Creating and Manipulating Probabilistic Programs with Figaro

Avi Pfeffer
Charles River Analytics
apfeffer@cra.com

Abstract

Probabilistic programming languages (PPLs) allow probabilistic models to be represented using the power of programming languages and general-purpose reasoning algorithms to be applied to new applications. This paper presents an approach to probabilistic programming in which the program is represented as a set of data structures using a rich library of model elements. While the data structures alone have the representational power of previous programming languages, the ability to create and manipulate them from within an ordinary program provides a number of benefits, including clean and natural ways to incorporate potentials and undirected models, represent models with inter-related objects, creating dynamic probabilistic programs, reason about multiple models simultaneously, and compile other languages. Our approach is implemented in Figaro, which has been released open-source.

1. GENERAL FORMATTING INSTRUCTIONS

Probabilistic models are ever growing in richness and diversity. Creating models for specific applications presents both representation and reasoning challenges. Probabilistic programming languages (PPLs) can address these challenges by allowing models to be represented using the power of programming languages and by providing general-purpose reasoning algorithms that automatically work in new applications. This is an extremely powerful idea, as it gives you the ability to create probabilistic models with arbitrary control flow, including recursion, as represented by a Turing-complete programming language. It also allows you to create probabilistic models over structures of arbitrary complexity, as long as these structures can be represented in a programming language.

Most existing probabilistic programming languages are based on either functional or logic programming (see Section 2). We present an alternative, object-oriented approach to probabilistic programming, captured in our probabilistic programming language Figaro. Figaro has been release open source and is available from www.cra.com/figaro.

The key idea behind Figaro that differentiates it from existing languages is that probabilistic models are represented as data structures built up out of Figaro elements. These data structures are front and center in the language. The data structures can be constructed and manipulated within a program in the host programming language in a number of ways, such as building up complex elements out of simpler elements, connecting elements to each other, or adding constraints to elements. In Figaro's case, the host language is Scala, which is a language that combines functional and object-oriented styles and compiles to the Java Virtual Machine. A side benefit of the fact that Figaro programs are objects in Scala is that Figaro elements can be created, manipulated and used from the extremely popular Java programming language.

Figaro's data structures alone provide all the expressivity of existing probabilistic programming languages like Church, in that any Church program can be easily represented as Figaro data structures. However, it is the ability to create and manipulate these structures from within an ordinary program that gives Figaro its power. This capability has numerous benefits, including:

- Providing a clean and simple way to incorporate potentials and undirected models.
- Providing a way to represent models with mutually influencing objects.
- Providing a way to handle uncertainty about the relationships between objects.

- Reasoning about multiple models simultaneously.
- Supporting creating and reasoning about dynamic probabilistic programs.
- Providing a target for compiling other probabilistic representations.

In Section 3, we describe the building blocks of Figaro programs, and show how they are combined to produce a program. Section 4 goes into details of manipulating Figaro programs and the benefits thereof. Although inference is not a major focus of this paper, it is clearly important for a probabilistic programming language, and we describe the goals of Figaro's inference engine and its approach to achieving them in Section 5. Figaro's capabilities make it possible to do things like create cyclic models that are not obvious in other languages, so we need to make sure that Figaro programs have a well-defined semantics, which we describe in Section 6. Finally, we conclude in Section 7.

2. RELATED WORK

Probabilistic programming languages have typically come in a variety of flavors. In functional probabilistic programming languages, like IBAL (Pfeffer, 2007) and Church (Goodman et al., 2008), a program represents a generative process that describes the stochastic generation of a possible world. Evaluating a program involves computing probability distributions or conditional distributions over program outputs. In IBAL, a special-purpose language was used, which was then interpreted in a host language. In Church, a dialect of Scheme is used and the program is itself interpreted in Scheme. In these languages, the approach has generally been to view the program as a given unit that is then evaluated.

In logic-based probabilistic programming languages, such as PRISM (Sato, 2008) and ProbLog (De Raedt, Kimmig, & Toivonen, 2007), an ordinary logic program is augmented with probabilistic choices; the two together constitute the probabilistic program. As with functional languages, this program is then evaluated as a unit.

