took me a fair amount of time before I found a good proposal scheme for the movies example. It also required me implementing the SwitchingFlip element class, which, as we will see below, is not difficult. Unfortunately, a problem with Metropolis-Hastings algorithms is that they can be quite difficult to debug. Developing good methodologies and tools for debugging Metropolis-Hastings is an important research problem. For now, Figaro provides a couple of tools that I have found to be useful.

The Metropolis-Hastings class has a debug variable, which by default is set to false. If you set it to true, you get debugging output when you run the algorithm. This includes every element that is proposed or updated and whether each proposal is accepted or rejected. The debugging output uses the names of elements, so to make use of it, you need to give the elements you are interested in a name.

In addition, if you have a Metropolis-Hastings object mh, you can define an initial state by setting the values of elements. Then call mh.test and provide it a number of samples to generate. It will repeatedly propose a new state from the initial state and either accept or reject it, restoring to the original state each time. You can provide a sequence of predicates, and it will report how often each predicate was satisfied after one step of Metropolis-Hastings from the initial state. You can also provide a sequence of elements to track, and it will report how often each element is proposed. For example, in the movies example, you could set the initial state to be one in which exactly one appearance is awarded and test the fraction of times this condition holds after one step.

## Probability of evidence algorithm

The previous three algorithms all computed the conditional probability of query variables given evidence. Sometimes we just want to compute the probability of evidence, which here means the probability of the conditions, taking into account the constraints. Figaro currently provides one algorithm for this, which is a simple sampling algorithm. At each step, it forward samples values for all elements and weights the sample by the value of the constraints. It computes the weighted fraction of samples for which the conditions are satisfied.

Figaro provides anytime and one-time versions of this algorithm. To create an anytime version, simply use ProbEvidenceSampler(). For a one-time algorithm, pass the number of samples. The algorithms use the same start, stop, resume, and kill interface as the previous algorithms. After starting the algorithm, call the probabilityOfEvidence to get the probability of evidence.

## Computing the most likely values of elements

Rather than computing a probability distribution over the values of elements given evidence, a natural question to ask is “What are the most likely values of all the elements given the available evidence?” This is known as computing the most probable explanation (MPE) of the evidence. An example that shows how to compute the MPE is:

createNew()

val e1 = Flip(0.5)

e1.setConstraint((b: Boolean) => if (b) 3.0; else 1.0)

val e2 = If(e1, Flip(0.4), Flip(0.9))

val e3 = If(e1, Flip(0.52), Flip(0.4))

val e4 = e2 === e3

e4.observe(true)

val alg = MPEVariableElimination()

alg.start()

println(alg.mostLikelyValue(e1)) // should print true

println(alg.mostLikelyValue(e2)) // should print false

println(alg.mostLikelyValue(e3)) // should print false

println(alg.mostLikelyValue(e4)) // should print true

## Reasoning with dependent universes

Earlier we saw that variable elimination does not work for all models. One way to get around this in some cases is to use dependent universes. As an example, consider a problem in which we have a number of sources and a number of sample points, and we want to associate each point with its source. The distance between a point and a source depends on whether it is its correct source or not. We can capture this situation with the following model:

class Source(val name: Name)

abstract class Sample(val name: Name) {

val fromSource : Element[Source]

}

class Pair(val source: Source, val sample: Sample) {

val isTheRightSource =

Apply(sample.fromSource, (s: Source) => s == source)

val rightSourceDistance = Normal(0.0, 1.0)

val wrongSourceDistance = Uniform(0.0, 10.0)

val distance =

If(isTheRightSource, rightSourceDistance, wrongSourceDistance)

}

Now, suppose that each sample has a set of potential sources, and at most one sample can come from each source. This creates a constraint over the samples that could come from each source. First, we create some sources, samples, and pair them up.

val source1 = new Source("Source 1")

val source2 = new Source("Source 2")

val source3 = new Source("Source 3")

val sample1 = new Sample("Sample 1") {

val fromSource = Select(0.5 -> source1, 0.5 -> source2)

}

val sample2 = new Sample("Sample 2") {

val fromSource = Select(0.3 -> source1, 0.7 -> source3)

}

val pair1 = new Pair(source1, sample1)

val pair2 = new Pair(source2, sample1)

val pair3 = new Pair(source1, sample2)

val pair4 = new Pair(source3, sample2)

Note that Sample is an abstract class, so when we create particular samples we must provide a value for fromSource. Now we can enforce the constraint as follows:

val values = Values()

val samples = List(sample1, sample2)

for {

(firstSample, secondSample) <- upperTriangle(samples)

sources1 = values(firstSample.fromSource)

sources2 = values(secondSample.fromSource)

if sources1.intersect(sources2).nonEmpty

} {

^^(firstSample.fromSource, secondSample.fromSource).addCondition(

(p: (Source, Source)) => p.\_1 != p.\_2)

}

The first thing we do is create a Values object, because we will need to repeatedly get the possible sources of each sample. The for comprehension first generates all pairs of elements in the samples list in which the first element precedes the second in the list. It then sees if the two samples have a possible source in common. If they do, it imposes a constraint on the pair of sources of the two samples saying that they must be different. We go through this process to avoid setting a constraint on the source variables of all pairs of samples, which would lead them to be one large clique.

