Figaro Tutorial

Avi Pfeffer, Brian Ruttenberg, and Michael Howard, Charles River Analytics

Contents

Introduction

What is Figaro?

Reasoning under uncertainty requires taking what you know and inferring what you don't know, when what you know doesn't tell you for sure what you don't know. A well established approach for reasoning under uncertainty is probabilistic reasoning. Typically, you create a probabilistic model over all the variables you're interested in, observe the values of some variables, and query others. There is a huge variety of probabilistic models, and new ones are being developed constantly. Figaro is designed to help build and reason with the wide range of probabilistic models.

Developing a new probabilistic model normally requires developing a representation for the model and a reasoning algorithm that can draw useful conclusions from evidence, and in many cases also an algorithm to learn aspects of the model from data. These can be challenging tasks, making probabilistic reasoning require significant effort and expertise. Furthermore, most probabilistic reasoning tools are standalone and difficult to integrate into larger programs.

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
- models with structured decisions
- models with unknown parameters

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 or simulated annealing
- Probability of evidence using importance sampling
- Particle Filtering
- Parameter learning using expectation maximization

Figaro provides both regular (the algorithm is run once) and anytime (the algorithm is run until stopped) 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 is free and is released under an open-source license (see license file). The public code repository for Figaro can also be found at https://github.com/p2t2

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. This means that any function, data structure or operation in Scala or Java be incorporated into a Figaro model, giving the user many powerful tools for building probabilistic models.

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. "Programming in Scala" by Martin Odersky is available for free online.

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, including a brief discussion of probabilistic programming for dynamic models, decision networks, parameter learning and hierarchical reasoning. 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. All of the code for the examples presented in this tutorial can be found with the set of examples distributed with Figaro.

Hello world!

Make sure Scala version 2.10.0 or later is installed on your machine. Follow the instructions to either extract the figaro.jar to some location or build the jar from the code repository. 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.language._
```

This loads the portion of the Figaro package 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 com.cra.figaro.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
```

While this scenario is quite trivial, this example outlines the typical process involved with using probabilistic models in Figaro: Build the model, run an inference algorithm, and query for a result.

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 data structures that are part of a Figaro model 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)
```

Figaro classes are capitalized, while Scala reserved words are not

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]`.

Scala uses type inference, so the value type of the parameter can often be omitted at class creation (the compiler will determine the type)

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`.

Every Figaro `Element[U]` has a *value*, which represents the current value of the element and is of type `U`. For `Constant` elements, the value of the element never changes. However, for stochastic elements, the value of the element may change depending on the usage of the model, as explained in the next section.

Atomic elements

An *atomic element* is one that does not depend on any other 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]`.

- The continuous `Uniform(0.0, 2.0)` is an `Element[Double]` that represents the continuous uniform probability distribution between 0 and 2.

Elements also contain a method for generating a new value for the element. This is accomplished by called the `generate()` method of an element. Generating a new value of an element is a two step process. First, the `generate()` method generates a new *randomness* for the element. Then, the value of an atomic element is *deterministically* produced as a function of the randomness of the element. For example, consider the `Flip(0.7)` shown above. To determine the value of this `Flip`, a number between 0 and 1 is uniformly chosen as the randomness of the element, and if the value is less than 0.7, we assign the `Flip` a value of `true`, and `false` otherwise.

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._
```

The _ is the Scala version of Java's * for imports

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. Most of 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))
```

Note `If` is a Figaro class, not the Scala `if` reserved word

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. Similar to the atomic elements, the value of a compound element can be generated by calling the `generate()` method of the element. In this case, however, the value of an element is a deterministic function of its randomness and the current value of the element's parents.

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` is 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 parent 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 randomly produced from the generated `Element[U]`. Therefore, a `Chain[T,U]` is an `Element[U]`.

Scala notation for the type of a function is: ***inType => outType***

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 (all Figaro classes are capitalized). Now,

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

Anonymous functions in Scala are created by defining an argument list and the body of the function. The return type is inferred by the compiler

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. Note that there is a version of `Chain` that utilizes two parents and requires a function from a tuple of the parent types to the output type. If more parents are required for a `Chain`, multiple `Chains` can be nested together.

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,

Figaro `Apply` is a class, different than the Scala apply which is a method defined on many classes

```
(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.

Sequences in Scala are similar to Java. `Seq` is the superclass in Scala for many types of data structures, such as List.

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]]`. 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

The notation: _* will turn any Scala sequence into a variable argument list. The function receiving the variable argument list will have a * on the last argument

```
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)):_*)
```

which has the same effect as listing the individual entries of the List in the argument list.

Calling `map` on a Scala sequence will apply the same function to every value in the sequence and return a new sequence. It uses anonymous function semantics

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 (6, 7) with probability 0.2 and (6, 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 assign it to a Scala value:

```
val burglary = Flip(0.01)
```

A `val` represents a field (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 (Note that the scala assignment of the field `burglary` cannot change, but the value of the Figaro element that is assigned to it, `Flip(0.01)`, *can* change). You can use the value of a field by referring to its name:

`val` in Figaro represents an immutable value. When a thing is assigned to a val, data inside the thing can change but the reference stored in the val is constant

```
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 (each call produces a new `Flip`), 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:

This example is found in Burglary.scala

```
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),
```

Scala statements can be written on multiple lines

```
                    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.language._
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. All possible values of the parent still need to be accounted for in the argument list using a combination of `One of`, `NoneOf` and `*`.

Conditions and constraints

So far, we have described models that generate the values of elements. It is also possible to influence the values of elements by imposing conditions or constraints on them.

A *condition* represents something the value of the element must satisfy. 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`. Conditions can have multiple purposes. One is to assert evidence, by specifying something that is known about an element. Alternatively, a condition can specify a structural property of a model, for example, that only one of two teams playing a game can be the winner.

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)
```

Note that using `observe` will remove all previous conditions on an element.

A *constraint* provides a way to specify a potential or weighting 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.

This example is found in Firms.scala

```
import com.cra.figaro.language._
import com.cra.figaro.library.atomic._
import com.cra.figaro.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 has 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` (and separate Figaro elements in each class). 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.

This example is found in Smokers.scala

```
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)
}
```

Single line function definitions in Scala do not need bracketing

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))
```

Abstract classes in Scala are similar as in Java; they cannot be instantiated

Defining class contents at instantiation time will override undefined values

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 (the from Source field has not been assigned an `Element[Source]` yet). 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 in line 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 exact observations on continuous elements can be problematic for many types of inference algorithms.

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:

```
import com.cra.figaro.library.compound.CPD
import com.cra.figaro.language._

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)
```

This example can be found in SimpleMovie.scala

The ‘ in front of a string creates a Scala symbol, which are treated like `String` constants

The `_.award` notation is Scala shorthand to retrieve the award value of each element of the map

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. Finally, we impose the `uniqueAwardCondition` on `allAwards`, ensuring that exactly one appearance is awarded.

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:

```
import com.cra.figaro.library.compound.CPD
import com.cra.figaro.language._

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
```

::= is Scala shorthand for list concatenation

```
  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 current `actor.movies` list, and replaces the current list with the new 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, we have found it convenient to use a hash table that maps concepts to their associated elements. This allowed us to create the element associated with a concept as the concept was introduced. If we later had to refer to the same concept again, we 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._
import com.cra.figaro.language.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()`. When a compound element is created that uses a parent element, the parent must already be active.

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. See the documentation in Universe.scala for more details.

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:

This example can be found in CarAndEngine.scala

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

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 `Car` being created, which is referred to by `this` within the `Car` class. We similarly make the abstract `Engine` class inherit element collections and assign `power` the name "power" within `I4`, `V8`, and `MyEngine` within each subclass of `Engine`.

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:

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

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:

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

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")
}
```

The body of the sum function is shorthand notation for Scala's fold function. Fold iterates through a sequence and applies a function to the previous result and each new entry in turn. The (_ + _) notation will add the previous value to each value in `xs`.

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 define 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:

This example can be found in OpenUniverse.scala

```
import com.cra.figaro.language._
import com.cra.figaro.library.atomic.continuous.Uniform
import com.cra.figaro.library.atomic.continuous.Normal
import com.cra.figaro.library.atomic.discrete.Geometric
import com.cra.figaro.library.compound.{MakeList, IntSelector}

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. `MakeList` could also have been used in the previous section's example to define the `components` element of the `Container` class.

