FIGARO TUTORIAL

Avi Pfeffer, Brian Ruttenberg, Michael Howard,
Alison O'Connor, and William Kretschmer
Charles River Analytics

charles river analytics

CONTENTS

1 INTRODUCTION

1.1 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
- Belief propagation
- Lazy factored inference for infinite models
- Importance sampling
- Metropolis-Hastings, with an expressive language to define proposal distributions
- Support computation
- Most probable explanation (MPE) using variable elimination, belief propagation, or simulated annealing
- Probability of evidence using importance sampling, belief propagation, variable elimination, and particle filtering
- Particle filtering
- Factored frontier
- Gibbs sampling
- 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`.

1.2 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.

The tutorial is not a complete and comprehensive guide to all of Figaro's features. The official reference is the Scaladoc, which documents Figaro's methods. For a broader introduction to probabilistic programming and Figaro, see "Practical Probabilistic Programming" by Avi Pfeffer, published by Manning (`http://www.manning.com/pfeffer/`).

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.

1.3 INSTALLATION

Please see the Quick Start Guide for instructions on installing Figaro. There are several ways to use Figaro, including just using the binary distribution or compiling from the source code.

Figaro is maintained as open source on GitHub. The GitHub project is Probabilistic Programming Tools and Techniques (P2T2), located at `https://github.com/p2t2`. If you want to see the source code and build Figaro yourself, please visit our GitHub site. We welcome contributions from the community.

2 HELLO WORLD!

Make sure Scala version 2.11.4 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 the directory to that location and enter the line below in 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 the `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 1.0. 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.

3 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.

3.1 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:

Figaro classes are capitalized, while Scala reserved words are not

```
Constant(6)
```

This defines the probabilistic model that produces the integer 6 with probability 1.0. Another atomic element is:

```
Constant("Hello")
```

which produces the string "Hello" with probability 1.0. These two examples illustrate that every Figaro element has a *value type*, which in the first case is `Int` and in the second case is `String`. The value type is the type of values produced by the probabilistic model defined by the element.

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

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

A constant is a particular type of element that is an instance of the `Constant` class, which is a subclass of `Element`. So, more specifically, the first element Scala uses type inference, so the value type of the parameter can often be omitted at class creation (the compiler will determine the type) 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.

3.2 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.

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.

3.3 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:

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

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

represents the `Element[Int]` in which with probability 0.7, `Constant(1)` is chosen, producing 1 with probability 1.0, 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.

3.4 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.

3.5 APPLY

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.

There are a variety of operators and functions that are defined using Apply. For example:

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

- ^^ 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.

3.6 processes and containers

New to Figaro 3.0 is a collections library. The general trait of Figaro collections is Process, which represents a possibly infinite collection of random variables. Formally, a Process is a mapping from an index set to an element. A Process is parameterized by two types: the type of the indices and the type of the values of the elements in the collection. The Process is an extremely general class that can be used to represent things like Gaussian processes or continuous time Markov processes.

When creating a Process, you need to specify how elements in the collection are generated given an index. Not only that, in some collections, the elements are dependent. Therefore, the Process class contains a method to generate elements for many indices simultaneously, including the dependencies between them. This method must also be provided by the user. If all the elements are independent, you can use the IndependentProcess trait to specify this method.

There are a number of operations that are defined on every process. These include:

- Getting the element at an index. If p is a Process[Int, Double], p(5) gets the Element[Double] at index 5. This method throws IndexOutOfRangeException if no element is defined at index 5.

- Getting elements at many indices simultaneously, for example, using p(List(4,5,6)). This method can also throw IndexOutOfRangeException. The method creates a Scala Map from indices to elements. Any elements representing dependencies between the elements at these indices are also created but they are not returned by this method.

- Safely getting an optional element at an index. p.get(5) will return an Element[Option[Double]]. This element will always have value None if no element is defined at index 5.

- Safely getting an optional element at many indices.

- Mapping the values of every element in the process through a function. For example, `p.map(_ > 0)` will produce a `Process[Int, Boolean]`.
- Chaining the value of every element in the process through a function that returns an element. For example, `p.chain(Normal(_, 1))` will produce a new collection in which every element is normally distributed with mean equal to the value of the corresponding element in the original process.

If you have a finite index set, you can use a `Container`, which takes a sequence of indices. Because they are finite, containers have many more operations defined on them, including a variety of folds and aggregates. See the Scaladoc for the available operations.

A specific kind of container is a `FixedSizeArray`, which takes the number of elements as the first argument and a function that generates an element for a given index as the second argument. For example, `new FixedSizeArray(10, (i: Int) => Flip(1.0 / (i + 1)))` creates a container of ten Boolean elements.

There is a `Container` constructor that takes any number of elements and produces a container with those elements. For example, `Container(Flip(0.2), Flip(0.4))` creates a container consisting of the two elements.

You can, naturally, have elements whose values are processes or containers. Figaro provides the `ProcessElement` and `ContainerElement` classes to represent these. Similar operations are defined for `ProcessElement` and `ContainerElement` as for processes and containers.

`VariableSizeArray` represents a collection of an unknown number of elements, where the number is itself defined by an element. It takes two arguments, the number element, and a function that generates an element for a given index, like a fixed size array. For example, `VariableSizeArray(Binomial(20, 0.5), (i: Int) => Flip(1.0 / (i + 1))` creates a container of between 0 and 20 Boolean elements.

4 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.

4.1 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)
```

`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

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 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 1.0. 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),
   true -> Flip(0.7))
```

Scala statements can be written on multiple lines

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._
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 OneOf, NoneOf and *.

4.2 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`.

Constraint values should always be non-negative. Also, although it is not strictly enforced, we recommend that constraint values be at most 1. Some algorithms compute upper bounds on probabilities and need to assume an upper bound on constraint values. An upper bound of 1 is assumed, so if the actual value can be higher, these algorithms will be incorrect. For algorithms that don't compute upper bounds in this way, it doesn't matter. Currently, the only algorithms that compute upper bounds like this are the lazy factored inference algorithms. If you have a constraint value greater than 1, a warning will be issued.