Depending on the structure of which samples can come from which sources, we might want to solve this problem using variable elimination. Unfortunately, the distances are defined by atomic continuous elements that cannot be used in variable elimination. The solution is to use dependent universes. We create a universe for each Pair as follows:

class Pair(val source: Source, val sample: Sample) {

val universe = new Universe(List(sample.fromSource))

val isTheRightSource =

Apply(sample.fromSource, (s: Source) => s == source)(

"", universe)

val rightSourceDistance = Normal(0.0, 1.0)("", universe)

val wrongSourceDistance = Uniform(0.0, 10.0)("", universe)

val distance =

If(isTheRightSource, rightSourceDistance,

wrongSourceDistance)("", universe)

}

Now, we can use variable elimination and condition each of the source assignment on the probability of the evidence in the corresponding dependent universe. To do this, we pass a list of the dependent universes as extra arguments to variable elimination, along with a function that provides the algorithm to use to compute the probability of evidence in a dependent universe, as follows:

pair1.distance.addCondition((d: Double) => d > 0.5 && d < 0.6)

pair2.distance.addCondition((d: Double) => d > 1.5 && d < 1.6)

pair3.distance.addCondition((d: Double) => d > 2.5 && d < 2.6)

pair4.distance.addCondition((d: Double) => d > 0.5 && d < 0.6)

def peAlg(universe: Universe) =

probEvidenceSampler(100000)(universe)

val alg = VariableElimination(

List(pair1.universe, pair2.universe,

pair3.universe, pair4.universe),

peAlg \_,

sample1.fromSource

)

## Abstractions

An alternative way to dealing with elements with many possible values, such as continuous elements, is to map the values to a smaller abstract space of values. An element can have *pragmas*, which are instructions to algorithms on how to deal with the element. The only pragmas currently defined are abstractions, but more might be defined in the future. To add an abstraction to an element, use the element’s addPragma method.

Let us build abstractions in steps. We start with a PointMapper. A point mapper defines a map method that takes a concrete point and a set of possible abstract points and chooses one of the abstract points. A natural point mapper for continuous elements maps each continuous value to the closest abstract point.

Next, we define an AbstractionScheme. In addition to being a point mapper, an abstraction scheme also provides a select method that takes a set of concrete points and a target number of abstract points and chooses a set of abstract points from the concrete points of the given size. A default abstraction scheme is provided for continuous elements that provides a uniform discretization of the given concrete values. More intelligent abstraction schemes that perform other discretizations can easily be developed.

An Abstraction consists of a target number of abstract points, a desired number of concrete points per abstract point from which to generate the abstract points (which defaults to 10), and an abstraction scheme. An example of using abstractions to discretize continuous elements is as follows:

createNew()

val flip = Flip(0.5)

val uniform1 = Uniform(0.0, 1.0)

val uniform2 = Uniform(1.0, 2.0)

val chain = If(flip, uniform1, uniform2)

val apply = Apply(chain, (d: Double) => d + 1.0)

apply.addConstraint((d: Double) => d)

uniform1.addPragma(Abstraction(10))

uniform2.addPragma(Abstraction(10))

chain.addPragma(Abstraction(10))

apply.addPragma(Abstraction(10))

val ve = VariableElimination(flip)

ve.start()

println(ve.probability(flip, true)) // should print about 0.4

It is up to individual algorithms to decide whether and to use a pragma such as an abstraction. For example, importance sampling, which has no difficulty with elements with many possible values, ignores abstractions. The process of computing ranges, which is a subroutine of variable elimination and can also be used in other algorithms, does use abstractions.

The process used by range computation to determine the range of an abstract element is as follows. First it generates concrete values, then selects the abstract values from the concrete values. If the element is atomic, it generates the concrete points directly. The number of concrete values is equal to the number of abstract values times the number of concrete values per abstract value, both of which can be specified. If the element is compound, it uses the sets of the values of the element’s arguments and the definition of the element to produce concrete values. Remember that the sets of values of the arguments (e.g., for the apply in the above example) may themselves be the result of abstractions. Once it has generated the concrete points, the range computation calls the select method of the abstraction scheme associated with the element to generate the abstract values.

# Dynamic models and filtering

Figaro provides constructs to create dynamic probabilistic programs that describe a domain that changes over time. All the power of the language can be used in creating dynamic programs. A dynamic probabilistic program consists of two parts: (1) an initial model, which is a universe, describing the distribution over the initial state, and (2) a transition model, which is a function from a universe representing the distribution at one time point to a universe representing the distribution at the next time point.