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, we describe an algorithm that simply computes the range of possible values of all elements in a universe. Then, we describe three algorithms for computing the conditional probability of query elements given evidence (conditions and constraints) on elements. These are variable elimination, importance sampling, and Markov chain Monte Carlo. Next, we describe algorithms for performing other kinds of reasoning. One is an importance sampling algorithm for computing the probability of evidence in a universe. We also discuss a variable elimination algorithm and a simulated annealing algorithm for computing the most likely values of elements given the evidence. Finally, we 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 elements in a universe, as long as expanding the probabilistic model of the universe does not (1) result in generating an infinite number of elements; (2) result in an infinite number of values for an element; or (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 memoized (cached), meaning that the range is only computed once for each object and then stored for future use.

Asserting evidence

Most Figaro reasoning involves drawing conclusions from evidence. Evidence in Figaro is specified in one of two ways. The first is through conditions and constraints, as we described earlier. The second is by providing *named evidence*, in which the evidence is associated with an element with a particular name or reference.

There are a variety of situations where using named evidence is beneficial. One might have a situation where the actual element referred to by a reference is uncertain, so we can't directly specify a condition or constraint on the element, but by associating the evidence with the reference, we can ensure that it is applied correctly. Names also allow us to keep track of and apply evidence to elements that correspond to the same object in different universes, as will be seen below with dynamic reasoning. Finally, associating evidence with names and references allows us to keep the evidence separate from the definition of the probabilistic model, which is not achieved by conditions and constraints.

Named evidence is specified by

`NamedEvidence(reference, evidence)`

where `reference` is a reference, and `evidence` is an instance of the `Evidence` class. There are three concrete subclasses of `Evidence`: `Condition`, `Constraint`, and `Observation`, which behave like an element's `setCondition`, `setConstraint`, and `observe` methods respectively.

For example,

`NamedEvidence("car.size", Condition((s: Symbol) => s != 'small)))`

represents the evidence that the element referred to by `"car.size"` does not have value `'small`.

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 of the universe 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.language._
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, predicate)` will return the probability that the value of `e2` satisfies the given predicate. For example, `(b: Boolean) => b` is the function that takes a Boolean argument and returns true precisely if its argument is true. So, `ve.probability(e2, (b: Boolean) => b)` computes the probability that `e2` has value true. The `probability` method also provides a shorthand version that specifies 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`. If you just want the expectation of the element, you just provide a function that returns the value of the function.

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. Note that only elements provided in the argument list of the `VariableElimination` class can be queried; if at a later point you want to query a different element not in the argument list, you must create a new instance of `VariableElimination`

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

Figaro's importance sampling algorithm 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.language._
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(10000, e2)
```

The first argument to `Importance` is an indication of how many samples the algorithm should take. The second argument (and subsequent arguments) lists the element(s) that will be queried. After calling `imp.start()`, you can use the methods `distribution`, `probability`, and `expectation` to answer queries.

The importance sampling algorithm used above is an example of a "one-time" algorithm. That is, the algorithm is run for 10,000 iterations and terminates; it cannot be used again. Figaro also provides an "anytime" importance sampling algorithm that runs in a separate thread and continues to accumulate samples until it is stopped. A major benefit of an anytime algorithm is that it can be queried while it is running. Another benefit is that you can tell it how long you want it to run.

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.language._
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.

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.

This example can be found in SimpleMovie.scala

```
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. Since `award` is now defined as a `SwitchingFlip`, each `award` will switch

value so there will still be only one award awarded. In general, a `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.

Chains and Metropolis-Hastings

In designing a Metropolis-Hastings algorithm using chains, there are design considerations of the model that can affect the run-time and memory performance of the algorithm. Chains contain an internal cache of previously generated elements from different combinations of its argument values. When a chain's function is invoked on an argument to produce a result element, the cache is first checked to determine if there exists an entry for the argument value. If an entry does exist, the cached element is retrieved and used to determine the value of the chain. If no entry exists, then the chain's function is invoked, an element is returned from the function and placed in the cache. The cache also contains a maximum capacity; once the capacity of the cache is reached, a random element is selected in the cache and discarded. The capacity of the cache can significantly impact the performance of the Metropolis-Hastings algorithm

The standard advantage of a large cache capacity is that it can save significant time if the function is executed repeatedly on a finite set of argument values. 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. Large capacity caches increases the chance that this work is saved and reused every time the parent of the chain returns to the same value. With a small capacity cache, elements can be evicted if there are many different parent values. If at a later stage the parent returns to the same original value, it may have been evicted from the cache and we'll need to begin the search process from scratch.

However, the standard disadvantage of large caches is that they use more memory. In particular, a different element is stored for every value of the parent that has been seen, and may never be released if the cache is large enough. If the parent can have a large or infinite number of possible values, this can lead to exhausting the memory of the machine.

Fortunately, most of the cache management is automatically handled internally by Figaro. There are two types of chains defined in Figaro: `CachingChain` and `NonCachingChain`. `CachingChain` by default instantiates a chain with a cache capacity of 1000, whereas a `NonCachingChain` instantiates a chain with a capacity of 1. In general, a `CachingChain` is usually better for elements with discrete parents with relatively few values, and a `NonCachingChain` is better for elements with continuous parents. When a user creates a new `Chain` class, Figaro attempts to determine the best chain to use given the parents of the chain. In most cases, the cache capacity selected by Figaro will be adequate to use the model efficiently in a Metropolis-Hastings algorithm. However, should you need to ensure the efficiency of the model in a Metropolis-Hastings algorithm, the user can still explicitly instantiate a `Chain` class with specific cache capacity.

Debugging Metropolis-Hastings

Designing good proposal schemes is more of an art than a science and can be quite challenging. Finding a good proposal scheme for the movies example was quite time consuming. It also required 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 may be useful to users.

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. Since there is the potential for ambiguity here, Figaro is careful to define what constitutes evidence for computing probability of evidence. Conditions and constraints often constitute evidence. Sometimes, however, they can be considered to be part of the model specification. Consider, for example, the constraint on pairs of friends that they share the same smoking habits—this is part of the model definition, not evidence.

For this reason, Figaro allows the probability of evidence to be computed in steps. To compute the probability of conditions and constraints that are in the Figaro program, you can use

```
import com.cra.figaro.language._
import com.cra.figaro.algorithm.sampling.ProbEvidenceSampler

val alg = new ProbEvidenceSampler(universe) with
  OneTimeProbEvidenceSampler { val numSamples = n }
alg.start()
```

where `n` is an integer indicating number of samples for one-time sampling. To retrieve the probability of the evidence, you simply call `alg.probEvidence`.

If you want to compute the probability of additional evidence, in addition to the conditions and constraints in the program, you can pass this additional evidence as the second argument to `new ProbEvidenceSampler`. This argument takes the form of a list of `NamedEvidence` items, where each item specifies a reference and evidence to apply to the element pointed to by the reference. For example, you could supply the following list as the second argument to `ProbEvidenceSampler.`

```
List(NamedEvidence("f", Observation(true)),
  NamedEvidence("u", Observation(0.7)))
```

`ProbEvidenceSampler` will then compute the probability of all the evidence, both the named evidence and the existing evidence in the program. It does this by temporarily asserting the named evidence, running the probability of evidence computation, and then retracting the named evidence.

If you don't want to include the existing conditions and constraints in the program in the probability of evidence calculation, there are four ways to proceed. Each method is more verbose than the previous but provides more control. The simplest is to use

```
ProbEvidenceSampler.computeProbEvidence(n, namedEvidence)
```

This takes care of running the necessary algorithms and returns the probability of the named evidence, treating the existing conditions and constraints as part of the program definition. You can also use the following

```
val alg = ProbEvidenceSampler(n, namedEvidence)
alg.start()
```

This method enables you to control when to run alg, and also to reuse alg for different purposes. The final two methods explicitly compute probability of the conditions and constraints in the program, which becomes the denominator for subsequent probability of evidence computations. The `ProbEvidenceSampler` class provides a method called `probAdditionalEvidence` that creates a new algorithm that uses the probability of evidence of the current algorithm as denominator. You could proceed as follows:

```
val alg1 = new ProbEvidenceSampler(universe) with
  OneTimeProbEvidenceSampler { val numSamples = n }
alg1.start()
val alg2 = alg1.probAdditionalEvidence(namedEvidence)
alg2.start()
```

The major advantage of this method is that you can call `alg1.probAdditionalEvidence` multiple times with different named evidence without having to repeat the denominator computation. The final method, which provides maximum control, is:

```
val alg1 = new ProbEvidenceSampler(universe) with
  OneTimeProbEvidenceSampler { val numSamples = n1 }
alg1.start()
val alg2 = new ProbEvidenceSampler(universe) with
  OneTimeProbEvidenceSampler { val numSamples = n2 }
alg2.start()
```

In this example, a different number of samples is used for the initial denominator calculation and the subsequent probability of evidence calculation.