We are not suggesting that Figaro is the only probabilistic programming language that provides data structures that can be manipulated. In particular, in Scheme code is data, so Church programs could in principle be manipulated. However, this aspect of probabilistic programming has not been explored in the past, and Figaro provides a particularly direct and explicit way to manipulate programs, as this capability is essential to the design of the language. In addition, the idioms and benefits of manipulating probabilistic programs have not been examined. As a result, the benefits described in the introduction have not generally been realized in the past. For example, existing probabilistic programming languages do not generally support dynamic models explicitly. In a logic-programming based language, for example, time can be treated by defining a relation to the previous time step, but this does not provide support in a natural way for filtering.

There is an additional object-oriented language, FACTORIE (McCallum et al., 2008), incidentally also written in Scala, that also has the property that elements of models are data structures. There are two key differences between FACTORIE and Figaro. First, FACTORIE's data structures represent factors in a factor graph, whereas Figaro's represent functional probabilistic programs with a process of generating values stochastically. Second, a major part of the specification of FACTORIE objects includes details of how to perform Metropolis-Hastings inference using those objects, and this capability is one of the driving forces behind the design of FACTORIE. Therefore, FACTORIE can be better viewed as an expressive language for defining Metropolis-Hastings computations over factor graphs, rather than declarative probabilistic programs that can be operated on by a variety of inference algorithms, some of which involve factor graphs.

3. FIGARO PROGRAMS

3.1 ELEMENTS

The central concept in Figaro is an *element*. Intuitively, an element defines a process that stochastically produces a value, given the values of argument elements. Elements are instances of the `Element` class. The `Element` class itself is abstract; a particular element will be an instance of a concrete subclass of `Element`. The `Element` class provides the key ingredients of what it means to be an element of a Figaro model. We describe the contents of an element by going through the key part of the definition of the `Element` class in Scala:

```
1 abstract class Element[V] {
2   type Value = V
3   type Randomness
4   def genRand(): Randomness
5   def genValue(rand: Randomness): Value
6   val value: Value
7 }
```

The notation `Element[V]` (line 1) indicates that the `Element` class takes a type parameter. This is the type of value produced by the element. Line 2 declares `Value` to be a name for this type parameter. In addition to the value, each element contains an intrinsic randomness, whose type is captured by the `Randomness` type (line 3). The specific `Randomness` type must be defined in a concrete subclass of `Element`.

The process of generating a value for the element has two stages. In the first stage, the randomness is stochastically generated using the `genRand` method defined in a concrete subclass (line 4). The second stage is a

deterministic function `genValue` (line 5) that generates the value of the element given the randomness and argument elements. The resulting value of the element is stored in the `value` field (line 6). The `genValue` method can be any purely functional Scala function. It can use the current value of related elements, but it is not allowed to have side effects (such as changing another element's value). This is a key restriction that allows us to give a coherent semantics to Figaro programs. The purpose of separating `genRand` from `genValue` is to isolate the random component of an element from the logic through which the value of the element is determined based on values of other elements. Algorithms can focus on changing the randomness of elements and observing the deterministic effects of these changes.

A Figaro program is built up out of a number of related elements, each of which belongs to an element class. Fortunately, Figaro provides a rich library of element classes, so much of the time all the programmer needs to do is stitch elements together. In case an appropriate library class is not found, it is often straightforward to create a new one, involving defining the `Randomness` and writing `genRand` and `genValue`.

3.2 DETERMINISTIC ELEMENTS

A special case of element is a deterministic element whose randomness is the trivial `Null` type, and whose `genValue` method does not depend on the randomness. All a programmer needs to do to create a deterministic element class is describe how the value is generated from the values of the arguments.

An element can be deterministic even if related elements on which it depends are not. Although we may have uncertainty over its value, due to uncertainty over the values of its arguments, it is deterministic given particular values of the arguments.

3.3 ATOMIC ELEMENTS

If the `genValue` method of an element does not depend on the values of any arguments, the element is called *atomic*. There is a single deterministic atomic class of element, which is the `Constant` class that always produces the same value. For example, `Constant(8)` is an `Element[Int]` that always produces 8. This is atomic because 8 is a fixed integer, not an element.