The following code shows the typical method for creating initial and transition models:

val u1 = createNew()

val f = Flip(0.2)("f", u1)

def trans(previousUniverse: Universe): Universe = {

createNew()

val b: Element[Boolean] = previousUniverse.get("f")

val i = If(b, Flip(0.8), Flip(0.3))("f", universe)

universe

}

The second line creates a new universe for the initial model and assigns it to a variable so that we can use it later. We then define an element to appear in the initial model and give it the name “f”. When a name is given explicitly to an element, you also need to specify the universe, which in this case is the initial universe.

We then define the transition model. It takes the previous universe as argument and returns a universe. The first thing it does is create a new universe, which is returned at the end of defining the transition model. It then creates an element named “f” that depends on the previous value of “f”. The previous value of “f” is the value of the element named “f” in the previous universe. We get at this element using previousUniverse.get("f"). Using this procedure, we can create any manner of dependency between the previous state and the current state by referring to elements in the previous universe by reference.

#### Particle filtering

Currently, the only algorithm provided by Figaro for reasoning about dynamic models is a vanilla particle filter. To create the particle filter, use

val pf = ParticleFilter(u1, trans, numParticles)

where u1 is the initial universe, trans is the transition model, and numParticles is the number of particles the algorithm should produce at each time step.

One tricky aspect about using a particle filter is that the universes are produced by a function, so it is hard to get a handle on them to observe evidence. Instead, evidence is observed by referring to an element by name, which is the same in every universe. To specify evidence, we create an instance of the Evidence class. Evidence is an abstract class with three concrete subclasses: Condition, Constraint, and Observation, whose meaning should be clear from the name. For example, Observation specifies that an element takes on a particular value, as in Observation(true). Once we have created an item of evidence, we associate it with a name using the NamedEvidence class, for example with NamedEvidence(“f1”, Observation(true)).

To tell the particle filter to create the initial set of particles from the initial model, we call the start method. The filter then waits until it is told it is time to move to the next time step. To tell the particle filter to move forward in time and tell it the evidence at the new time point, we call the advanceTime method, which takes a list of NamedEvidence as argument. For example,

pf.start()

pf.advanceTime(List(NamedEvidence("f2", Observation(**true**))))

pf.advanceTime(List(NamedEvidence("f2", Observation(**false**))))

This creates the initial particles and advances two time steps with different evidence at each time.

The query methods provided for a filtering algorithm are currentDistribution, currentExpectation, and currentProbability. These are similar to the corresponding methods for algorithms that compute conditional probabilities for static models, except that they return the distribution, expectation, or probability at the current point in time. For example,

pf.start()

pf.advanceTime(List(NamedEvidence("f2", Observation(**true**))))

pf.advanceTime(List(NamedEvidence("f2", Observation(**false**))))

pf.probability(“f1”, true)

returns the probability that the element named “f1” is true after two time steps, given that “f2” was true in the first time step and false in the second.

# Creating a new element class

For many applications, Figaro’s built-in element classes will suffice. However, if you do need a new element class, it is usually not hard to create one. The easiest way to create a new class is to inherit from an existing class. We describe how to do this for atomic and compound classes. Then we describe how to create an atomic or compound class without inheritance. After that, we describe how to make a class usable by range computation and variable elimination. Finally, we show how to create a class with special behavior under Metropolis-Hastings.

More examples of element classes can be found under com.cra.figaro.library. If you do create a new element class and think it might be generally useful, we would appreciate if you would consider sharing it, either as a library or possibly as part of a future Figaro release.

## Creating an atomic class with inheritance

The easiest way to create a new class is to inherit from an existing class. For example, a discrete uniform distribution is just a special case of a discrete selection where every element has the same probability. We can create this element class simply with

class AtomicUniform[T](name: Name[T], options: Seq[T], collection: ElementCollection)

extends AtomicSelect

[T](name, options.toList map b, collection) with Atomic[T] {

override def toString = "Uniform(" + options.mkString(", ") + ")"

}

The atomic uniform class is one for which the options are explicitly specified values of type T, as opposed to the compound uniform in which the options are elements over values of type T. The atomic uniform class takes three arguments: a name (which every class takes), an element collection (likewise), and a sequence specifying the options the uniform distribution can produce. The class inherits the AtomicSelect class, which represents selection over a discrete set of options with their associated probabilities.

To carry out the inheritance, we need to transform the sequence of options into a list of (probability, value) pairs, which are the argument to AtomicSelect. This is accomplished by the expression options.toList map (1.0 -> \_). This turns the sequence of options into a list and applies to all elements of the list the function that maps an option to the pair (1.0, option).

Let us understand the notation (1.0 -> \_). This is Scala shorthand for the function which maps an option to the pair (1.0, option). There are two things in this shorthand worth noting. First, 1.0 -> \_ is another way of describing the pair (1.0, \_). I find it to be a more descriptive way of saying “with probability 1.0, you get \_,” rather than just “the pair of 1.0 and \_.” Second, \_ denotes the argument to the function, when you know you are defining a function. Here, you know you are defining a function because it appears in the context of applying a function to all elements of a list. This underscore notation can only be used when the argument appears exactly once in the body of the function. Thus (1.0 -> \_) is Scala’s shorthand for the function (t: T) => (1.0, t). It really doesn’t matter if this shorthand is meaningful to you; feel free to use the longer version wherever you want. Note that the probabilities in the AtomicSelect are not normalized; AtomicSelect automatically takes care of the normalization.