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 } {
```

Single line function definitions in Scala do not need bracketing

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

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

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

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

4.3 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.

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

```
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))
```

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:

This example can be found in SimpleMovie.scala

```
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) {
  def probAward(quality: Symbol, famous: Boolean) =
    (quality, famous) match {
    case (’low, false) => 0.001
    case (’low, true) => 0.01
    case (’medium, false) => 0.01
    case (’medium, true) => 0.05
    case (’high, false) => 0.05
    case (’high, true) => 0.2
    }
  val award = SwitchingFlip(Apply(movie.quality,
actor.famous, (q: Symbol, f: Boolean) => probAward(q, f)))
}

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: List[Boolean]) =
  awards.count((b: Boolean) => b) == 1
val allAwards: Element[List[Boolean]] =
  Inject(appearances.map(_.award):_*)
allAwards.setCondition(uniqueAwardCondition)
```

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 list 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 list 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 list is true. So awards.count((b: Boolean) => b) counts the number of elements in the list that are true.

We then define the allAwards element to be the element over lists 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 list and returns a new list 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.

4.4 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. This code also illustrates a use of Figaro collections.

This example can be found in MutableMovie.scala

```
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 qualities = Container(movies.map(_.quality):_*)
  lazy val numGoodMovies = qualities.count(_ == 'high)
  lazy val famous = Chain(numGoodMovies, (n: Int) =>
  if (n >= 2) Flip(0.8) else Flip(0.1))
}

class Movie {
  var actors: List[Actor] = List()
  lazy val skills = Container(actors.map(_.skillful):_*)
  lazy val actorsAllGood = skills.exists(b => b)
  lazy val probLow =
  Apply(actorsAllGood, (b: Boolean) => if (b) 0.2; else
0.5)
  lazy val probHigh =
  Apply(actorsAllGood, (b: Boolean) => if (b) 0.5; else
0.2)
  lazy val quality =
  Select(probLow -> ’low, Constant(0.3) -> ’medium,
probHigh -> ’high)
}

class Appearance(actor: Actor, movie: Movie) {
  actor.movies ::= movie
  movie.actors ::= actor

  def probAward(quality: Symbol, famous: Boolean) =
    (quality, famous) match {
    case (’low, false) => 0.001
    case (’low, true) => 0.01
    case (’medium, false) => 0.01
    case (’medium, true) => 0.05
    case (’high, false) => 0.05
    case (’high, true) => 0.2
    }
  lazy val award = SwitchingFlip(Apply(movie.quality,
actor.famous, (q: Symbol, f: Boolean) => probAward(q, f)))
}

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: List[Boolean]) =
  awards.count((b: Boolean) => b) == 1
val allAwards: Element[List[Boolean]] =
```

::= is Scala shorthand for list concatenation

```
  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. 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. Care should be taken with declaring elements as lazy. Side effects and unintended consequences can occur if a lazy element declared outside a `Chain` is first required (i.e., created) during the execution of a `Chain`.

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.

4.5 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.

4.6 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 V8 extends Engine {
  val power = Select(0.8 -> 'low, 0.2 -> 'high)
}
class V6 extends Engine {
  val power = Select(0.8 -> 'low, 0.2 -> 'high)
}

object MySuperEngine extends V8 {
  val power = Constant('high)
}
class Car {
  val engine = Uniform[Engine](new V8, new V6,
MySuperEngine)
  val speed = CPD(?,
  'high -> Constant(90.0),
  'medium -> Constant(80.0),
  'low -> Constant(70.0))
}
```

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 = Uniform[Engine](new V8, new V6,
MySuperEngine)("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 last 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 `V8, V6, and MySuperEngine` 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 V8 extends Engine {
  val power = Select(0.8 -> 'low, 0.2 -> 'high)("power",
this)
}
class V6 extends Engine {
  val power = Select(0.8 -> 'low, 0.2 -> 'high)("power",
this)
}
object MySuperEngine extends V8 {
  val power = Constant('high)("power", this)
}
class Car extends ElementCollection {
  val engine = Uniform[Engine](new V8, new V6,
MySuperEngine)("engine", this)
  val speed = CPD(
   get[Symbol]("engine.power"),
   'high -> Constant(90.0),
   'medium -> Constant(80.0),
   'low -> Constant(70.0))
)
}
```

4.7 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:

This example can be found in MultiValuedReferenceUncertainty.scala

```
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 = MakeList(Select(0.5 -> 1, 0.5 -> 2),
makeComponent)("components", this)

val sum = getAggregate((xs: MultiSet[Int]) => (0 /: xs)(_ +
_))("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 `sum` 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 code defining `sum` might look mysterious. This code 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.

5 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.

5.1 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.

5.2 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.

5.3 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 exact factors. Atomic continuous elements with infinite range are handled in one of two ways. As discussed later in the section, abstractions can be used to make variable elimination work for continuous classes. If no abstractions are defined for continuous elements, then each continuous element is sampled and a factor is created from the samples. Figaro outputs a warning in this instance to ensure the user intends to use a continuous variable in a factored algorithm. 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.

For convenience, Figaro also provides a one-line query method using variable elimination. Just use:

`VariableElimination.probability(element, value)`

This will take care of instantiating the algorithm and running inference and returns the probability that the element has the given value.

5.4 APPROXIMATE INFERENCE USING BELIEF PROPAGATION

Figaro also contains another factored inference algorithm called belief propagation (BP). BP is a message passing algorithm on a factor graph (a bipartite graph of variables and factors). On factor graphs with no loops, BP is an exact inference algorithm. On graphs with loops (loopy factor graph), BP can be used to perform approximate inference on the target variables. Note that in Figaro, the way that Chains are converted to factors always produces a loopy factor graph, even if the actual definition of the model contains no loops. Therefore, most inference with BP in Figaro is approximate.

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 and create a factor graph from the factors.
3. Pass messages between the factor nodes and variables nodes for the specified number of iterations.
4. Queries are answered on the targets using the posterior distributions computed at each variable node.

Steps 1 and 2 operate in the same manner as variable elimination, and the same restrictions on factors also applies. Just like in 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._
import
com.cra.figaro.algorithm.factored.beliefpropagation._

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 bp = BeliefPropagation(100, e2)
```

This will create a `BeliefPropagation` object that will pass messages on a factor graph created from the universe containing `e1, e2, and e3`. The first argument is the number of iterations to pass messages between the factor and variable nodes. However, it won't perform BP immediately. To tell it to run the algorithm, you have to say:

```
bp.start()
```

When this call terminates, you can use `bp` to answer the same queries as defined in the variable elimination section. You can also use a one-line shortcut like for variable elimination.

Continuous elements are handled in BP the same was as in variable elimination (abstractions or sampled).

5.5 lazy factored inference

Ordinarily, factored inference algorithms like variable elimination and belief propagation cannot be applied to infinitely recursive models. It's easy to define such models, such as probabilistic grammars for natural language, in Figaro. Figaro provides lazy factored inference algorithms that expand the factor graph to a bounded depth and precisely quantify the effect of the unexplored part of the graph on the query. It uses this information to compute lower and upper bounds on the probability of the query.

To use lazy variable elimination, create an instance of `LazyVariableElimination`. You can use the `pump` method to increase the depth of expansion by 1. You can also use `run(depth)` to expand to the given depth. You can find an example of lazy variable elimination in action in `LazyList.scala` in the Figaro examples. You can also use lazy belief propagation.