Currently, only forward sampling probability of evidence is provided, but the framework exists to create additional probability of evidence algorithms. There is also an anytime version of the probability of evidence algorithms. To create one, use

```
new ProbEvidenceSampler(universe) with AnytimeProbEvidenceSampler
```

For the methods that require you to specify the number of samples `n`, replace `n` with `t`, where `t` is a long value indicating the number of milliseconds to wait while computing the denominator (and also while computing the probability of the named evidence for the `computeProbEvidence` shorthand method.)

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. There are two ways to compute MPE: Variable Elimination and Simulated Annealing. An example that shows how to compute the MPE using Variable Elimination is:

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

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
```

Computing the most likely value of an element can also be accomplished using simulated annealing, which is based on the Metropolis-Hastings algorithm. The main idea behind simulated annealing is to sample the space of the model and make transitions to higher probability states of the model. Over many

iterations, the algorithm slowly makes it less likely that the sampler will transition to a lower probability state than the one it is already in, with the intent of slowly moving the model towards the global maximum probability state.

Central to this idea is the cooling schedule of the algorithm; this determines how fast the model converges toward the most likely state. A faster schedule means the algorithm will quickly converge upon a high probability state, but since it allows for little exploration of the model space the risk that algorithm gets stuck in a local maxima is high. Conversely, a slow schedule allows for a more thorough exploration of the model space but can take long to converge.

In Figaro, the Metropolis-Hastings based simulated annealing is instantiated very similarly to the normal MH algorithm. Consider an example of using simulated annealing on the smokers model presented earlier:

This example can be found in AnnealingSmokers.scala

```
import com.cra.figaro.language._
import com.cra.figaro.library.compound.^^
import com.cra.figaro.algorithm.sampling.ProposalScheme
import com.cra.figaro.algorithm.sampling.MetropolisHastingsAnnealer
import com.cra.figaro.algorithm.sampling.Schedule

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)
}

val mhAnnealer = MetropolisHastingsAnnealer(ProposalScheme.default,
  Schedule.default(2.0))
```

The second argument is an instance of a Schedule class (similar to a ProposalScheme), and contains the method that slowly moves the sampler towards a more likely state. It is defined as:

```
class Schedule(sch: (Double, Int) => Double) {
   def temperature(current: Double, iter: Int) = sch(current, iter)
}
```

This class takes in a function from a tuple of `(Double, Int)` to a `Double`. At each iteration (after any burn-in), the simulated annealing will call `schedule.temperature` with the current transition probability and iteration count. The schedule will then return a new transition probability that will be used to accept or reject the new sampler state. The default schedule is defined as:

```
def default(k: Double = 1.0) = new Schedule((c: Double, i: Int) =>
  math.log(i.toDouble+1.0)/k)
```

To run simulated annealing, one simply calls `run()` as in a normal Metropolis-Hastings algorithm. Once the algorithm has completed, one can retrieve the most likely value of an element by calling `mhAnnealer.mostLikelyValue(element)`. Note that when the algorithm finds the most probable state of the model, it records the values for each active element. Therefore, queries on the most likely values of temporary elements that are *not* part of the optimal state of the model may fail.

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:

This example can be found in Sources.scala

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

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 (`upperTriangle` is in the Figaro package) . It then sees if the two samples have a possible source in common. If they do, it imposes a condition 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)
}
```

Observe that each element created in the `Pair` class is added to the universe of the `Pair`, not the universe that contains `sample.fromSource`. 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:

```
import com.cra.figaro.language._
import com.cra.figaro.library.atomic.continuous.Uniform
import com.cra.figaro.library.compound.If
import com.cra.figaro.algorithm.{AbstractionScheme, Abstraction}
import com.cra.figaro.algorithm.factored._

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.

Reproducing inference results

Running inference on a model is generally a random process, and performing the same inference repeatedly on a model may produce slightly different results. This can sometimes make debugging difficult, as bugs may or may not be encountered, depending on the random values that were generated during inference. For that reason, Figaro has the ability to generate reproducible inference runs.

All elements in Figaro use the same random number generator to retrieve random values. This can be accessed by importing the `util` Figaro package and using the value `random`, which is Figaro's random number generator. For example, the `generateRandomness()` function in the Select element is:

```
import com.cra.figaro.util._

def generateRandomness() = random.nextDouble()
```

To reproduce the results of an inference algorithm, you must set the seed in the random number generator. Repeated runs of the same algorithm with the same seed will then be identical, making debugging much easier since errors can be tracked between runs. To set the seed, you import the util package, and simply call `setSeed(s: Long)`. To retrieve the current random number generator seed, one calls `getSeed()`.

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 initial = Universe.createNew()
val f = Flip(0.2)("f", initial)

def trans(previousUniverse: Universe): Universe = {
  val newU = Universe.createNew()
  val b = previousUniverse.get[Boolean]("f")
  val f = If(b, Flip(0.8), Flip(0.3))("f", newU)
  newU
}
```

The first 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. Note that we can give this new element the same name as the previous element since they are part of different universes. We get at this element using `previousUniverse.get[Boolean]("f")`. Essentially, when elements in different universes at different time points have the same name, they represent the state of the same variable at different points in time. 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(initial, trans, numParticles)
```

where `intial` is the initial universe, `trans` is the transition model (a function from `Universe => Universe`), 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. This problem is solved by the use of named evidence, so we can refer to the correct element without having a handle on the specific universe.

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.

There are a couple of implementation notes about particle filtering that a user should be aware of. First, since particle filtering estimates probabilities of elements using many particles, it can get expensive (memory wise) to store the state estimates for every element in the model. Therefore, the states of only *named* elements are tracked through time. This means that queries to filter for an element probability or expectation must be on named elements. In addition, because filtering tracks estimates through time, we want to free up memory from old universes that are no longer used. To accomplish this, when `advanceTime` is called, all named elements from the previous universe are copied as constants to a new, temporary universe. The temporary universe is then used in the transition function, allowing the real previous universe to be freed while still letting the new universe use the correct values from the old universe.

Decisions

Figaro also contains the ability to solve and query structured decision problems. These types of models, also known as influence diagrams, are generalizations of Bayesian networks that contain additional decision and utility variables. Figaro generalizes ordinary decision models by allowing the information on which a decision is based to be an arbitrary data structure. Also, the full power of the programming language is available to build decision models. However, Figaro does require that the possible values of the decision variables themselves be discretely enumerated.

In this section, we first give a very brief introduction into decision models and decision-making. We then provide a small example of decision-making in Figaro. We also delve deeper into the decision-making implementation in Figaro. Finally, we discuss the different ways that decision-making can be performed on single and multiple decision models.

Decision models

Decision models are generalizations of Bayesian networks that contain two additional variable types. The first is a decision variable, which represents a set of actions that a decision-maker can perform. The parents of a decision variable represent the information available to the decision-maker at the time of the decision. Decision models also contain utility variables, which represent some gain or loss in the model that directly or indirectly depends upon some previous decisions or random variables.

The purpose of a decision model is usually to compute an optimal policy for each decision in the model, where a policy defines what action a decision-maker should take for every possible value of the decision's parent variable(s). An optimal policy is when every action specified by the policy for each value of the parents is optimal with respect to some measure. To measure the optimality of an action, Figaro uses the maximum expected utility of the action. That is, for each value of a decision's parents, Figaro determines the action that will result in the highest expected utility of the model.

Basic example

Using Figaro's decision-making capabilities is generally quite simple. For example, consider the code for a simple decision model shown below:

This example can be found in SingleDecision.scala

```
import com.cra.figaro.language._
import com.cra.figaro.algorithm.decision._
import com.cra.figaro.library.compound._
import com.cra.figaro.library.decision._

val market = Select(0.5 -> 0, 0.3 -> 1, 0.2 -> 2)
val survey = CPD(market, 0 -> Select(0.6 -> 0, 0.3 -> 1, 0.1 -> 2),
                         1 -> Select(0.3 -> 0, 0.4 -> 1, 0.3 -> 2),
                         2 -> Select(0.1 -> 0, 0.4 -> 1, 0.5 -> 2))

val found = Decision(survey, List(true, false))

def valueFcn(f: Boolean, m: Int): Double = {
  if (f) {
    m match {
      case 0 => -7.0
      case 1 => 5.0
      case 2 => 20.0
    }
  } else {
    0.0
```