The only thing the body of AtomicUniform does is to override the toString method that every Scala class has. The method produces something meaningful when the element is converted into a string. options.mkString(“, “) creates a string consisting of each of the options separated by a comma and a space.

A problem with the above class definition is that to create an instance, you have to say

new AtomicUniform(name, options, collection)

i.e., you have to use the keyword new, you have to call it AtomicUniform (as opposed to CompoundUniform, described below), and you have to supply the name and collection explicitly. To provide a more convenient way to create instances, we provide the following code:

object Uniform {

def apply[T](options: T\*)(implicit name: Name[T], collection:

ElementCollection) =

new AtomicUniform(name, options, collection)

}

Using this definition, you can simply say Uniform(options)to create an atomic uniform element.

This snippet uses a number of features of Scala. It is not important that you understand all these features in detail, as the snippet shows a pattern that can be copied directly to your class.

First, an *object* is a Scala class that only has a single instance. There can be an object with the same name as a class; in that case they are called *companions*. The object holds what are commonly known as static methods, i.e., methods that don’t depend on the state of a specific instance, as well as methods that create elements of the class. The latter are known as *factory methods*. In our example, the factory method creates a new instance of AtomicUniform.

Second, a method named apply is special. It can be invoked simply by providing the name of the object and listing its arguments in parentheses. So instead of saying Uniform.apply(options), you can say Uniform(options). Methods named apply are often used for defining factory constructors.

Third, Scala allows *curried functions*. These are functions that can be applied to one set of arguments to yield a function that can be applied to more arguments. Scala indicates this by providing multiple argument lists to a function. So, in our example, the first argument list consists of the sequence of options, while the second consists of the name and element collection.

Finally, the second argument list to apply is *implicit*. This means that you can leave out the argument list and Scala will implicitly fill it in with special values defined elsewhere. In this case, “” is the implicit value of type Name and the current universe is the implicit value of type ElementCollection. This is why you don’t have to supply these arguments when you create an element unless you explicitly want to specify a different name or element collection.

## Creating a compound class with inheritance

Most compound classes inherit from either Chain or Apply. We will show an example of both.

#### Inheriting from Chain

First, let us continue with discrete uniform elements, but now let us define one whose argument is itself a sequence of elements. We define it as follows:

class CompoundUniform[T](name: Name[T], options: Seq[Element[T]], collection: ElementCollection)

extends CachingChain[Seq[T],T] (

name,

new Inject("", options, collection),

(options: Seq[T]) => new AtomicUniform("", options, collection),

collection

) {

override def toString = "Uniform(" + options.mkString(", ") + ")"

}

First, note that it inherits from CachingChain. When you inherit from Chain, you have to specify whether it is caching or not. Here we chose caching, but if you want a non-caching version, all you have to do is create a new class that is identical to this one except that it inherits from NonCachingChain. Also, when you inherit from a class, you have to explicitly pass along the name and collection arguments.

The operation of the chain can be thought of as follows: first, produce specific values for each of the options. Then, given such a specific set of values, create an atomic uniform element over those values. Finally, generate a specific value from the atomic uniform element, i.e., a uniformly chosen value from those values.

The meat of the definition is the second and third arguments. The second argument defines the parent of the chain, which is the element that generates the sequence of option values. We have to convert the sequence of elements that are the arguments to CompoundUniform to an element over sequences; this is achieved using Inject. The third argument defines the function of the chain. Given a particular set of values of the options, it creates an atomic uniform with those values.

That’s all there is to it. The Uniform object also defines an apply method that allows you to create compound uniform elements conveniently.

#### Inheriting from Apply

Inheriting from Apply will typically be used when you want to create an element class that captures a common function. When you inherit from Apply, you have to explicitly inherit from the Apply class that has the right number of arguments. For example, if your function has two arguments, you inherit from Apply2. For example, the element class that represents the comparison of the values of two elements for equality is defined by

class Eq[T](name: Name[Boolean], arg1: Element[T], arg2: Element[T], collection: ElementCollection)

extends Apply2(name, arg1, arg2, (t1: T, t2: T) => t1 == t2, collection) {

override def toString = arg1.toString + " === " + arg2.toString

}

In addition to the name and element collection, we need to pass to Apply2 the two arguments and the function to be applied.