5.6 importance sampling

Figaro's importance sampling algorithm is actually a combination of likelihood weighting and rejection sampling. When Figaro's Importance sampling encounters a condition, it attempts to push the condition through any Chains in the model and weight the sample by the probability of the condition. However, it is not always possible to do this (especially when it encounters an Apply), so in that case it will reject a sample if it does not match the constraint. 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.resume()` 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()
```

Importance sampling also provides a one-line query shortcut.

There is also a parallel version of Importance sampling that uses Scala's built in parallel collections. The interface to use parallel Importance sampling is similar to the original algorithm with a few exceptions. First, this version sampling uses a model generator, which is a function that produces a universe (Importance sampling is run in parallel on separate but identical universes). Second, the user must indicate the number of threads to use. Finally, instead of taking a set of elements to query, the algorithm takes in a set of references, where each reference refers to the same element on each of the parallel universes.

5.7 METROPOLIS-HASTINGS 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. When using the one-time Metropolis-Hastings algorithm, if the supplied constraints are bound between 0 and 1, the *constraintsBound* Boolean flag should be set to *true* to optimize the algorithm's efficiency prior to starting the algorithm. For example:

```
val mh = MetropolisHastings(200000, ProposalScheme.default,
e2)
mh.constraintsBound = true
mh.start()
```

Metropolis-Hastings also provides a one-line query shortcut.

5.7.1 *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(200000, chooseScheme, 5000,
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.

5.7.2 *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.

5.8 GIBBS SAMPLING MARKOV CHAIN MONTE CARLO

Figaro also provides another Markov chain Monte Carlo algorithm known as Gibbs sampling. Gibbs sampling is an algorithm that traditionally will iterate through all the variables in the model, and sample each variable conditioned on the rest of the model (or the Markov blanket, as that is all that is needed). By successively sampling variables in the model in this manner, the Markov chain eventually reaches convergence, and subsequent samples from the model are from the joint distribution defined by the model.

Figaro's Gibbs sampling is similar to traditional implementations of Gibbs sampling except for two key differences: It is implemented on a factor graph, and blocks of variables are sampled at each iteration instead of a single variable (this is required because of chains). This means that when Gibbs sampling is run, factors are generated based on the model; continuous variables are sampled into factors. Hence, Gibbs samples in Figaro is really a mix between a factored algorithm and a sampling algorithm.

To create an anytime Gibbs sampling algorithm, use:

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

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 gs = Gibbs(e2)
```

Gibbs sampling takes three additional optional arguments. The first represents the burn-in, which is the number of steps the algorithm goes through before collecting samples, the second is the number of steps between samples, and the final is the method of creating blocks (which most people do not need to change). 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) Gibbs sampling algorithm, simply provide the number of samples as the first argument. Gibbs sampling also provides a one-line query shortcut.

5.9 STRUCTURED FACTORED INFERENCE

Figaro contains a set of algorithms known as structured factored inference (SFI) algorithms. These are not new algorithms per se, but rather a method of decomposing a model into a set of smaller sub–models that are solved recursively (i.e., structured) using one of Figaro's three existing factored algorithms (VE, BP, and Gibbs sampling). There is a rich and extensible library of the different strategies that guide how a model is decomposed, and which algorithms are applied to the generated sub–models. However, we provide a base set of structured algorithms.

Figaro also includes an interface for lazy SFI (LSFI) algorithms, which combine the functionality of lazy factored inference (Section 5.5) and SFI. LSFI can operate on infinite or recursive models, where it produces bounds on queries rather than an exact queries. Possible queries include bounds on the probability of a value, bounds on the probability of a predicate evaluating to true, or bounds on the expectation of a bounded function over the values of an element. Note that in order to produce correct bounds, LSFI algorithms require that all constraints are between 0 and 1; otherwise an exception will be thrown.

5.9.1 *The process of SFI: ranging, refining, solving*

SFI proceeds in two steps: refining and solving. Refining includes the process of decomposing the model into sub–models, and also involves computing a finite range for each element. The refining process to use is specified by a strategy, but refining strategies also make calls to an algorithm-defined ranging strategy as a subroutine.

Currently, all built-in refining strategies decompose a model by its Chains. More specifically, refining strategies create sub–models for Chains, one for each value of the parent of a Chain.

The solving step involves traversing the graph of sub–models and applying a factored algorithm to each sub–model until the top-level model has a solution to its inference problem. Solving is likewise defined by a strategy.

Figaro includes a number of built-in ranging, refining, and solving strategies, but as we will see later, these strategies are configurable in general.

5.9.2 *One-time and anytime SFI*

Figaro offers both one-time and anytime SFI algorithms. One-time SFI algorithms perform a single pass of refining and solving. Most of the default algorithms included are one-time algorithms. Anytime SFI algorithms alternate refining and solving repeatedly to produce better solutions over time. This is particularly useful for LSFI, where one might choose to incrementally expand more of the model at each iteration. Alternatively, anytime SFI is applicable to models containing elements with infinite support, as it allows the algorithm to choose better finite approximations of these distributions at each iteration.

5.9.3 *Basic SFI algorithms*

Though SFI is intended to be highly configurable, Figaro also contains several complete algorithms that work with reasonable defaults. The basic structured algorithm recursively decomposes a model by its Chains until the entire model is decomposed. A lazy variant works similarly, but only recursively decomposes to a finite depth. To create such an SFI algorithm that uses the same algorithm for each sub–model, use:

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

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 sve = StructuredVE(e2)
```

The algorithm can then be used like a normal one-time Figaro algorithm. There are also one-time structured BP and Gibbs algorithms, as well as a lazy structured VE algorithm with one-time and anytime variants.

To use a strategy that automatically chooses between VE, BP, and Gibbs sampling, use:

```
import
com.cra.figaro.algorithm.structured.algorithm.hybrid._
val structAlg = StructuredVEBPGibbsChooser(0.0, 0.5, 100,
1000, e2)
```

Where the first argument is the threshold that determines when VE is run (e.g., the cost of running VE is less than the threshold). The second argument controls when to choose between BP and Gibbs sampling. If the fraction of deterministic elements in the model is greater than the second argument, then BP is run, otherwise Gibbs is run. The following arguments are the number of BP and Gibbs iterations to perform, and the final argument is the query target.

Additional SFI and LSFI algorithms are found in the com.cra.figaro.algorithm.structured.algorithm package; see the Scaladoc for details on configuring them.

5.9.4 *Custom SFI algorithms*

The previous sections cover the basics of SFI, and are sufficient for users looking to use the base algorithms described above. Nevertheless, advanced users may want to customize SFI algorithms for better performance on particular models. The sections that follow describe the inner workings of SFI in technical detail, explaining the configuration options for each part of the SFI process.

One can customize SFI algorithms by extending from one of the base abstract classes (e.g. LazyStructuredProbQueryAlgorithm) and mixing in the appropriate one-time or anytime traits. This leaves the user to define the ranging, refining, and solving strategies to use. It is important to note that whereas a single ranging strategy is used over the course of inference, most refining and solving strategies are not reusable. Thus, for anytime SFI, one should generally override the refiningStrategy() and solvingStrategy() methods to return a new strategy for each iteration of inference, like so:

```
import com.cra.figaro.algorithm.structured.algorithm._
import com.cra.figaro.algorithm.structured.solver._
import com.cra.figaro.algorithm.structured.strategy.range._
import
com.cra.figaro.algorithm.structured.strategy.refine._
import com.cra.figaro.algorithm.structured.strategy.solve._

val alg =
  new StructuredProbQueryAlgorithm(universe, elements:_*)
  with AnytimeStructuredProbQuery {