```
    }
  }

  val value = Apply(found, market, valueFcn)

  val alg = DecisionVariableElimination(List(value), found)
  alg.start()
  alg.setPolicy(found)
```

The first four lines import the packages needed for decision models. The elements `market` and `survey` are random variables in a normal Figaro model. We create a decision variable called `found` that uses the element `survey` as a parent, with the possible actions of the decision as `true` or `false`. The element named `value` is a utility variable that computes a `Double` conditioned upon the action of the decision (`found`) and the current value of the `market` element. It uses the function `valueFcn` to compute current utility. Finally, we use Figaro's decision variable elimination to compute an optimal policy for the `found` decision, and set the policy in the `found` element when the algorithm completes so that it can be used for querying.

Decisions in Figaro

As can be seen in the previous section, decision-making can be implemented in Figaro with little effort. Decisions are created using the `Decision[T,U]` element. The `Decision[T,U]` element actually inherits from `Chain`; that is, a decision is simply an element that uses an `Element[T]` as a parent, and generates an `Element[U]` as the action. A new decision is instantiated simply as:

```
Decision(Flip(0.7), List(0, 1, 2))
```

where the first argument is the parent of the decision, and the second argument is a list of the possible actions of the decision. The possible actions must always be finite and discrete. However, the parent of a decision may be an element over any Scala type. So, we could imagine making a decision based on a social network or a DNA sequence. One thing to note is that decision elements only support single parent decisions. However, multiple parent decisions can be easily created by grouping several parent elements into an element tuple. There are various other ways to instantiate a decision that can be found in the code for the `Decision` class.

Also, the no-forgetting assumption in decision models is not explicitly enforced in Figaro, hence Limited Memory Influence Diagrams (LIMIDs) can be represented in Figaro, though there is not an explicit LIMID reasoning algorithm implemented.

In decision models, there are also variables that represent the utility of the model. In Figaro, there is no need to explicitly create a utility element; this can be easily done using the `Apply` element, as shown in the example above. Utility elements must be of type `Element[Double]`.

A decision is similar to a chain, but unlike the chain, a decision element can change its functionality after an optimal policy has been computed for the decision. Most of the time, setting the policy of a decision can be done simply through the algorithm that computes the optimal policy. However, a user may manually set the policy of a decision element by calling the `setPolicy` function of the decision, defined as:

```
def setPolicy(new_fcn: (T => Element[U])): Unit
```

That is, setting the policy of a decision is just providing a new function from the value of a parent to an `Element[U]`. Users can also get the policy for a specific value of the parent by calling

`getPolicy(p: T): Element[U]`. Various other ways to set the policy can also be found in the Decision code.

Single decision models and policy generation

Single decision models can be created in Figaro by simply inserting a Decision element into the model. Once the model has been created, the goal is usually to compute the optimal policy for the decision that maximizes the expected utility of the model. This is done as two explicit steps in Figaro; computing the expected utility of each parent and decision pair, then determining the decision that has the maximum expected utility for each parent value. The policy is then set as a function that returns the maximum expected utility decision as a `Constant` for any parent value. This policy computation is performed using one of Figaro's built-in inference algorithms. Two alternative methods are provided. One is generally used when the support of the parent is finite, the other when it is infinite. However, there are some cases where the support is finite but very large and the infinite support method is preferable. Alternatively, for some distributions with infinite support, like Poisson or Geometric, only a small number of values are likely, and the finite support method can be used.

Infinite parent support

In this case, computing the optimal policy can be performed using the variable elimination, importance sampling, or Metropolis-Hastings algorithms. In addition to the normal parameters that each algorithm takes (as explained in previous sections), the decision version of these algorithms also takes a `List[Element[Double]]` that indicates the utility nodes in the model. The target of the algorithm is always the decision you wish to compute an optimal policy for. To find the optimal policy for discrete decisions, you simply instantiate one of the algorithms, for example:

```
val alg = DecisionVariableElimination(List(value), found)
val alg = DecisionImportanceSampling(10000,  List(value), found)
val alg = DecisionMetropolisHastings(10000, ProposalScheme.default,
  1000, List(value), found)
```

Where `List(value)` is the list of utilities in the model, and `found` is the decision. To compute the optimal policy, you simple start the algorithm, i.e., `alg.start()`. One the algorithm has completed running, you can call `alg.setPolicy(found)`, which will set the optimal policy in the Decision element that was computed from the algorithm.

Infinite parent support

When the parent(s) of a decision have infinite support, it is more difficult to compute an exact optimal policy. This is because it is not possible to compute the maximum expected utility action for each value of the parent since the range of the parent is infinite. In such a case, we use Figaro's sampling algorithms to compute an *approximate* optimal policy that attempts to provide a maximal expected utility decision for any possible value of the parent. Since we use sampling algorithms to compute the approximation, only importance sampling and Metropolis-Hastings can be used with continuous decisions.

Instantiating a decision with infinite parent support is similar to finite parent support, except that one must explicitly instantiate a `NonCachingDecision`, which is based on `NonCachingChain`:

```
NonCachingDecision(Normal(0.0, 1.0), List(0, 1, 2))
```

The creation of the algorithm and setting of the policy is the same as discrete decisions. Internally in Figaro, however, there are major differences between the implementation of policies for discrete and continuous decisions.

When a sampling algorithm is run on a continuous decision, the algorithm records the utility of the model for each parent and action value that is randomly sampled. When `setPolicy` is called on the algorithm, all of the generated samples are stored in the decision element. That is, no optimal policy is generated when `setPolicy` is called; the optimal action to take for a parent value is only computed when the model is queried for a decision with a particular parent value.

When the decision is queried, i.e., `getPolicy(p: T)` or `generate()` is invoked on the decision element, the optimal action for parent value `p` is computed using a nearest-neighbor method. The N closest samples to the parent value are retrieved from the stored samples, the expected utility is computed for each possible action, and the maximum is chosen as the optimal action for this parent value.

Since nearest-neighbor is used to find nearby parent values, a distance metric must also be defined for the parent type `T`. To use an `Element[T]` as the parent to a decision, the type `T` must implement the `Distance[T]` trait, defined as:

```
trait Distance[T] {
  def distance(that : T) : Double
}
```

This trait just defines a function that computes a Double distance between two values of the type. For built-in types (Double, Int and Boolean), we use Scala's implicit conversion mechanism to automatically handle conversion to a class that implements the `Distance[T]` interface so that no changes are needed by the user. For examples, the Double implementation is:

```
case class DoubleDistance(value : Double) extends Distance[Double] {
  def distance(that : Double) = math.abs(value-that)
}
implicit def double2Dist(x : Double) = DoubleDistance(x)
```

See the `Distance` class for more details on default conversions of basic types and parents that are element tuples. For user defined classes, all the user needs to do is implement a distance function in the `Distance` trait, and the type can be used as a parent to a decision element. For instance, we can use an element over the range of graphs as a parent to a decision by declaring the `dGraph` class as such:

```
class dGraph() extends Distance[dGraph] {
  …
  def distance(that: dGraph): Double = {
    …
  }
}
```

Since the number of samples generated from the algorithm may be large, and the optimal policy method retrieves the nearest neighbors for *every* parent value that is queried from the decision, computing the optimal action can be quite slow. To ameliorate this slowdown, Figaro stores the samples in an index. The default implementation is a VP-index, used for metric distances. Different indices can be created an integrated as well. See the `Index` and `DecisionPolicy` classes for more information.

Multiple decision models and policy generation

Figaro also supports for multiple decision models using a backward induction algorithm. In this algorithm, the optimal policies are computed in reverse order on a set of partially ordered decision variables. To create policies for multiple decision models, a user uses the multi-decision versions of the algorithms:

A multiple decision example can be found in MultiDecision.scala

```
val alg = MultiDecisionVariableElimination(List(utility1,
  utility2), decision1, decision2)
val alg = MultiDecisionImportanceSampling(10000,  List(utility1,
  utility2), decision1, decision2)
val alg = MultiDecisionMetropolisHastings(10000, maker:
  ProposalMakerType, 1000, List(utility1, utility2), decision1,
  decision2)
```

Note that the interface for the `MultiDecisionMetropolisHastings` is different than the `DecisionMetropolisHastings` algorithm. `MultiDecisionMetropolisHastings` needs a `ProposalMakerType`, since inside the algorithm, `DecisionMetropolisHastings` is run for each decision. The `ProposalMakerType` is defined as:

```
type ProposalMakerType = (Universe, Element[_]) => ProposalScheme
```

Only the one-time versions of the decision algorithms can be used for multi-decision models. To compute the optimal policy for every decision in the model, the user simply does, for example,:

```
val propmaker = (mv: Universe, e: Element[_]) =>
  ProposalScheme.default(mv)
val alg = MultiDecisionMetropolisHastings(200000, propmaker, 20000,
  List(value, cost), test, found)
alg.start()
```

The ProposalMaker for this small example just uses the default proposal for each instantiation of `DecisionMetropolisHastings` for a decision. However, we could also change the proposal scheme for each decision. There is also no need to call `alg.setPolicy`, since the multi-decision algorithm will set the optimal policy for each decision as it is needed for backward induction. Figaro will automatically compute the partial order of the decisions that are in the parameter list.

Learning model parameters from data

Figaro provides support for learning model parameters from data. In this section, a special type of compound element will be presented which allows a distribution to be learned from observed evidence. Details are given about the algorithm Figaro provides for learning parameters. Lastly, an example using parameters and learning algorithms to determine the fairness of a set of die rolls is presented.

Parameters and parameterized elements

This section discusses elements which are learnable parameters. For clarity, a distinction should be made on the meaning of the word *parameter* in this context. This is different from a method parameter or Scala type parameter. In this section, we use *parameter* to refer to a Figaro element which can be learned from data. There are currently two such types of parameters in Figaro – `BetaParameter` and `DirichletParameter`.

A customary illustration of parameter learning is to consider the outcomes of a coin flip and determine whether or not the coin is fair. In the case of a `Flip` element (which is a Bernoulli distribution), the conjugate prior distribution is a Beta distribution. If the coin is not fair, we would expect a prior distribution to have a higher value of alpha or beta (the shape variables of a Beta). First, we will create the conjugate prior distribution of a `Flip`

```
val fairness = BetaParameter(1,1)
```

The element `fairness` is the parameter we will use to model the bias of our coin. The behavior of a `BetaParameter` is similar to a normal `Beta` element, with a few extra methods and properties. Most importantly, we later use it to create a model learned from parameterized elements. Creation of a parameterized element is accomplished in exactly the same way as creating a compound element.

```
val f = Flip(fairness)
```

This element models a flip of a coin having the fairness specified by the beta parameter, using a value of `true` to represent heads and `false` to represent tails. We have actually created an instance of `ParameterizedFlip`, which is a special type of compound element. A `ParameterizedFlip` is created simply by providing a `BetaParameter` as the argument to `Flip`.

By using a `ParameterizedFlip`, the evidence we observe on `f` can be used to learn the value of `fairness`. Thus, the next step is to provide the data observed from flips of the coin. Values can be observed just as with other elements, by using `f.observe(true)` or `f.observe(false)`. We could also use conditions or constraints.

As a more detailed example, suppose we have seen 24 heads and 62 tails. One way to represent this data is in a Scala sequence. Note that for readability, the sequence is truncated here.

This example is found in FairCoin.Scala

```
val data = Seq('H', 'H', 'H', 'T', 'H', 'H', 'T', 'H', …
```

The following block of Scala code will iterate through each of the items in the sequence, create a Flip element using the parameter, and observe true or false based on the side of the coin:

```
data.foreach { d =>
  val f = Flip(fairness)
  if (d == 'H') {
    f.observe(true)
  } else if (d == 'T') {
```

```
      f.observe(false)
    }
  }
```

We have created a parameter, parameterized elements and considered a set of data. Note that each time a parameterize flip is created, it is using the same `BetaParameter`. It is now desirable to employ a learning mechanism to determine the fairness of the coin, and to create a new element corresponding to the learned value. This is possible by using a learning algorithm.

Expectation maximization

A learning algorithm can be used to determine the values of parameterized elements. At the end of the process, a parameter targeted by the learning algorithm can provide a new element according to the observed data. Presently, Figaro provides one learning algorithm, expectation maximization, based on variable elimination to compute sufficient statistics. Recall that expectation maximization is an iterative algorithm consisting of an expectation step and a maximization step. During the expectation step, an estimate is produced for the sufficient statistics of the parameter. The estimates are then used in the maximization step to find the most likely value of the parameter. This continues for a set number of iterations and converges toward the true parameter value.

From a practical standpoint, learning a parameter with expectation maximization is very simple. We need only provide the target parameter and, optionally, the number of iterations to the algorithm. The default number of iterations is 10. This can be done in the following way:

```
val learningAlgorithm = ExpectationMaximization(fairness)
learningAlgorithm.start

val coin = fairness.getLearnedElement
println("The probability of a coin with this fairness showing
  'heads' is: ")
println(coin.prob)
```

After the algorithm has finished running, we can retrieve an element learned from the parameter by using `fairness.getLearnedElement`. The element `coin` is a `Flip`, where the probability of producing true is determined from the data we observed above. It is important to make a distinction between parameterized elements and learned elements. Parameterized elements are compound elements and serve as our means of supplying data about a parameter. A learned element is an atomic element with arguments learned from the data we have observed.

After running the program, we see:

```
The probability of a coin with this fairness showing 'heads' is:
0.7159090909090909
```

We may want to make further explorations about the learned model. For instance, if we wanted to know the probability that two flips of this coin show the same side, we could use

```
val t1 = fairness.getLearnedElement
val t2 = fairness.getLearnedElement
val equal = t1 === t2
```

We can then use variable elimination to determine the probability that the coins show the same side:

```
val alg = VariableElimination(equal)
alg.start()
println("The probability of two coins which exhibit this fairness
  showing the same side is: " + alg.probability(equal, true))
alg.kill()
```

This results in the following output:

```
The probability of two coins which exhibit this fairness showing the
  same side is: 0.5932334710743803
```

A second example

In the previous sections, parameter learning was discussed using a Beta parameter. This section will explain the use of Dirichlet parameters. The Dirichlet distribution is a multidimensional generalization of the Beta with a variable number of concentration parameters or alpha values. These values correspond to the weight of each possible outcome in the posterior distribution. In a Dirichlet parameter with two dimensions, the alpha values might again correspond to the outcome of heads and tails, or true and false. Using a higher number of dimensions, we can model a number of different categories or outcomes.

Suppose we are given a set of data in which each record represents a roll of two die out of three possible die. The sum of the die is available, as well as which die were selected for the roll. However, the individual outcome of each die is not available. Our task is to learn the fairness of each die.

The first step is to define the possible outcomes from a dice roll. This is easily accomplished by using a Scala list:

This example is found in FairDice.Scala

```
val outcomes = List(1, 2, 3, 4, 5, 6)
```

Next, we create a parameter representing the fairness of each die:

```
val fairness1 = DirichletParameter(1.0,1.0,1.0,1.0,1.0,1.0)
val fairness2 = DirichletParameter(1.0,1.0,1.0,1.0,1.0,1.0)
val fairness3 = DirichletParameter(1.0,1.0,1.0,1.0,1.0,1.0)
val parameters = Seq(fairness1, fairness2, fairness3)
```

Each die is initially assumed to be fair. For convenience, we can place all three parameters in a Scala sequence named `parameters`. An item this sequence at index *i* can be accessed with `parameters(i)`. This sequence will help us to concisely observe the data from which the parameters are learned. We can represent the data in another sequence:

```
val data = Seq((2, 3, 8), (1, 3, 7), (1, 2, 3), (1, 2, 3), …
```

`data` is a sequence of 50 Scala tuples. The first two values in each tuple indicate which two die were chosen to roll. The third value is the sum of the two die.

To model the outcome of the sum, we can use an Apply element with a function which sums the outcome of its arguments.

```
def trial(p1: AtomicDirichletParameter, p2:
AtomicDirichletParameter, result: Int) = {
  val sum = (i1: Int, i2: Int) => i1 + i2
  val die1 = Select(p1, outcomes: _*)
  val die2 = Select(p2, outcomes: _*)
```

```
  val t = Apply(die1, die2, sum)
  t.observe(result)
}
```

The code section above defines a Scala function which accepts two Dirichlet parameters and an integer value. `val sum = (i1: Int, i2: Int) => i1 + i2` defines a Scala function which accepts two integer values and returns their sum. Next, two Select elements are created and parameterized by the input parameters. Note that the arguments to Select are different from what has been presented previously. Instead of directly enumerating each probability and outcome, we specify a Dirichlet parameter and the list of possible outcomes. The last two lines of `trial` apply the sum function to the die and observe the result. By calling the `trial` function for each tuple in the sequence, we can create a model learned from the data.