## Creating an atomic class without inheritance

Since most atomic classes are non-deterministic and creating a non-deterministic class requires more work than a deterministic class, we will use a non-deterministic example, specifically, continuous uniform elements. A non-deterministic atomic element class needs to define the following things:

* The Randomness type
* A generateRandomness method that produces a randomness according to an appropriate generation process
* A generateValue method that deterministically generates the value of the element given its randomness
* A density method that returns the density of any possible value

The class that defines continuous uniform distributions between given lower and upper bounds is defined as follows:

class AtomicUniform(name: Name[Double], val lower: Double, val upper: Double, collection: ElementCollection)

extends Element[Double](name, collection) with Atomic[Double] {

type Randomness = Double

val diff = upper - lower

def generateRandomness() = random.nextDouble() \* diff + lower

def generateValue(rand: Randomness) = rand

val constantDensity = 1.0 / diff

def density(d: Double) = if (d >= lower && d < upper) constantDensity; else 0.0

override def toString = "Uniform(" + lower + ", " + upper + ")"

}

This should be self-explanatory given everything we’ve seen so far. In this class, we defined generateRandomness to actually produce the value, and generateValue to simply pass it along, but a different design would have been possible. For other atomic non-deterministic classes, the logic of the methods would be richer, but the general structure would be the same.

## Creating a compound class without inheritance

Creating a compound class without inheritance is unusual, as Chain and Apply are ubiquitous. The most common use will probably be to create variants of Chain and Apply that take more arguments than the built-in classes. To do that, you should take the code for Chain or Apply as a model and base your new class on that. Otherwise, for a deterministic compound class, you need to define the following elements:

* The args method that returns a list of the elements on which this element depends. Make sure this is a def, not a val. (Otherwise, you might run into a nasty Scala issue with abstract fields in a superclass being initialized in a concrete subclass. When an instance of the subclass is constructed, the superclass instance is constructed first, and a superclass of all element classes is the Element class, which uses args in its constructor. If args were a val, it would be uninitialized at that time and throw a null pointer exception.)
* The generateValue method that takes no arguments and produces the value of the element as a function of the values of the arguments of the element.

For example, Apply1 is defined by

class Apply1[T1,U](name: Name[U], val arg1: Element[T1], val fn: T1 => U, collection: ElementCollection)

extends Deterministic[U](name, collection) {

def args: List[Element[\_]] = List(arg1)

type Arg1Type = T1

def generateValue() = fn(arg1.value)

override def toString = "Apply(" + arg1 + ", " + fn + ")"

}

For non-deterministic classes, you need to define the additional elements Randomness, generateRandomness, and density, as before.

## Making a class usable by variable elimination

Certain algorithms rely on element classes being able to support specific functionality. For example, computing ranges requires that it be possible to enumerate the values of every element in the universe. One way to make a new element class support value enumeration would be to modify the code that enumerates values in Values.scala, This approach would not be modular; it is undesirable for a user to have to modify library code.

Figaro provides a different solution. There is a trait called ValuesMaker that characterizes element classes for which values can be enumerated. If you want your element class to support range computation, make it extend ValuesMaker and have it implement the makeValues method, which produces an enumeration of the possible values of the element. For example, we might want to enumerate the possible values of an atomic binomial element. If n is the number of trials of the binomial, we can define the function

def makeValues: Set[Int] = (for { i <- 0 to n } yield i).toSet

The makeValues method returns a set of values. For a binomial, this is simply all the integers from 0 to the number of trials. This set is computed through a for comprehension whose result is turned into a set.

We also make AtomicBinomial extend ValuesMaker.

Similarly, variable elimination requires both that it be possible to enumerate the values of an element and that it be possible to turn into a set of factors. To specify that it has the latter capability, you make it extend ProbFactorMaker and implement the makeFactors method. Factors are parameterized by the type of values they contain; in this case, since we are creating a factor representing probabilities, we make a Factor[Double].

For example, the AtomicBinomial class extends ProbFactorMaker and includes the following code:

def makeFactors: List[Factor[Double]] = {

val binVar = Variable(this)

val factor = new Factor[Double](Array(binVar))

for { (value, index) <- binVar.range.zipWithIndex } {

factor.set(Array(index), density(value))

}

List(factor)

}

The makeFactors method returns a list of factors. A factor is a table defined over a set of *variables*. To create a variable out of an element, use Variable. For example, Variable(this) creates a variable out of this atomic binomial element. Creating variables is memoized, so you can be sure that every time you call Variable on an element you get the same variable. This is important if an element participates in multiple factors. To create a factor, you pass it an array of its variables.

Each row in a factor associates a value with a set of indices into the variable’s ranges. To specify the factor, you need to set these values. This is accomplished with the set method of Factor. In the above example, we have

for { (value, index) <- binVar.range.zipWithIndex } {

factor.set(Array(index), density(value))

}

The first line uses a for comprehension to get at pairs of values of the binomial variable together with their index into the range. The standard Scala library method zipWithIndex takes a list and associates each element of the list with its index in the list. For example, List(“a”,”b”).zipWithIndex is List((“a”,0), (“b”,1)). The first argument to factor.set is an array of indices into the ranges of the variables, in the same order as the array used to create the factor. The second argument is the value to associate with those indices.

At the end, makeFactors returns a list consisting of this single factor. This is the basic principle behind creating factors. You can find a variety of more complex examples, including some with multiple variables, in Factor.scala. For atomic elements, the process should usually be similarly simple to that for binomials.