  // Lazy val to avoid uninitialized value bug
  // Alternatively one could make this a def
  override lazy val rangingStrategy: RangingStrategy = ...

  override def refiningStrategy(): RefiningStrategy = ...

  override def solvingStrategy(): SolvingStrategy = ...
}
```

As one can see above, relevant classes for building custom SFI algorithms are found in the com.cra.figaro.algorithm.structured package. One can find the base algorithms in the algorithm package, the various strategies in the strategy package, and the included solvers in the `solver` package. The relevant algorithms, strategies, and solvers are all additionally documented in the Scaladoc.

Additional examples showing creation and use of custom SFI algorithms are provided in the `lazystructured` package in the Figaro examples.

5.9.5 *Function memoization*

SFI caches sub–models across Chains by looking at the particular function and parent value that produced the sub–model. However, some care needs to be taken to use this optimization correctly. Consider the following example:

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

val e1 = Select(0.1 -> 1, 0.2 -> 2, 0.3 -> 3, 0.4 -> 4)
val e2 = Select(0.4 -> 2, 0.2 -> 3, 0.4 -> 4)
def expand(i: Int): Element[Int] = Select(0.5 -> i, 0.5 ->
i + 1)
val e3 = Chain(e1, expand)
val e4 = Chain(e2, expand)
```

Notice how `e3` and `e4` use the same function to produce a child Element. This type of reuse is an opportunity to use SFI's automatic memoization. Unfortunately, SFI will not recognize this equivalence because SFI memoizes on the basis of reference equality of functions. Indeed, equality testing of functions in Scala currently only returns true if the functions point to the same reference in memory. This is the case even if the two functions appear to compute the same values on inputs. Here, turning the `expand` method into a function actually creates a new instance of an `Int => Element[Int]` both times it is called. To take advantage of SFI memoization, one must instead define the model like this:

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

val e1 = Select(0.1 -> 1, 0.2 -> 2, 0.3 -> 3, 0.4 -> 4)
val e2 = Select(0.4 -> 2, 0.2 -> 3, 0.4 -> 4)
val expand = (i: Int) => Select(0.5 -> i, 0.5 -> i + 1)
val e3 = Chain(e1, expand)
val e4 = Chain(e2, expand)
```

This ensures that both Chains use the same function.

5.9.6 *Component collections*

Each SFI algorithm is associated with a central data structure: a `ComponentCollection`. A component collection maintains a set of problem components (SFI wrappers for elements) and a graph of sub–problems (SFI sub–models). Figaro offers several implementations of component collections, some of which are applicable to different classes of models. This collection is specified in the constructor to an SFI algorithm, but Figaro also includes default constructors that assign a collection type based on the choice between SFI and LSFI.

The basic implementation is `ComponentCollection`, which applies only to non-recursive models. This non-recursive collection is provided as the default for regular SFI, and guarantees runtime linear in the number of expansions. However, it fails on models containing a Chain function that calls itself recursively. An example model that contains such a recursive call is shown below:

This example can be found in InfiniteExpectation.scala

```
// Prior on the probability of termination at each
// iteration
val prob = Beta(1, 5)

// A simple recursive element that uses Chain function
// memoization
def recursiveElement(): Element[Int] = Chain(Flip(prob),
   recursiveFunction)
val recursiveFunction: Boolean => Element[Int] =
   (b: Boolean) => {
      if(b) Constant(0)
      else recursiveElement().map(i => (i + 1) % 3)
   }
val elem = recursiveElement()
```

For these recursive models, Figaro offers several other implementations. Each implementation makes a trade-off between the performance of the refining and solving steps. The most basic collection (and the default for LSFI) is `IncrementingCollection`. This collection offers the least refining overhead among the collections that support recursive models, but sometimes results in a large solving overhead. In contrast, `SelectiveIncrementingCollection` and `MinimalIncrementingCollection` try to reuse sub–problems more intelligently, potentially improving performance by reducing the number of sub–problems that need to be solved. However, doing so requires performing a search on the problem graph each time a sub–problem is expanded, potentially incurring overall quadratic refining costs. The difference between these last two implementations is that `MinimalIncrementingCollection` incurs this cost every time a particular function is expanded from one problem into a sub–problem, which can happen multiple times at different depths. However, `SelectiveIncrementingCollection` only incurs this cost the *first* time such an expansion is made; each successive identical expansion at greater depths will take constant time. Nevertheless, `MinimalIncrementingCollection` potentially reuses more subproblems, so once again there is a trade-off.

In general, it is difficult to predict which of these three recursive collections will yield the best performance on a particular model. The trade-offs are also not strictly related to runtime: the trade-offs that benefit solving time often also yield stronger bounds on a particular query. For this reason, it may be best to experiment with all three collection types to determine which one produces the fastest convergence to the desired bounds. More details regarding the differences in implementation for these recursive collections are available in the Scaladoc. These classes are located in the `com.cra.figaro.algorithm.structured` package.

Component collections offer one additional optimization. In many cases, when creating factors for sub–problems of a Chain, it is possible to combine the solutions to sub–problems into a single factor representing a conditional probability distribution. This reduces memory and inference costs when applicable. One can enable this optimization by setting the boolean flag `useSingleChainFactor` to `true` in the component collection associated with an algorithm. This flag is disabled by default, however all of Figaro's basic built-in SFI algorithms enable it.

5.9.7 *Atomic rangers*

SFI algorithms operate on discrete factor graphs. Generating such a factor graph requires computing a finite range for each element involved. For elements with finite support, this range is often just the entire support; for elements with infinite support, this is typically a finite approximation. Ranging currently proceeds by generating ranges for atomic elements (i.e. elements that use no other elements in their generation), then propagating ranges deterministically through other elements, including Chain, Apply, and Dist.

When using LSFI only, ranges are allowed to contain a special value called $*$ (pronounced "star"). $*$ essentially represents missing or uncomputed values. LSFI automatically uses $*$ to quantify the effects of unexpanded sub–models in a partially expanded model. Regular SFI will throw an exception if a query is made to an element with range containing $*$.

SFI algorithms require each atomic element to define a distribution over a finite range. If using LSFI, this range can contain $*$. Ranging for a particular atomic element of type `T` is handled by an implementation of `AtomicRanger[T]`. `AtomicRanger` specifies two methods: `discretize()` and `fullyRefinable()`. The `discretize()` method returns a distribution over extended values of type `T` (i.e. either a regular value or $*$). The distribution is represented as a map from extended values to probabilities, with the property that the probabilities must sum to 1. This method is called once each time a refining strategy generates the range of an atomic element. Thus, if one wants to incrementally refine an infinite element (e.g. using an anytime SFI algorithm that refines once per iteration), it is up to the `AtomicRanger` to return an incrementally better discretization each time the method is called. The `fullyRefinable()` method returns a boolean indicating if `discretize()` returns a complete distribution that cannot be refined further. Generally, this is only true for rangers over finite-support elements that enumerate the entire distribution.

Figaro includes several implementations of `AtomicRanger`.

The `FiniteRanger` class returns a complete distribution for Figaro's built-in atomic elements with finite range (including Flip, Select, and Binomial). Otherwise, it returns a distribution that assigns probability 1 to $*$.

The `SamplingRanger` class approximates a distribution by sampling, returning a uniform distribution over the sampled values. It takes additional samples with each call to `discretize()`, and returns a distribution containing all previously sampled values. The number of samples to take per iteration is configurable.

The `CountingRanger` class is only applicable to atomic distributions of type `Int` that have support over all integers greater than or equal to `lower` for some integer `lower`. Among Figaro's built-in elements, this applies only to Geometric and Poisson distributions. This ranger incrementally counts from `lower` to a variable (exclusive) upper bound that increases by `valuesPerIteration` each time the `discretize()` method is called. Probabilities are assigned to these counted values according to the actual underlying distribution, and all remaining probability mass is placed on $*$. Because `CountingRanger` can return ranges with $*$, it is only applicable to LSFI.

One can also define custom rangers. For example, the following defines an incremental "binning" ranger for a Beta distribution (recall that the support of a Beta distribution is the interval $[0, 1]$):

This example can be found in InfiniteExpectation.scala