```
data.foreach { d =>
  if (d._1 == 1 && d._2 == 2) {
    trial(parameters(0), parameters(1), d._3)
  } else if (d._1 == 1 && d._2 == 3) {
    trial(parameters(0), parameters(2), d._3)
  } else {
    trial(parameters(1), parameters(2), d._3)
  }
}
```

Just as in the fair coin example, we create an expectation maximization algorithm and use the list of parameters as input. Additionally, this time we have chosen to raise the number of iterations to 100.

```
val numberOfIterations = 100
val algorithm = ExpectationMaximization(numberOfIterations,
  parameters: _*)
algorithm.start

val die1 = fairness1.getLearnedElement(outcomes)
val die2 = fairness2.getLearnedElement(outcomes)
val die3 = fairness3.getLearnedElement(outcomes)
```

The code block above will retrieve learned elements from the parameters. Note that for a Dirichlet parameter, a list of outcomes must be supplied as an argument to `getLearnedElement`. This is because the number of concentration parameters is may vary, and the type of the outcomes is not fixed. Running this code results in the following output, in which we see the model has estimated the probabilities of each value for each die. If one examines the full data declaration in the example code, it is quite easy to see that there are only three observed values of the sum of the die (3, 7 and 8), so the learning algorithm has correctly inferred that the most likely values of the die are 1, 2 and 6, respectively.

```
The probabilities of seeing each side of d_1 are:
   0.8188128994726971 -> 1
   0.02758618154457299 -> 2
   0.027187536514221913 -> 3
   0.02718754004129875 -> 4
   0.0720455051458327 -> 5
   0.027180337281376442 -> 6
```

```
The probabilities of seeing each side of d_2 are:
   0.024601050364553095 -> 1
   0.8475328824964514 -> 2
   0.024642652617350196 -> 3
   0.024642566789335425 -> 4
   0.024649300514031563 -> 5
   0.05393154721827825 -> 6

The probabilities of seeing each side of d_3 are:
   0.026691136667745217 -> 1
   0.028357892704022745 -> 2
   0.027128077618261647 -> 3
   0.027127956278181584 -> 4
   0.0271337269201185 -> 5
   0.8635612098116704 -> 6
```

Hierarchical reasoning

As was previously shown, Figaro is well suited for building PRMs due to the object-oriented nature of Figaro and Scala. PRMs are a powerful alternative to traditional Bayesian networks because they can represent structural uncertainty about the model. There are many different types of structural uncertainty: *type uncertainty*, in which the class of an object is unknown; *number uncertainty*, in which the number of objects to which an object is related is unknown; *existence uncertainty*, in which we do not know if a particular object exists; and *reference uncertainty*, in which we do not know to which other object a given object is related. Figaro presents an easy for users to model structural uncertainty in a probabilistic model and perform reasoning over that uncertainty.

Consider a situation with type uncertainty. Using Scala's object-oriented features, we can construct a class hierarchy that represents different features and properties of a parent class and use the hierarchy to reason about observations. For example, consider the abstract class below:

```
import com.cra.figaro.language._
import com.cra.figaro.algorithm.factored.VariableElimination
import com.cra.figaro.library.atomic.discrete.Uniform

abstract class Vehicle extends ElementCollection {
  val size: Element[Symbol]
  val speed: Element[Int]
  lazy val capacity: Element[Int] = Constant(0)
}
```

A `Vehicle` is an abstract class that contains a size, speed and capacity. The class is abstract since there are many types of vehicles and each type of vehicle may have different distributions of size, speed and capacity. Now, we will create some specific vehicles:

This example is found in Hierarchy.scala

```
class Truck extends Vehicle {
  val size: Element[Symbol] =
    Select(0.25 -> 'medium, 0.75 -> 'big)("size", this)
  val speed: Element[Int] = Uniform(50,60,70)("speed", this)
  override lazy val capacity: Element[Int] =
    Chain(size, (s: Symbol) => if (s == 'big) Select(0.5 -> 1000,
    0.5 -> 2000); else Constant(100))("capacity", this)
}

class Pickup extends Truck {
  override val speed: Element[Int] = Uniform(70,80)("speed", this)
  override val size: Element[Symbol] =
    Constant('medium)("size", this)
}

class TwentyWheeler extends Truck {
  override val size: Element[Symbol] = Constant('huge)("size", this)
  override lazy val capacity = Constant(5000)("capacity", this)
}

class Car extends Vehicle {
  val size = Constant('small)("size", this)
  val speed = Uniform(70,80)("speed", this)
}
```

Each class definition specifies more details about the properties of vehicles; most trucks, for example are either medium or big, except for a specific type of truck (Twenty Wheeler) which is always huge. This is an example of how Figaro can be combined with class hierarchies, where specific classes can override or modify the parent class probabilistic model.

We can now perform reasoning about the hierarchy. First, let us define some methods to create types of vehicles:

```
object Vehicle {
  def generate(name: String): Element[Vehicle] =
    Dist(0.6 -> Car.generate, 0.4 -> Truck.generate)(name, universe)
}
object Truck {
  def generate: Element[Vehicle] =
    Dist(0.1 -> TwentyWheeler.generate, 0.3 -> Pickup.generate,
    0.6 -> Constant[Vehicle](new Truck))
}object Pickup {
  def generate: Element[Vehicle] = Constant(new Pickup)
}
object TwentyWheeler {
  def generate: Element[Vehicle] = Constant(new TwentyWheeler)
}
object Car {
  def generate: Element[Vehicle] = Constant(new Car)
}
```

We have introduced a new element here: `Dist`. `Dist` is a combination of `Chain` and `Select`. `Dist` selects an element at random from the elements in the argument list, using the provided probabilities, and sets the value of the `Dist` as the value of the element selected. We use the objects above and the `Dist` element to generate vehicles from the vehicle hierarchy. That is, with probability 0.4, the `Vehicle.generate` produces a `Truck` class, and the specific type of truck generate is determined by `Truck.generate`, and so forth. Now let's create the rest of the Figaro model and perform some reasoning:

```
val myVehicle = Vehicle.generate("v1")
universe.assertEvidence(List(NamedEvidence("v1.size",
  Observation('medium))))
val isPickup = Apply(myVehicle, (v: Vehicle) =>
  v.isInstanceOf[Pickup])
val alg = VariableElimination(isPickup)
alg.start()
```

In this example, we're reasoning about the *type* of an instance of a class. First, we apply the evidence that the vehicle's size is medium using the `assertEvidence` method. Here, we apply the evidence by referring the name of the element, without actually specifying which element to apply the evidence; the 'medium will be applied to the size element in any instantiation of the `Vehicle` class. Next, we instantiate a `Boolean` element that is true when the type of the vehicle is an instance of `Pickup`. The `isInstanceOf[Pickup]` is a Scala operation that returns true when the variable is an instance of the specified class. We then run variable elimination on the model to determine the probability that the instantiated class is a `Pickup`.

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. Next, an explanation is given on how to create parameters and parameterized elements suitable for use with expectation maximization or other learning algorithms. 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 (1.0 -> _), collection)
  with Atomic[T] with Cacheable[T] {
  override def toString = "Uniform(" + options.mkString(", ") + ")"
}
```

The `with` keyword in Scala will add a trait to a class. Traits can be parameterized but they cannot have constructors

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. There is also one other trait that is extended in the `AtomicUniform`, `Cacheable[T]`. This trait is used to determine what type of chain should be created at the chain instantiation time. If the parent of a `Chain` extends `Cacheable`, a `CachingChain` is instantiated when a chain element is created. This trait is not required (elements by default are assumed to be not cacheable), but can result in increased performance if the support of the new element is small.