## Creating a class with special Metropolis-Hastings behavior

By default, proposing an element in Metropolis-Hastings uses the class’s standard generateRandomness to propose the new randomness. Earlier, we described how it is sometimes useful to create a special proposal distribution and gave SwitchingFlip as an example. SwitchingFlip is just like an ordinary Flip except that each time it is proposed, it switches to the opposite value.

Creating a different proposal distribution is achieved through the nextRandomness method. The nextRandomness method returns the new randomness, as well as the proposal probability ratio, which is defined by



where *r*0 is the original randomness, *r*1 is the proposed randomness, *P*(*r*1) is the probability of generateRandomness returning *r*1, and  indicates the probability of nextRandomess returning *r*1 when its argument is *r*0. By default, the nextRandomness method simply uses the element’s generateRandomness method and returns 1.0 for the proposal probability ratio. This is correct in most cases, and is used for most of the built-in elements. However, it can be overridden if desired. For example, the definition of SwitchingFlip includes

override def nextRandomness(rand: Randomness) =

if (rand < probValue)

(uniform(probValue, 1.0), (1.0 - probValue) / probValue)

else (uniform(0.0, probValue), probValue / (1.0 - probValue))

private def uniform(lower: Double, upper: Double) =

random.nextDouble \* (upper - lower) + lower

Everything else is inherited from Flip. The randomness of Flip is a double uniformly distributed between 0 and 1. The generateValue method of Flip tests whether this random number is less than the probability of a true outcome, which is contained in the probValue field. So, SwitchingFlip’s nextRandomness method first checks if the randomness is less than this value, which would imply that the current value is true. If it is, the new randomness is uniformly chosen between probValue and 1, which would make the next value false. On the otherhand, if the randomness is greater than probValue, the new randomness is chosen uniformly between 0 and probValue, which would make the next value true. Now, the correct randomness model is uniformly distributed, but this proposal distribution does not define an equal density in the two directions. In particular, for going from true to false, the density of the new randomness is 1 / (1 – probValue), while in the other direction it is 1 / probValue. So we must compensate by returning (1 – probValue) / probValue in the first case, and the inverse ratio in the second case.

# Creating a new algorithm

In addition to creating new element classes, Figaro provides support for creating new algorithms and integrating them into the existing library. Support is provided for query answering algorithms (like Metropolis-Hastings and variable elimination), probability of evidence algorithms, and defining new kinds of algorithms. Support is also provided both for anytime and one-time algorithms. We start this section by describing how to create a new one-time query-answering algorithm. We then discuss creating an anytime version of the algorithm, paying attention to sharing code between the one-time and anytime versions. We then describe how to create a probability of evidence algorithm, how to define an algorithm to be extensible to new classes, and how to define a new category of algorithm.

A good way to learn about creating algorithms, after reading this section, is to examine the Figaro code in com.cra.figaro.algorithm and its subpackages. If you do develop a new algorithm, please consider sharing it.

## One-time query answering algorithm

One-time query answering algorithms inherit from the trait OneTimeProbQuery. To implement such an algorithm, you need to provide implementations for:

* A constructor that allows the universe on which to operate and the set of query elements to be specified.
* run(), which runs the algorithm, putting it in a state where it can answer queries. For example, for a sampling algorithm, it collects and stores the required number of samples. For variable elimination, it eliminates all variables except the query variables.
* computeDistribution(element), which returns a distribution over values of the element. The element must be one of the query elements specified when the algorithm is created. The distribution is represented as a stream of probabilities paired with values. A stream is a lazy data structure that is potentially infinite. Streams are used for the return values of distributions to allow for algorithms that can return distributions with a non-zero probability of an infinite number of elements, although there are no such algorithms currently.
* computeExpectation(element, function), which computes the expectation of the element under the given function that maps a value of the element to a double.
* Optionally, computeProbability(element, predicate), which computes the probability that the element satisfies the given predicate that maps a value of the element to a Boolean.

#### Sampling

Extra support is provided for sampling algorithms in the form of UnweightedSampler and WeightedSampler classes. These take care of everything for you except for the process of producing a single sample. All you have to do for an unweighted sampler is extend UnweightedSampler and write a sample method that returns an instance of the Sample type, which stores the values of elements. The Sample type is defined to be Map[Element[\_], Any]. The \_ in place of the type parameter of Element indicates that the type parameter is unspecified, so any element can appear here. The element is mapped to an instance of Any which is the common supertype of all Scala types. So any element can be mapped to any value. To get a value out of a sample.

A useful thing to know is the clearTemporaries method of a universe. When a non-caching chain is expanded, a new element is created that should be reclaimed once the sample has finished being produced. universe.clearTemporaries deactivates all such elements, releasing their memory. This is necessary to avoid memory leaks during sampling.

#### Expansion and factors

A useful operation is to expand all chains in a universe to obtain the complete set of elements in the universe. This is achieved using the syntax

Expand(universe).expand()

As usual, the universe argument can be omitted, using the current default universe.