```
class BetaBinningRanger(beta: AtomicBeta,
valuesPerIteration: Int) extends AtomicRanger(beta) {
  val dist = new BetaDistribution(beta.aValue, beta.bValue)

  // This is not fully refinable because it acts on an
  // element that has infinite range
  override def fullyRefinable() = false

  // Total number of values (bins) to use at the current
  // iteration
  var totalValues: Int = 0

  override def discretize() = {
    // Take additional values each iteration
    totalValues += valuesPerIteration
    // Make equally-spaced bins, each weighted by the
    // prior probability of sampling from that bin
    val probs = for(i <- 0 until totalValues) yield {
      // Lower and upper bounds on this bin
      val lower = i.toDouble / totalValues
      val upper = (i+1).toDouble / totalValues
      // Assign the value to the middle of the bin
      val mid = (lower + upper) / 2
      Regular(mid) -> dist.probability(lower, upper)
    }
    // This returns a discrete distribution that
    // approximates the Beta distribution given
    probs.toMap[Extended[Double], Double]
  }
}
```

Notice the call to the constructor `Regular(mid)`. `Regular[T]` is one of the subtypes of `Extended[T]`, the other being `Star` (i.e. $*$).

5.9.8 *Ranging strategies*

Atomic rangers are assigned to atomic elements by a ranging strategy. Specifically, the `RangingStrategy` class defines a single abstract method with signature `apply[T](atomic: Atomic[T]): AtomicRanger[T]`. Each SFI algorithm is provided with a single `RangingStrategy`.

Figaro provides two ranging strategies. The default ranging strategy uses `FiniteRanger` for built-in finite atomic elements, and otherwise uses `SamplingRanger`. Such a strategy is obtained by calling `RangingStrategy.default(numValues)`, where `numValues` is the number of additional samples to take at each iteration.

The default *lazy* strategy (intended for LSFI) works the same way, except that it applies the `CountingRanger` to Geometric and Poisson elements. Such a strategy is obtained by calling `RangingStrategy.defaultLazy(numValues)`, where `numValues` is the number of additional values (from Geometric or Poisson elements) or samples (from other atomic elements) to take at each iteration.

Of course, one can also define a custom ranging strategy. For example, the following defines a ranging strategy that applies `FiniteRanger` to Flip elements, and applies the binning strategy defined in the previous section to Beta elements (for a model containing only Flip and Beta elements):

This example can be found in InfiniteExpectation.scala

```
new RangingStrategy {
  override def apply[T](atomic: Atomic[T]):
    AtomicRanger[T] = {
    atomic match {
      case flip: AtomicFlip =>
        new FiniteRanger(flip)
      case beta: AtomicBeta =>
        new BetaBinningRanger(beta, valuesPerIteration)
    }
  }
}
```

This definition assigns an atomic ranger to each element by its type. However, one could also assign atomic rangers by reference, e.g. to specify a unique ranger for each atomic element in the model.

5.9.9 *Refining strategies*

Recall that refining strategies have two jobs: to generate ranges for elements relevant to the query, and to expand sub–problems through Chains. Refining strategies operate on a component collection, and define a single `execute()` method that performs the refinement.

The most generally applicable refining strategy is `BacktrackingStrategy`, which performs both expansion and ranging. It takes as input a list of problem components and a maximum depth of expansion. This depth parameter enables lazy partial expansion: starting from a depth of 0 at the top-level, the strategy performs a depth-first search through elements in the model. The strategy increments its own depth each time it enters a sub–problem. It is called a "backtracking" strategy because it uses backtracking to propagate updates when an element is visited multiple times at different depths. This backtracking process can cause the algorithm to take superlinear time on some models.

An alternative strategy with similar applications is `RecursionDepthStrategy`. It performs the same kind of depth-first expansion, but uses a notion of depth that is instead defined by the associated component collection. The exact definition depends on the type of component collection (and indeed, it is precisely this definition that makes the distinction between each of the component collection types). The main benefit of `RecursionDepthStrategy` is that it does not need to perform backtracking, often resulting in substantially better performance.

There is just one type of failure mode for `RecursionDepthStrategy` that prevents it from being as general as `BacktrackingStrategy`. If using a `SelectiveIncrementingCollection` or `MinimalIncrementingCollection` (5.9.6) on a recursive model that does *not* use function memoization (5.9.5), then `RecursionDepthStrategy` will fail to terminate. An example of such a recursive model that fails to use function memoization is given below:

```
def recursiveElement(): Element[Int] = Chain(Flip(0.5),
  (b: Boolean) => {
    if(b) Constant(0)
    else recursiveElement().map(_ + 1)
  })
val elem = recursiveElement()
```

An appropriate transformation into a use of function memoization might look like this:

```
val recursiveFunction: Boolean => Element[Int] =
  (b: Boolean) => {
    if(b) Constant(0)
    else recursiveElement().map(_ + 1)
  })
def recursiveElement(): Element[Int] = Chain(Flip(0.5),
  recursiveFunction)
val elem = recursiveElement()
```

The reason for this failure mode is that these two collection types only increment the depth at *recursive* calls by a reference-equal Chain function. In contrast, `IncrementingCollection` always increments the depth blindly at expansions. For this reason, `RecursionDepthStrategy` always works with `IncrementingCollection`. This combination of a component collection and refining strategy is in fact the default for built-in LSFI algorithms.

Another included strategy is `FlatStrategy`. This takes a set of existing components in the component collection and updates the range for each one. This will not update the ranges of any other components; thus a `FlatStrategy` will not enter sub–problems to refine them. Closely related is `TopDownStrategy` which also takes a fixed set of components to update. Instead of updating only these components, it also propagates updates to all dependent components in the component collection, ensuring consistency across generated ranges. These strategies can be useful for refining infinite-ranged elements in models that have fully expanded their sub–models.

5.9.10 *Solving strategies*

The most basic solving strategy is `ConstantStrategy`, which solves in a uniform manner. It takes three arguments: an inference problem to solve, a factor solver, and a raising criteria. The factor solver can be one of Figaro's built in solvers, which include Variable Elimination, Belief Propagation, and Gibbs sampling. The raising criteria is a decision function to decide whether to solve or "raise" a sub–problem into a higher-level problem without solving. Solving a sub–problem enables reusing the solution elsewhere in the model, and often induces a better elimination order. Nevertheless, raising is available for the occasional cases where it is known to perform better. Figaro includes several built-in raising criteria: `structuredRaising` (always solves), flatRaising (always raises), and raiseIfGlobal (raises if a sub–problem uses *globals*, i.e. elements declared in a higher-level problem).

More generally, one can subclass RaisingStrategy for additional flexibility. Here, one can override the eliminate method, which takes a set of factors and eliminates any number of variables to produce a solution. This is how the hybrid strategies VEBPGibbsStrategy, VEBPStrategy, and VEGibbsStrategy are implemented. Additionally, one can override the recurse method of RaisingStrategy, which specifies the strategy to use to solve a sub–problem. This allows one to, for example, use one solver for the top–level problem, and a different solver for each nested problem. See the RaisingStrategy Scaladoc for more information regarding the specifications of these methods and how to override them.

It is important to note that if using LSFI, one must use a *non-normalizing* solver for the top-level problem to produce correct bounds on a query. Currently, the only non-normalizing solver implemented in Figaro is Variable Elimination.

5.10 probability of evidence algorithms

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.

There is also an anytime version of the probability of evidence algorithm forward sampling algorithm. 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).

Additionally, the probability of evidence can be computed using algorithms like importance sampling, belief propagation and particle filtering. Examples are shown for the simple model below:

```
val universe = Universe.createNew()
val u = Uniform(0.0,0.2,0.4,0.6,0.8,1.0)("u", universe)
val condition = (d: Double) => d < 0.4
val evidence = List(NamedEvidence("u",
Condition(condition)))
```

This model defines a uniform with six outcomes, and a condition having two satisfying outcomes.

With belief propagation, we compute the probability of evidence with and without the condition and divide.

```
val bp1 = BeliefPropagation(10, u)(universe)
bp1.start
bp1.stop
val withoutCondition = bp1.computeEvidence()
bp1.kill()

universe.assertEvidence(evidence)
bp2.start
bp2.stop
val withCondition = bp2.computeEvidence()
bp2.kill()
val e1 = withConditionwithoutCondition
```

For importance sampling, the evidence is provided as an argument to the `computeProbEvidence` method.

```
val importance = Importance(100000, u)
importance.start()
importance.stop()
val e2 = importance.probabilityOfEvidence(evidence)
```

In particle filtering, the probability of evidence at the current time step can be computed using `probEvidence()`.

```
val pf = ParticleFilter(universe, t, 10000)
pf.start()
val condition = (d: Double) => d < 0.4
val evidence = List(NamedEvidence("u",
Condition(condition)))
pf.advanceTime(evidence)
val e3 = pf.probEvidence()
pf.stop()
pf.kill()
e3
```

The result of each computation is approximately .333. `println(e1 + " " + e2 + " " + e3)` yields:

```
0.3333333333333333 0.3338586042039474 0.3269
```

5.11 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: (1) Variable elimination, and (2) 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(3.0))
```

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.

5.12 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: String)
```

```
abstract class Sample(val name: String) {
   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)("isTheRightSource", universe)
  val rightSourceDistance = Normal(0.0,
1.0)("rightSourceDistance", universe)
  val wrongSourceDistance = Uniform(0.0,
10.0)("wrongSourceDistance", universe)
  val distance =
  If(isTheRightSource, rightSourceDistance,
wrongSourceDistance)("distance", 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:

```
val evidence1 = NamedEvidence("distance", Condition((d:
Double) => d > 0.5 && d < 0.6))
val evidence2 = NamedEvidence("distance", Condition((d:
```

```
Double) => d > 1.5 && d < 1.6))
val evidence3 = NamedEvidence("distance", Condition((d:
Double) => d > 2.5 && d < 2.6))
val evidence4 = NamedEvidence("distance", Condition((d:
Double) => d > 0.5 && d < 0.6))
val ue1 = (pair1.universe, List(evidence1))
val ue2 = (pair2.universe, List(evidence2))
val ue3 = (pair3.universe, List(evidence3))
val ue4 = (pair4.universe, List(evidence4))
def peAlg(universe: Universe, evidence:
List[NamedEvidence[_]]) = () =>
ProbEvidenceSampler.computeProbEvidence(100000,
evidence)(universe)
val alg = VariableElimination(List(ue1, ue2, ue3, ue4),
peAlg _, sample1.fromSource)
```

5.13 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.

5.14 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()`.

6 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 transition model may also optionally take a static universe as an input that represents static, non–time dependent variables.

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.

There are two dynamic reasoning algorithms available in Figaro: (1) Particle filtering, and (2) factored frontier.

6.0.1 *Particle filtering*

To create the particle filter, use:

```
val pf = ParticleFilter(initial, trans, numParticles)
```

where `initial` 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.currentProbability("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.

There is also a parallel version of particle filtering that uses Scala's built in parallel collections. The interface to use parallel particle filtering is similar to the original algorithm with a few exceptions. First, this version uses a model generator, which is a function that produces a universe (particle filtering is run in parallel on separate but identical universes). Second, the user must indicate the number of threads to use.

6.0.2 *Factored frontier*

Factored frontier is a dynamic reasoning algorithm that uses factors instead of sampling. To create an instance of factored frontier, use:

```
val ff = FactoredFrontier(static, initial, transition,
numIterations)
```

where `static` is a universe with the static variables, `initial` is the initial universe, `transition` is the transition model (a function from `(Universe, Universe) => Universe`), and `numIterations` is the number of internal BP iterations that should be performed at each time step (as BP is used internally in the algorithm). Optionally, an anytime version of BP can be used instead, where the `numIterations` is replaced by `stepTimeMillis`, a long that indicates how long to run BP at each time step.

The same restrictions and behaviors for continuous variables that apply to BP also apply to factored frontier. The interfaces for advancing time, applying evidence, and querying the model are the same for factored frontier and particle filtering. As in particle filtering, only named elements are propagated to the next time step for building the model and querying.

7 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.

7.1 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.

7.2 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.

7.3 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 `com.cra.figaro.algorithm.decision` package.

7.4 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.

7.4.1 *Finite 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.

7.4.2 *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.

7.5 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 = MultiDecisionImportance(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.

8 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.

8.1 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: (1) `Beta`, and (2) `Dirichlet`.

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 = Beta(1,1)
```

The element `fairness` is the parameter we will use to model the bias of our coin. The creation of the parameter is no different than 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 `Beta` 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.

This example is found in FairCoin.Scala

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.