Other atomic classes implement a probability distribution over values. An example is `Flip`, which takes a Double argument and produces true with probability equal to the argument. To illustrate the way concrete subclasses of `Element` are defined, we show the definition of `Flip`:

```
1 class Flip(prob: Double)
2       extends Atomic[Boolean] {
3   type Randomness = Double
4   def genRand() = random.nextDouble()
5   def genValue(rand: Randomness) =
6       rand < prob
7 }
```

`Flip` takes a Double argument named `prob` (line 1) and an instance of `Flip` is an atomic element that produces Boolean values (line 2). Line 3 declares `Flip`'s randomness to be a Double, and line 4 says that generating a value for the randomness is accomplished simply by getting the next random Double, which is uniformly distributed between 0 and 1. `genValue` (lines 5-6) is then a deterministic function that returns true if the randomness is less than the input argument, which happens with probability equal to the argument.

Continuous atomic elements are also available, such as `Uniform(x,y)`. A wide variety of atomic elements are provided as library classes.

3.4 CHAIN

A *chain* is the glue that puts things together. Figaro's language is based on the *probability monad*. Monads are structures used in functional programming to lift computation from a space of values to a space of concepts over those values. The probability monad lifts values to elements that generate those values. Monads are defined by two functions: *unit*, and *bind*. The unit function takes a value and produces the monad over that value. Figaro's unit is defined by the `Constant` class. Figaro implements the bind function by the `Chain` class, which represents chaining of probabilistic processes. A chain takes two arguments: *p*, which is an `Element[V]`, and *f*, which is a function that maps a value of type `V` to an `Element[W]`. It defines the process in which it first gets the value *v* of *p* and then gets the value *w* of *f*(*v*). For example, `Chain(Flip(0.7), Uniform(0.0, 1.0), Constant(0.5))` represents the process that flips a coin with weight 0.7. If it produces true, it generates an element uniformly between 0 and 1, otherwise it generates 0.5 with probability 1.

The essential definition of `Chain` is

```
abstract class Chain[T,U](
  p: Element[T],
  f: T => Element[U]
) extends Deterministic[U] {
  def genValue() = f(p.value).value
}
```

Given the machinery we have developed for deterministic elements, the only thing we need to define is `genValue`, and its definition is very simple: we get the value of *p*, apply *f* to it, and get the value of the result. Note that the process of applying *f* happens as part of `genValue`, so *f* must be purely functional.

3.5 SYNTACTIC SUGAR

All Figaro elements can be implemented using atomic elements and chain. However, it is useful to provide other element classes, both for convenience of building models

and because particular common types of element can be optimized. Figaro provides a variety of such element classes. For example, `If(test, thn, els)` is the deterministic element whose generation involves getting the value of `test`, and if it is true getting the value of `thn`, otherwise getting the value of `els`.

The `Dist` element class provides a very general way to describe discrete probabilistic dependency. A `Dist` element *D* takes a list of clauses as its argument, where each clause consists of a probability element and an outcome element. The randomness of *D* is a Double uniformly distributed between 0 and 1. The meaning of *D* is as follows: First, get the values of the probability elements. When normalized, these define a multinomial over the outcome elements. Use the randomness to select an outcome element from the multinomial. Finally, get the value of this outcome element; it becomes the value of *D*.

An important element class is `Apply`, which takes an `Element[V]` and a function *f* from `V` to `W`, and produces an `Element[W]`. Its meaning is to take the value of its element argument and apply *f* to it. Apply serves to lift functions from the space of values (`V` → `W`) to the space of elements (`Element[V]` → `Element[W]`). Other versions of `Apply` lift functions of more than one argument, or of a list of arguments. Likewise, there are extensions of `Chain` to more than one argument or a list of arguments. Standard functions can be defined using `Apply`. For example, `===` tests for equality between the values of two elements, while `Duo(e1,e2)` forms the element whose value is the pair of the values of its inputs.

Another useful element class is `Inject`, which takes a list of elements and produces an element of lists. For example, the element `Inject(List(Dist(0.2 -> 2, 0.8 -> 3), Constant(1)))` produces the list (2,1) with probability 0.2 and (3,1) with probability 0.8.

To illustrate the power of combining these concepts in an example, let us create a `MakeList` element class that creates a list of a stochastic number of items, each item generated according to a particular process. `MakeList` will take two arguments: `numItems`, which is an `Element[Int]`, and `itemMaker`, which is a function of 0 arguments that produces an item according to the process. For example, `MakeList(Geometric(0.9), () => Flip(0.5))` will generate a list of a geometrically distributed number of independent coin tosses. `MakeList` can be defined directly using `Chain` and `Inject` as follows.

```
Chain(numItems, (n: Int) =>
  Inject(List.fill(n)(itemMaker())))
```

In this code, `List.fill` is Scala's library function for creating a list with a given number of elements. It has two argument lists: the first is the number of items, and the second is the code to create each item.