Support is provided for algorithms that are based on factors. Variable elimination is one example, but there are many other such algorithms. To create all the factors for an element, use

ProbFactor.make(element)

The standard procedure to turn a universe into a list of factors is to

1. Expand the universe.
2. Call universe.activeElements to get all the elements in the universe.
3. Make the factors for every element and collect them

## Anytime algorithms

An anytime algorithm proceeds in a series of steps. The algorithm can be interrupted after any step. For a sampling algorithm, a natural step is taking a single sample. The algorithm blocks while running a step, only answering queries when the step has terminated.

To create an anytime algorithm, in addition to the query answering methods like computeDistribution, you need to define the following:

* initialize(), which is called immediately when the algorithm starts
* runStep(), which is called repeatedly to run a single step. Answering queries should be a valid operation after any step.

#### Code sharing

Some algorithms, such as Figaro’s built-in sampling algorithms, might come in both anytime and one-time versions. It is desirable to share as much code as possible between these versions. In addition, different algorithms might share the same underlying code. For example, Metropolis-Hastings and importance sampling are both sampling algorithms, but they are somewhat different because the first uses unweighted samples while the second uses weighted samples. Two different unweighted sampling algorithms will want to share even more code. Figaro uses Scala’s abstract classes traits to help achieve code sharing.

A word on abstract classes versus traits. Neither can be instantiated. The main differences are that classes can take arguments, while traits support multiple inheritance. An inherited class must always be the first thing from which a subclass inherits, while traits can appear subsequently in the inheritance list.

All algorithms that compute conditional probabilities inherit from ProbQueryAlgorithm, from which OneTimeProbQuery and AnytimeProbQuery inherit. Algorithms that implement both versions can contain their core functionality in a class and provide a subclass or a constructor that inherits from one or the other of these traits, providing the specific methods for anytime or one-time algorithms.

For sampling algorithms, AnytimeSampler and OneTimeSampler are provided. These take care of the mechanics of running the sampler repeatedly. In particular, the AnytimeSampler implements the initialize and runStep methods so all you have to write is sample. These traits have the subtraits AnytimeProbQuerySampler and OneTimeProbQuerySampler that specifically capture sampling algorithms that compute the conditional probability of queries. In addition, Figaro provides UnweightedSampler and WeightedSampler that handle the mechanics of Sample data types, initializing sample sets, accumulating samples, and answering queries involving samples.

Using all these traits and classes, anytime and one-time importance sampling can be defined easily. First we create an Importance class, as follows:

abstract class Importance(universe: Universe, targets: Element[\_]\*)

extends WeightedSampler(universe, targets:\_\*) {

// implementation of sample() goes here

}

It takes the universe to operate on as its first argument and a comma-separated sequence of target query elements as its second. It is specified to be a weighted sampler using the same universe and targets. The body of the class implements the sample method. Note that this class is abstract and cannot be instantiated. We provide a companion Importance object that provides two factory constructors, one for anytime and one for one-time importance sampling:

object Importance {

def apply(targets: Element[\_]\*)(implicit universe: Universe) =

new Importance(universe, targets:\_\*) with AnytimeProbQuerySampler

def apply(myNumSamples: Int, targets: Element[\_]\*)(implicit universe: Universe) =

new Importance(universe, targets:\_\*) with

OneTimeProbQuerySampler { val numSamples = myNumSamples }

}

The first constructor takes has two argument lists. The first is a comma-separated sequence of query targets, and the second provides the universe. Since it implicit, it can be omitted and the default universe is used. Since the number of samples is not explicitly provided, it is assumed that the anytime version is wanted, so the constructor inherits from AnytimeProbQuerySampler. In the second, case, the number of samples is specified, so it inherits from OneTimeProbQuerySampler. One detail to note is that OneTimeProbQuerySampler contains an abstract field named numSamples that must be defined to create an instance of the trait. This is accomplished through the code

OneTimeProbQuerySampler { val numSamples = myNumSamples }

This creates an anonymous subclass of OneTimeProbQuerySampler in which the numSamples field is defined to be the value passed into the constructor.

## Probability of evidence algorithm

Creating an algorithm that produces probability of evidence is similar to what we have seen for algorithms that compute probability of query elements. You need to define a computeProbEvidence() method that returns the probability of evidence. For one-time algorithms, you also need to inherit from OneTimeProbEvidence and provide a run() method. For anytime algorithms, you need to inherit from AnytimeProbEvidence and provide initialize(), and runStep() methods.

For sampling algorithms, the run method can be inherited from OneTimeSampler and the initialize and runStep methods can be inherited from AnytimeSampler. As a result, all our sampling-based probability of evidence algorithm needs to implement is the doSample method. The algorithm uses as a subroutine the Forward() function which runs a forward sampling pass to generate values for elements, ignoring conditions and constraints.