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, _)`. It is 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

Scala contains many transformations on sequences besides `toList`

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)
}
```

The * in the argument list defines a variable length argument list

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[List[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 two options. You can specify either a caching or non-caching version of the chain (which has preset cache capacities), or you can directly instantiate `Chain` with a specified cache capacity. Note that chains themselves do not extend the `Cacheable` trait, since the support of a can be infinite. 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:

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

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 instance, an atomic normal distribution would compute its randomness value using the standard normal distribution, and the value of the element would be the randomness shifted by the mean and scaled by the variance. For other atomic non-deterministic classes, the logic of the methods would be richer, but the general structure would be the same. Note that the `generateRandomness` function uses the Figaro random number generator called `random` to generate random values. One can use any random number generator to generate the randomness of an element. However, using the Figaro supplied random number generator allows one to globally set the seed of the generator in the Figaro package, thus enabling reproducible random processes.

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 and its randomness.

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, the `Variable(this)` line above 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 `ProbFactor.scala`. For atomic elements, the process should usually be similarly simple to that for binomials.

Making parameters and parameterized elements

Support for learning models from data is accomplished through parameters and parameterized elements. Defining new elements of these types requires the use of a couple of traits - `Parameter` and `Parameterized` – and a method to produce factors. Much of the required code is centered on the idea of sufficient statistics, as they are currently the means by which parameters are learned.

A parameter must extend the `Parameter` trait. This trait contains several important methods which allow use with the learning algorithms. First, the method `zeroSufficientStatistics` must provide an appropriate result for this parameter type. For example, a Beta parameter has two hyperparameters, alpha and beta. Hence, `zeroSufficientStatistics` returns a sequence of length two.

```
override def zeroSufficientStatistics (): Seq[Double] = {
  Seq(0.0, 0.0)
}
```

An additional method, `sufficientStatistics`, provides sufficient statistics with a value of 1.0 in the position specified by an index or value. This method can be useful when creating factors for parameterized elements. The `parameter` trait also defines a method for calculating the expected value of the parameter. Expected value is used during the parameter learning process, and also as an argument during the creation of learned elements.

We can create a parameterized version of an existing element by extending that type of element and including the `Parameter` trait. In the case of `Beta`, we have

```
class AtomicBetaParameter(name: Name[Double], a: Double, b:
  Double, collection: ElementCollection) extends
  AtomicBeta(name, a, b, collection) with
  Parameter[Double] with ValuesMaker[Double] {
```

Next, we must decide which values are actually learned within the parameter. In the case of Beta, the alpha and beta hyperparameters are already inputs to the `AtomicBeta` element. We will use these to represent prior knowledge or belief about the parameter. This is a good practice when extending existing elements into parameters. To facilitate parameter learning, we can create Scala variables which are modified by learning algorithms. Again, in the case of Beta, we can define

```
var alpha = a
var beta = b
```

The expected value for a beta distribution is well known and easily defined:

```
def expectedValue: Double = {
  (alpha) / (alpha + beta)
}
```

It is also necessary to define how the parameter value is maximized according to expected sufficient statistics. This method is used inside expectation maximization. In this case, we can simply set the value of alpha and beta to their corresponding values.

```
def maximize(sufficientStatistics: Seq[Double]) {
  require(sufficientStatistics.size == 2)
  alpha = sufficientStatistics(0) + a
  beta = sufficientStatistics(1) + b
}
```

Of critical importance is the `getLearnedElement` method. This method provides an atomic element which is the result of parameter learning. In most cases, the learned element can simply use the expected value of the parameter as its argument. Note that when creating the learned element, we must be sure to supply arguments for the element name and collection.

```
def getLearnedElement: AtomicFlip = {
  new AtomicFlip("", expectedValue, collection)
}
```

Having created a parameter, we may now create an element which uses it. Compound elements which use `Parameter` as their arguments are defined by the `Parameterized` trait. This trait is quite simple and contains only a reference to the element's parameter. Continuing the example, we can create a version of `Flip` which uses `BetaParameter` in the following way:

```
class ParameterizedFlip(name: Name[Boolean], override val parameter:
  AtomicBetaParameter, collection: ElementCollection)
  extends Element[Boolean](name, collection) with Flip with
  Parameterized[Boolean]
```

This class inherits most of its behavior from Flip. Much like a compound flip, the probability of producing true is derived from the BetaParameter:

```
def probValue = parameter.value
```

If an existing element is being extended, it is advisable to define a factory method in the companion object which accepts a `Parameter` element as input, and creates an instance of the parameterized element. To illustrate, consider the apply method for `ParameterizedFlip`:

```
def apply(prob: AtomicBetaParameter)(implicit name: Name[Boolean],
collection: ElementCollection) =
    new ParameterizedFlip(name, prob, collection)
```

By defining this method, we are able create a parameterized flip simply by supplying a beta parameter as an argument to `Flip`.

To use a parameterized element with expectation maximization, the last step is to provide a method for making factors in SufficientStatisticsFactor.scala. This class creates factors intended to be used with learning algorithms. Each factor entry consists of a probability and a map from parameters to sufficient statistics. The map from parameters to sufficient statistics is created by the learning algorithm which uses the factors. This class is discussed in further detail in the Factors and Factored Algorithms section.

```
private def makeFactors(flip: ParameterizedFlip):
  List[Factor[(Double, Map[Parameter[_], Seq[Double]])]] = {

  val flipVar = Variable(flip)
  val factor = new Factor[(Double, Map[Parameter[_],
    Seq[Double]])](List(flipVar))
  val prob = flip.parameter.expectedValue
  val i = flipVar.range.indexOf(true)

  val falseMapping = mutable.Map(parameterMap.toSeq: _*)
  val trueMapping = mutable.Map(parameterMap.toSeq: _*)

  trueMapping.remove(flip.parameter)
  falseMapping.remove(flip.parameter)
  trueMapping.put(flip.parameter, Seq(1.0, 0.0))
  falseMapping.put(flip.parameter, Seq(0.0, 1.0))
  factor.set(List(i), (prob, trueMapping))
  factor.set(List(1 - i), (1.0 - prob, falseMapping))
```

```
    List(factor)
  }
```

First, note that for the value of `prob`, we use the expected value of the parameter. For a regular compound `Flip`, this would be the weight specified by the value of another element. However, since the hyperparameters of the parameter can change as a learning algorithm is executed, it is important to use expected value.

When creating factors for parameterized elements, care should be taken to ensure that the correct sufficient statistics correspond with the possible outcomes of the element. In the code block above, an entry created for both possible outcomes of the `Flip`. Each entry contains sufficient statistics with 1.0 in the position corresponding to the outcomes of true and false, respectively. Note that we also could have used the `sufficientStatistics` method in the Parameter trait to retrieve sequences corresponding to these outcomes.

The map structure is necessary because there may be multiple parameters being learned at a time, and sufficient statistics should only be counted for the parameter used by the element this factor is being created for. This map, called `parameterMap`, is created automatically by the learning algorithm based on its target parameters and provided as an input to instances of `SufficientStatisticsFactor`.

Consider the following line of code:

```
val falseMapping = mutable.Map(parameterMap.toSeq: _*)
```

This line created an empty Scala map of parameters to sequences of double values, and adds the contents of the parameter map. The result is that a mapping is created in which sufficient statistics for all parameters are zero. Then, in the subsequent lines, the entry in the map associated with this parameter is replaced with a value corresponding to a specific outcome of the parameterized element.

By creating factors with sufficient statistics, we can use implementations of algorithms like variable elimination to assist in learning parameter values. For instance, each iteration of expectation maximization uses variable elimination to produce an estimate of the sufficient statistics.

As a final note, if one is interested in creating learning algorithms that do not rely on sufficient statistics, the same traits for parameters and parameterized elements should be still used. Simply create an algorithm which learns the values of the hyperparameters and sets their values, then retrieves the resulting model by using `getLearnedElement`.

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 for an element is achieved through the `nextRandomness` method. In Metropolis-Hastings, the acceptable probability of a sample is defined as

$$\frac{P(r^1 \rightarrow r^0)P(r^1)}{P(r^0 \rightarrow r^1)P(r^0)}$$

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 $P(r^0 \rightarrow r^1)$ indicates the probability of `nextRandomness` returning r^1 when its argument is r^0. The `nextRandomness` method returns three values; the new randomness, the transition probability ratio ($P(r^1 \rightarrow r^0)/P(r^0 \rightarrow r^1)$) and the model

probability ratio ($P(r^1)/P(r^0)$). These ratios a returned as separate values because some algorithms, such as simulated annealing, need to access these ratios before they are multiplied together.

By default, the `nextRandomness` method simply uses the element's `generateRandomness` method and returns 1.0 for the both probability ratios. 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, (1.0 - probValue) / probValue)
  else (uniform(0.0, probValue), 1.0, 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 other hand, if the randomness is greater than `probValue`, the new randomness is chosen uniformly between 0 and `probValue`, which would make the next value true. The probability of going from false to true or from true to false are both 1, so the transition probability ratio is 1. However, the density of false is `1 - probValue`, while the density of true is `probValue`, so in the first case (new value is false), the model probability ratio is `(1.0 - probValue) / probValue`, and vice versa.

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, most probable explanation, 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 learning 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.

General Considerations

All algorithms inherit from the `Algorithm` class, which provides a basic framework for algorithms, including starting, stopping, and killing them. `Algorithm` contains `initialize` and `cleanup` methods. The default implementation of these methods is to do nothing. You can override this for your algorithms if they require bookkeeping. For example, probability of evidence algorithms assert the named evidence at the beginning and remove it at the end, so they override these methods as follows:

```
def initialize(){
  super.initialize()
  // assert the evidence
}

def cleanup() {
  // remove the evidence
  super.cleanUp()
}
```

Note that we make sure to do the superclass's initialization and cleanup, and note that the superclass's initialization happens before this class, and its cleanup happens after this class.

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, you can use the Scala `asInstanceOf[T]` method of the sample.

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

Operations in factored algorithms are defined by a semiring algebraic structure. There are several semiring definitions in the package `com.cra.figaro.algorithm.factored`. Each semiring defines a product and sum operation, and a value for zero and one which satisfy a set of properties. Different semirings are appropriate for certain algorithms and data types; however, the most frequently used set of operations is `SumProductSemiring`.

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:

- `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 and 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.

Learning algorithms

This section describes how to use learning algorithms and sufficient statistics factors. Learning algorithms (using sufficient statistics) require that the sufficient statistics of a parameter are modified with the learned result. Expectation maximization uses a version of variable elimination to produce an estimate of the sufficient statistics at each iteration. This version of variable elimination is called `SufficientStatisticsVariableElimination`. It relies on the `SufficientStatisticsFactor` class described in the 'Parameters and Parameterized Elements' section.

The semiring used by `SufficientStatisticsVariableElimination` is called `SufficientStatisticsSemiring`. This provides operations on sufficient statistics which can be used by factored algorithms like variable elimination.

Any factored inference algorithm can be adapted to use the sufficient statistics semiring and factors. First, the algorithm it must accept as an argument a mapping from the target parameters to their sufficient statistics. An instance of the sufficient statistics semiring class is then created using this mapping. Likewise, an instance of the sufficient statistics factor building class is instantiated. In sufficient statistics variable elimination, this is done as follows:

```
val semiring = new SufficientStatisticsSemiring(parameterMap)
val statFactor = new
  SufficientStatisticsFactor(parameterMap)
```

The only other requirement to use the parameter mapping is to create an instance of the sufficient statistics factor class inside the algorithm, call the `getFactors` method, and clear the factors when the algorithm is finished. This is important, because otherwise the factors will be cached and will not reflect changes in parameter values unless they are regenerated. The factored can be cleared by using the `removeFactors` method

At the end of the inference algorithm, it must provide the sufficient statistics factors for all of the parameters. This can be accomplished by calling the method `get` on the factor resulting from inference. For example, in variable elimination, we can do `result = finalFactor.get(List())` to retrieve the resulting sufficient statistics.

`SufficientStatisticsVariableElimination` does not change the values of parameters. It only produces an estimate of the sufficient statistics given the observed data. The actual modification of parameter elements is handled in the maximization step of expectation maximization, using the `maximize` method defined by the parameter. Other learning algorithms might use the sufficient statistics produced by variable elimination in a different way. The important part is to choose the hyperparameter values in `Parameter` correctly, and use methods to modify them inside the learning algorithm.

However, if an inference algorithm like `SufficientStatisticsVariableElimination` is to be used as part of the learning algorithm, then the parameter mapping must be derived from the input parameters to their respective sufficient statistics. We have already seen in several places where this parameter mapping is used. It is inside a learning algorithm that the mapping is actually created. This is easy to do, as shown in the `ExpectationMaximization` class:

```
  protected val paramMap : immutable.Map[Parameter[_],Seq[Double]]=
immutable.Map(targetParameters.map(p => p ->
p.zeroSufficientStatistics):_*)
```

This code simply maps each parameter to its sufficient statistics. This map can now be provided as an argument to an inference algorithm to reason about the parameters and their sufficient statistics.

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 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, probability of query algorithms, probability of evidence, and most likely value algorithms are all different categories. Figaro provides some infrastructure to help with creating a new kind of algorithm. We will illustrate how this is done for most likely value 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`. To define a new category of algorithm, you extend algorithm and define methods for the different ways the algorithm can be queried. For example,

The val in front of the universe argument indicates that universe is a field of MPEAlgorithm that can be accessed

```
abstract class MPEAlgorithm(val universe: Universe)
  extends Algorithm {
  /**
   * Returns the most likely value for the target element.
   */
  def mostLikelyValue[T](target: Element[T]): T
}
```

An `MPEAlgorithm` takes the universe on which it is defined as an argument. It provides one query method, which returns the most likely value of a target method. This method is abstract (it has no implementation) and must be implemented in a particular implementation of `MPEAlgorithm`.

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

```
trait OneTimeMPE extends MPEAlgorithm with OneTime
```

That's all there is to it. Figaro's `OneTime` implements the general algorithm operations for starting, stopping, and killing algorithms (fairly trivial in this case). It also declares an abstract `run()` method, which is called when the algorithm is started. This method must be implemented in implementations of `OneTime`, and, by extension, implementations of `OneTimeMPE`. An example of a one-time MPE algorithm is a one-time Metropolis-Hastings annealer, which is captured in the `OneTimeMetropolisHastingsAnnealer` class. This class extends (indirectly) `OneTimeSampler`, which is defined as follows.

```
trait OneTimeSampler extends Sampler with OneTime {
  /**
   * The number of samples to collect from the model.
   */
  val numSamples: Int

  /**
   * Run the algorithm, performing its computation to completion.
   */
  def run() = {
    resetCounts()
    for { i <- 1 to numSamples } { doSample() }
    update()
  }
}
```

In this case, `run` resets the statistics of the sampler, calls `doSample` the required number of times, and updates the representation of the result. Different categories of algorithms can use the same general sampling process; for example, the one-time importance sampling algorithm for computing the probability of query variables also inherits from `OneTimeSampler`.

For the anytime version of an `MPEAlgorithm`, we need to do more work to define the services provided by the thread that computes the MPE and the responses it produces.

```
trait AnytimeMPE extends MPEAlgorithm with Anytime {
  /**
   * A message instructing the handler to compute the most likely
value of the target element.
   */
  case class ComputeMostLikelyValue[T](target: Element[T]) extends
Service

  /**
   * A message from the handler containing the most likely value of
the previously requested element.
   */
```

```
case class MostLikelyValue[T](value: T) extends Response

def handle(service: Service): Response =
  service match {
    case ComputeMostLikelyValue(target) =>
      MostLikelyValue(mostLikelyValue(target))
  }
 }
```

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. A request can be sent to the runner to compute the most likely value of a target element. The syntax for sending the message to the runner is

```
runner ! Handle(ComputeMostLikelyValue(target))
```

which sends a message whose content is `Handle(ComputeMostLikelyValue(target))`. The runner dispatches this message to a method called `handle` (which is abstract in `Anytime` and defined in `AnytimeMPE`. This method knows how to handle `ComputeMostLikelyValue` – it calls the method `mostLikelyValue`, which, as we have seen, is abstract in `MPEAlgorithm` and must be provided by an implementation. It turns the resulting value `v` into a message `MostLikelyValue(v)`, which is sent back to the caller from the runner.

Case classes are simple classes in Scala that contain some values. Case objects are like classes but with no values; they are essentially constants.

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 class ComputeMostLikelyValue(target:
   Element[T]) extends Service
   ```
2. Create a case class or object to represent the responses provided by these services. Here,
   ```
   case class MostLikelyValue[T](value: T)
     extends Response
   ```
3. Create a handler in the method `handle` that takes a service, performs some computation, and returns a response.
4. In each method that provides an interface to querying the algorithm
   a. Send a message to the runner asking for the appropriate service
   b. Receive a message from the runner, extract the result, and return it.

Conclusion

As you can see, there's quite a lot to Figaro. We hope you will find it useful in your probabilistic reasoning projects. If you have any comments, suggestions, bug fixes, etc., please refer to the GitHub site (https://github.com/p2t2) for the best way to contact us, or send them to figaro@cra.com.

Some of the planned next steps for Figaro are:

- Approximate factor-based algorithms like belief propagation
- Lazy algorithms for reasoning about potentially infinite domains
- Better dynamic reasoning algorithms
- Distributed reasoning algorithms

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

Thanks for reading, and enjoy!