3.6 VARIABLES AND PROGRAMS

A probabilistic program in Figaro consists of a set of elements that can depend on each other in arbitrary ways. Unlike IBAL and Church, Figaro does not provide its own variable binding mechanism. Instead, it piggybacks on top of Scala's variables. If the same Scala object is used multiple times, it must have the same value every time, while two distinct objects with the same definition may have different values. For example, in the program

```
val x = Flip(0.9)
val y = x === x
```

`y` will always have the value true. In contrast, the element `Flip(0.9) === Flip(0.9)` will have value true with probability 0.9 * 0.9 + 0.1 * 0.1 = 0.82.

Using the ability to create atomic elements of a wide variety, chain together elements, and apply arbitrary functions, which can produce any data type, Figaro can implement any control flow in its programs. Since `Chain` applies a function to determine its result element, recursive models can easily be created. For example:

```
def f(n: Int): Element[Int] =
  Chain(Flip(0.9), Constant(n), f(n+1))
val i = f(0)
```

In this example, `def` is Scala's keyword for a function definition. The function `f` takes an integer argument and returns an element over integers that is itself defined using `f`. By changing the example slightly, we can also see how Figaro programs can represent probability distributions over arbitrary data structures:

```
abstract class Tree
case class Branch(left: Tree, right: Tree)
  extends Tree
case class Leaf(n: Int) extends Tree
def f(n: Int): Element[Tree] =
  Chain(Flip(0.9), Constant(Leaf(n)),
    Apply(f(n+1), f(n+2),
     (l: Tree, r: Tree) => Branch(l,r)))
```

In this code, we define a `Tree` class with two subclasses, `Branch` and `Leaf`. The recursive case now creates two subtrees, which are combined into a branch using `Apply`.

4. PROGRAMS AS DATA STRUCTURES

4.1 CONDITIONS AND CONSTRAINTS

Since Figaro is embedded in Scala, a Scala program can be written to generate a Figaro program and manipulate its elements. The Scala program can be an arbitrary computation and need not be purely functional. In particular, it can have side effects. This ability to apply side effects to a probabilistic program in a controlled, programmatic way is one of the main reasons representing programs as data structures is useful.

An immediate application of this idea is in Figaro's method for stipulating *conditions* and *constraints* on the

values of elements. In Figaro, a condition represents a hard condition that must be satisfied by the value of an element, while a constraint represents a soft constraint, i.e., a potential on the value. In other words, constraints represent a rescaling of the probability distribution over values of elements described by the elements' generative process, which is normalized across all possible values.

- Conditions and constraints can be defined on elements over any data structure. In addition, we can create conditions or potentials over multiple elements by combining them in tuple elements.
- Conditions and constraints are arbitrary Scala functions and have the resulting expressivity. For example, we can have an element over trees and define a constraint that is logarithmic in its size.
- We can accumulate conditions and constraints on an element without having to know which already exist. For example, we can separately specify that a tree's size be even and that it have at least five elements.
- Conditions and constraints can be applied systematically to a set of elements. We illustrate this with the following example, based on the Smokers and Friends example from Markov Logic Networks (Domingos & Richardson, 2007).

```
1 class Person { val smk = Flip(0.6) }
2 val x, y, z = new Person
3 val friends = List((x, y), (y, z))
4 def c(pair: (Boolean, Boolean)) =
5   if (pair._1 == pair._2) 3.0; else 1.0
6 for { (p1,p2) ← friends } {
7   Duo(p1.smk,p2.smk).constraint = c
8 }
```

Line 1 defines the `Person` class with the `smk` attribute. Line 2 defines three instances of this class, while line 3 indicates which people are friends. Lines 4-5 encode the constraint that friends are three times more likely to have the same smoking habits than non-friends. Finally, lines 6-8 apply the constraint to the pair of `smk` attributes of each pair of friends. In line 7, a new `Duo` element is created to hold the constraint. In this way, we constrain all the pairs of friends as specified by a Scala list, which could easily be passed as an argument to the program.