## Allowing extension to new element classes

We saw in the section on making a class usable by variable elimination how to make a new element class work under an existing algorithm without modifying the algorithm’s code. To allow this, the algorithm must be defined to support extension in this way. We illustrate how to do this using range computation. The computation uses at its heart a subroutine a function called concreteValues whose definition is as follows:

private def concreteValues[T](element: Element[T]): Set[T] =

element match {

case c: Constant[\_] => Set(c.constant)

case f: Flip => Set(true, false)

...

case v: ValuesMaker[\_] => v.makeValues.toSet

case \_ => throw new UnsupportedAlgorithmException(element)

}

This function takes an element and tests to see what kind of element it is. If it is a constant, the values is a singleton set containing the constant; if it is a flip, it is a set containing true and false, and so on. If the value fails to match any of the built-in types for which this function is defined, it arrives at the second to last case. This tests if the value is an instance of ValuesMaker. If it is, the values makeValues method is used. The final case is a catchall – the notation \_ represents a pattern that catches all values. If the value has arrived at this case, its values cannot be computed, so we throw an UnsupportedAlgorithmException.

## Creating a new category of algorithm

Suppose you want to create a new category of algorithm, for example one that computes the most probable values for the elements in the universe. We will illustrate how this is done for probability of evidence algorithms, and the same pattern can be used elsewhere. All algorithms extend the Algorithm trait, which defines the general interface to algorithms using start, stop, resume, and kill. For example,

abstract class ProbEvidenceAlgorithm(val universe: Universe)

extends Algorithm {

/\* Particular implementations of probability of evidence algorithms must define the following method. \*/

protected def computeProbEvidence(): Double

/\*

\* The following method is defined in either the onetime or anytime versions of this class, and does not need to

\* be defined by particular algorithm implementations.

\*/

def probabilityOfEvidence(): Double

}

It declares two abstract methods. probabilityOfEvidence is the interface for the user to query the probability of evidence. computeProbEvidence does the actual work and needs to be provided by a particular implementation.

Next, we provide one-time and anytime traits for probability of evidence algorithms. The one-time trait is very easy:

trait OneTimeProbEvidence extends ProbEvidenceAlgorithm with OneTime {

def probabilityOfEvidence(): Double = computeProbEvidence()

}

Because of the functionality of OneTime, the start, stop, resume, and kill methods are automatically defined, and the algorithm creator need only provide run (declared in OneTime) and computeProbEvidence. For the anytime version, we need to do more work. In the case of probability of evidence, the trait is defined by

trait AnytimeProbEvidence extends ProbEvidenceAlgorithm with Anytime {

case object ComputeProbEvidence extends Service

case class ProbEvidence(probability: Double) extends Response

protected def handle(service: Service): Response =

service match {

case ComputeProbEvidence =>

ProbEvidence(computeProbEvidence())

}

def probabilityOfEvidence(): Double = {

runner ! Handle(ComputeProbEvidence)

receive {

case ProbEvidence(result) => result

}

}

}

Anytime algorithms run in a separate thread, and we need to be able to communicate with the thread to get the probability of evidence out of it. This is accomplished using Scala’s *actors* framework. Actors communicate by sending and processing messages. The Anytime trait defines the runner field, which is the actor that runs the algorithm. The probabilityOfEvidence calculation sends a message to the runner telling it to compute the probability of evidence, listens for the result, and returns it. The syntax for sending the message to the runner is

runner ! Handle(ComputeProbEvidence)

which sends a message whose content is Handle(ComputeProbEvidence). runner dispatches this message to a method called handle (which is abstract in Anytime and defined in AnytimeProbEvidence. This method knows how to handle ComputeProbEvidence – it calls the method computeProbEvidence, which is abstract in ProbEvidenceAlgorithm and must be provided by an implementation. It turns the resulting probability p into a message ProbEvidence(p), which is sent back to the algorithm from the runner. The algorithm (in probabilityOfEvidence) then extracts the result and returns it.

To summarize, to define an anytime version of the algorithm, you need to do the following:

1. Create a case class or object to represent the services provided by your algorithm. Here, it is accomplished by

case object ComputeProbEvidence extends Service

1. Create a case class or object to represent the responses provided by these services. Here,

case class ProbEvidence(probability: Double)

extends Response

1. Create a handler in the method handle that takes a Service, performs some computation, and returns a response.
2. In each method that provides an interface to querying the algorithm
   1. Send a message to the runner asking for the appropriate service
   2. Receive a message from the runner, extract the result, and return it.

# Conclusion

As you can see, there’s quite a lot to Figaro. I hope you will find something useful here and will not find it too difficult to use. If you have any comments, suggestions, bug fixes, etc., please send them to me at [apfeffer@cra..com](mailto:apfeffer@cra..com).

Some of the planned next steps for Figaro are:

* Learning models from data
* Approximate factor-based algorithms like belief propagation
* Lazy algorithms for reasoning about potentially infinite domains
* Better dynamic reasoning algorithms

If you have any suggestions to make along these lines or about other possible next steps, I would love to hear from you. If you want to make a contribution, I would be delighted.

Thanks for reading, and enjoy!