```
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 zip model.trials foreach {
   (datum: (Char, Flip)) => if (datum._1 == 'H')
   datum._2.observe(true) else datum._2.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 `Beta`. 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.

8.2 expectation maximization

A learning algorithm can be used to determine the maximum a posteriori estimate for parameter elements. Parameter elements have a `MAPValue` which is set when the parameter is used as a target in a learning algorithm. Presently, Figaro provides one learning algorithm, expectation maximization, which uses existing Figaro algorithms to estimate 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 MAP 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. We can also choose an inference algorithm for estimating the sufficient statistics of the target parameters. Currently, Metropolis Hastings, importance sampling, variable elimination or belief propagation can be used for this purpose. Figaro's Generalized EM algorithm is used in the following way:

```
val learningAlgorithm = EMwithMH(fairness)
learningAlgorithm.start
learningAlgorithm.kill

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

The line `val learningAlgorithm = EMwithMH(fairness)` creates an EM algorithm which uses Metropolis Hastings to estimate sufficient statistics. We could also have used `EMwithBP(fairness)` or `EMwithImportance(fairness)`.

After the algorithm has finished running, we can create an element learned from the parameter by using `Flip(fairness.MAPValue)`. The element `coin` is a `Flip`, where the probability of producing true is determined from the data we observed above.

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 = Flip(fairness.MAPValue)
val t2 = Flip(fairness.MAPValue)
val equal = t1 === t2
```

We can then use an algorithm like 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
```

8.3 PARAMETER COLLECTIONS

In the previous sections, parameter learning was discussed using a Beta parameter. The Beta parameters were supplied individually to the learning algorithm, and the MAP value for each parameter was retrieved individually. For more complicated models, it is often useful to define a model structure with parameters which can be learned, and then use the values of the learned parameters in the same structure. The `ModelParameters` pattern is a simple way of accomplishing this. By using `ParameterCollection`, the model structure only needs to be defined once. Parameters are added to the collection in a similar fashion to `Element` collections. We specify the parameter name and add it to the collection when we create the parameters.

This section will also 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:

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

This example is found in FairDice.Scala

Next, we create a set of model parameters representing the parameters of the fair dice model. We create a parameter representing the fairness of each die and add it to the collection of model parameters.

```
val params = ModelParameters()
val fairness1 = Dirichlet(2.0, 2.0, 2.0, 2.0, 2.0,
2.0)("fairness1", params)
val fairness2 = Dirichlet(2.0, 2.0, 2.0, 2.0, 2.0,
2.0)("fairness1", params)
val fairness3 = Dirichlet(2.0, 2.0, 2.0, 2.0, 2.0,
2.0)("fairness1", params)
```

Each die is initially assumed to be fair. For convenience, the data which we will learn the parameters from is represented in Scala 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.

The next step is to define a class representing the model structure.

```
class DiceModel(val parameters: ParameterCollection, val
data: Seq[(Int, Int, Int)], val outcomes: List[Int])
```

This defines a class which accepts a `ParameterCollection`, a set of data, and a list of outcomes as its arguments. As we will see, the values which are retrieved from the `ParameterCollection` depend on whether we are working with the prior or posterior parameters. To model the outcome of the sum, we can use an `Apply` element with a function which sums the outcome of its arguments. We place the following loop inside the `DiceModel` class:

```
val sum = (i1: Int, i2: Int) => i1 + i2
val trials = for (datum <- data) yield {
   val die1 = Select(parameters.get("fairness" + datum._1),
outcomes: _*)
   val die2 = Select(parameters.get("fairness" + datum._2),
outcomes: _*)
   Apply(die1, die2, sum)
}
```

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. We retrieve the parameters by using the `get` method from the input `ParameterCollection`.

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.

We can now create an instance of the `DiceModel` class, using the prior parameters from the parameter collection.

```
val model = new DiceModel(params.priorParameters, data,
outcomes)
```

To apply evidence to the model, we can write another loop over the contents of the data and the trials defined inside the model class.

```
for ((datum,trial) <- data zip model.trials) {
   trial.observe(datum._3) }
```

Just as in the fair coin example, we create an expectation maximization algorithm. This time, instead of passing the parameters in a list or sequence, we can simply use the collection of parameters as an input argument.

```
val numberOfBPIterations = 10
val numberOfEMIterations = 10
val algorithm = EMWithBP(numberOfEMIterations,
numberOfBPIterations, params)
algorithm.start
algorithm.stop
val d1 = Select(params.posteriorParameters.get("fairness1"),
outcomes:_*)
val d2 = Select(params.posteriorParameters.get("fairness2"),
outcomes:_*)
val d3 = Select(params.posteriorParameters.get("fairness3"),
outcomes:_*)
```

The code block above will create `Select` elements using to the MAP value of the learned parameters. We retrieve the MAP value of the parameters by using the `posteriorParameters.get` method of our parameter collection. If we wanted to create another set of 50 trials using the learned parameter values, we could simply use:

```
val model = new DiceModel(params.posteriorParameters, data,
outcomes)
```

Note that for a `Select`, a list of outcomes must be supplied as an argument to along with their corresponding probabilities. 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:
   0.906250000442371 -> 1
   0.0 -> 2
   0.0 -> 3
   0.0 -> 4
   0.09374999955762903 -> 5
   0.0 -> 6

The probabilities of seeing each side of d_2 are:
   0.0 -> 1
```

```
  0.9999999996067813 -> 2
  0.0 -> 3
  0.0 -> 4
  0.0 -> 5
  3.9321864899990694E-10 -> 6

The probabilities of seeing each side of d_3 are:
  0.0 -> 1
  0.0 -> 2
  0.0 -> 3
  0.0 -> 4
  0.0 -> 5
  1.0 -> 6
```

9

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, name)
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`.

10 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 factored algorithms. 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.

10.1 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 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.

Scala contains many transformations on sequences besides `toList`

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.

10.2 CREATING AN COMPOUND CLASS WITH INHERITANCE

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

10.2.1 *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. Chains themselves perform no caching, but the cacheable or non–cacheable status of a chain tells an algorithm whether it is allowed to cache chain results. 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.

10.2.2 *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.

10.3 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.

10.4 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 Apply[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.

10.5 MAKING A CLASS USABLE BY FACTORED ALGORITHMS

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(depth: Int): ValueSet[Int] =
ValueSet.withoutStar((for { i <- 0 to n } yield i).toSet)
```

The `makeValues` method returns a set of values in a set called a ValueSet. A ValueSet is a set of over a type but also includes a special value call * (star) that is used for lazy inference. In addition, makeValues function requires a depth value that is also needed for lazy inference. For a binomial, the makeValues function 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`.

Generally, factored algorithms like 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. However, factored algorithms will operate on any Figaro model by sampling values from an element if no ValuesMaker is defined. It is still better though to explicitly specify how to make factors for a new element. To do this, you make it extend `FactorMaker` 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 one could extend the AtomicBinomial class to extend FactorMaker and include the following code to generate factors:

```
def makeFactors(binomial: AtomicBinomial):
List[Factor[Double]] = {
   val binVar = Variable(binomial)
   val factor = new Factor[Double](Array(binVar))
   for { (value, index) <- binVar.range.zipWithIndex } {
   factor.set(List(index), binomial.density(value.value))
}
   List(factor)
}
```

(Note that the AtomicBinomial class is included in Figaro and thus its factor creation is defined in the language, not in the element itself.)

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(binomial) 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(List(index), binomial.density(value.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 list 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 Factory.scala. For atomic elements, the process should usually be similarly simple to that for binomials.

10.6 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 AtomicBeta(name: Name[Double], a: Double, b: Double,
collection: ElementCollection) Element[Double](name,
collection) with Atomic[Double] with DoubleParameter 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. 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 learnedAlpha = a
var learnedBeta = b
```

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

```
def MAPValue: Double = {
   if (learnedAlpha + learnedBeta == 2) 0.5
   else (learnedAlpha - 1) / ( learnedAlpha + learnedBeta -
2)
}
```

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)
   learnedAlpha = sufficientStatistics(0) + a
   learnedBeta = sufficientStatistics(1) + b
}
```

Having created a parameter, we may now create an element which uses it. Compound elements which use `Parameters` 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 `Beta` in the following way:

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

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

```
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: Element[Double])(implicit name:
Name[Boolean], collection: ElementCollection) =
   new ParameterizedFlip(name,
prob.asInstanceOf[AtomicBeta], collection)
```

To learn the MAP value of the parameter, Figaro provides an implementation of the expectation maximization algorithm. During the expectation step, the algorithm retrieves the distribution of the element according to the current value of the parameter, then converts the distribution to sufficient statistics. This conversion needs to be defined inside the parameter, using the method distributionToStatistics. It accepts as an argument a Scala stream consisting of pairs of double values (probabilities) and possible outcomes of the element. The implementation for a Flip is shown below.

```
def distributionToStatistics(distribution: Stream[(Double,
Boolean)]): Seq[Double] = {
  val distList = distribution.toList
  val trueProb =
    distList.find(_._2) match {
      case Some((prob,_) => prob
      case None => 0.0
    }
  val falseProb =
    distList.find(!_._2) match {
      case Some((prob,_)) => prob
      case None => 0.0
    }
  List(trueProb, falseProb)
}
```

10.7 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:

$$\frac{P(r^1 \text{ -> } r^0)P(r^1)}{P(r^0 \text{ -> } r^1)P(^0)}$$

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 $P(r^0$ -> $r^1)$ 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 ($P(r^1$ -> $r^0)/P(r^0$ -> $r^1)$) and the model probability ratio ($P(r^1)/P(r^0)$). These ratios are 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.

11 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.

11.1 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.

11.2 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.

11.2.1 *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.

11.2.2 *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:

```
LazyValues(universe).expandAll(universe.activeElements)
```

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:

```
Factory.makeFactorsForElement(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`.

11.3 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.

11.3.1 *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.

11.4 learning algorithms

Learning algorithms (using sufficient statistics) require that the sufficient statistics of a parameter are modified with the learned result. The EM algorithm is defined in the base class GeneralizedEM, which accepts any `ProbQueryAlgorithm` inference algorithm as an argument. Figaro provides implementations of expectation maximization using algorithms like importance sampling, Metropolis-Hastings and belief propagation for the estimates the sufficient statistics of the target parameters using an inference algorithm like importance sampling or belief propagation. To complete the expectation step, the expected sufficient statistics factors must be retrieved for all of the parameters. This can be accomplished by calling the method `distribution` from the inference algorithm, then using `distributionToStatistics`, which is implemented by parameterized elements.

`GeneralizedEM` does not directly 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.

11.5 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 concreteValues[T](element: Element[T], depth: Int,
numArgSamples: Int, numTotalSamples: Int): ValueSet[T] =
element match {
   case c: Constant [_] => withoutStar(Set(c.constant))
   case f: Flip => withoutStar(Set(true, false))
   ...
   case v: ValuesMaker[_] => v.makeValues(depth)
   case _ => withStar(Set())
}
```

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, we can't compute the values and we just make them *, so the rest of the computation can proceed.

11.6 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

```
trait MPEAlgorithm extends Algorithm {
val universe: Universe
   /**
   * Particular implementations of algorithm must provide
the following method.
   */
   def mostLikelyValue[T](target: Element[T]): T
}
```

An `MPEAlgorithm` contains 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.

12 EXPERIMENTAL FEATURES

Figaro contains a number of experimental features and algorithms that generally work but have not been validated to the level necessary for official support. These methods are not supported by the Figaro team and may be changed at any time, though we anticipate officially moving them into the main Figaro package once the Figaro team is confident of their correctness.

12.1 MARGINAL-MAP

Marginal maximum a posteriori (Marginal–MAP) is an algorithm that combines marginal inference with MLE reasoning. In Marginal–MAP, the user specifies a set of variables to maximize after all non–query variables are marginalized away. Oftentimes Marginal-MAP is preferable to using an MLE algorithm since there are usually variables in the model that the user does not care about maximizing, and hence they can be marginalized out. Figaro contains several experimental Marginal–MAP algorithms, such as a BP–, sampling–, and SFI–based algorithms.

To use a BP–based Marginal–MAP algorithm, you create an instance similar to many other algorithms:

```
import com.cra.figaro.language._
import
com.cra.figaro.experimental.marginalmap.MarginalMAPBeliefPropagation

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 = MarginalMAPBeliefPropagation(20, e1)
alg.start()
println(alg.mostLikelyValue(e1))
```

Please see the Scaladoc in the experimental package for more details on running Marginal–MAP algorithms.

12.2 COLLAPSED GIBBS SAMPLING

Collapsed Gibbs sampling is a variant of Gibbs sampling where some variables are marginalized out (or collapsed) before sampling from the conditional probability distribution for a variable. The method of collapsing and which variables to collapse can vary widely. In the experimental package, Figaro has a collapsed Gibbs sampler with several strategies for collapsing. Using collapsed Gibbs sampling with the default collapser is similar to many other algorithms:

```
import
com.cra.figaro.experimental.collapsedgibbs.CollapsedGibbs

val alg = CollapsedGibbs(100, element)
alg.start()
```

The default collapser uses a heuristic based on the Hellinger distance to collapse the model to a smaller size.

12.3 NORMAL PROPOSALS FOR METROPOLIS–HASTINGS

The experimental package also contains a set of univariate, continuous elements that are customized for better Metropolis–Hastings sampling performance. These elements are exactly like their counterparts in the `com.cra.figaro.library.atomic.continuous` package except that their `nextRandomness` function has been redefined to propose a new value for the element that is normally distributed from the current value. In general, this results in better Metropolis–Hastings performance since the sampler can more efficiently move through the state space.

13 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, feature requests, 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.

Thanks for reading, and enjoy!