4.2 MUTUALLY INFLUENCING OBJECTS

A principle of functional probabilistic programs is that there is a flow of generation through the program. Object-oriented models, on the other hand, often encapsulate a number of variables within an object. There can be a conflict between the two styles when two objects mutually influence each other, e.g., variable *X* in object 1 influences variable *Y* in object 2, which in turn influences variable *Z* in object 1. To capture this pattern, a functional probabilistic programming language would ordinarily have to expose variables *X*, *Y*, and *Z* outside the objects to which they belong, thereby breaking encapsulation.

An example representation affected by this issue is probabilistic relational models (PRMs) (Koller & Pfeffer, 1998), which are a representation used for statistical relational learning. PRMs are object-oriented models in which probability models are associated with classes. A class probability model defines a number of attributes of the class and their probabilistic behavior. Instances derive their probabilistic model from the class to which they belong, and attributes of an instance may depend probabilistically on attributes of both the same instance and related instances. Instances can be inter-connected in arbitrary ways, including mutually influencing ways.

Scala's object-oriented nature is a good match for PRMs; in addition, Figaro solves the mutual dependence problem using side effects. Consider an Actors and Movies example. Movies have a quality attribute, while actors have skill and fame attributes. The quality of a movie depends on the skill of its actors, while the fame of an actor depends on the quality of the movies in which he or she appears. The following Scala program creates a Figaro model for this situation while maintaining encapsulation of the attributes within their respective classes. For simplicity, let these attributes be Booleans, and let `fraction` be a function that takes a list of Booleans and returns the fraction that are true. We can represent this situation as follows:

```
1 class Actor {
2   var movies: List[Movie] = List()
3   val skill = Flip(0.2)
4   val fame =
5     Chain(movies map quality,
6           (l: List[Boolean]) =>
7             Flip(fraction(l))
8 }
9 class Movie {
10  var actors: List[Actor] = List()
11  val quality =
12    Chain(actors map skill,
13          (l: List[Boolean]) =>
14            Flip(fraction(l))
15}
16def connect(actor: Actor, movie: Movie) {
17  actor.movies ::= movie
18  movie.actors ::= actor
19}
20val chaplin = new Actor
21val modernTimes = new Movie
22connect(chaplin, modernTimes)
```

Lines 1-8 define the `Actor` class. Line 2 defines the `movies` variable representing the movies the actor played in. The Scala keyword `var` defines a mutable variable whose value can be changed. `movies` starts out as an empty list and is added to later. Lines 4-7 define a model for the fame of an actor that is defined to be a `Flip` whose probability of being true is the fraction of the actor's movies that are of high quality. The definition of `Movie` is similar to that of `Actor`. The key feature of this example is `connect` (lines 16-19), which is a function

that connects an actor to a movie by prepending the movie to the list of movies of the actor (line 17) and vice versa (line 18). A PRM for a specific situation is defined by creating instances of the classes (lines 20-21) and connecting them (line 22). At the time that inference is applied to the PRM, the correct movies and actors are used with the correct dependencies.

4.3 NAMING AND REFERENCE UNCERTAINTY

It is important to point out that in the above example, the object-orientation is only present in the Scala program, not in the constructed Figaro program. This makes it challenging to handle situations in which we want to refer to a specific element, but we do not know which element it is. This is called *reference uncertainty* in PRMs. For example, suppose an actor has a favorite movie, which is uniformly distributed from amongst the actor's movies. We could create a `favorite` element with the appropriate distribution. Suppose now that we want to make the actor's happiness depend on the quality of the actor's favorite movie. We cannot do this using the current design. We cannot refer to `favorite.quality` because `favorite` is an element, not a movie.

Figaro solves this problem in two steps. First, it gives elements optional *names* by which they can be referred. Second, it provides the ability to lift the object-oriented nature of Scala programs into Figaro programs by creating *element collections*. An element collection, as its name describes, is simply a collection of elements in one container. The key point is that we can make an element collection the value of an element. For example, we can make `Movie` an element collection, and make the value of `favorite` be a `Movie`. If we then give `quality` and `favorite` appropriate names, we should be able to refer to the quality of the favorite movie of an `Actor` using something like `"favorite.quality"`.

Some extra machinery is needed to make this work, but it is not too difficult. Although there is uncertainty over to which specific element `favorite.quality` refers, `favorite` has a specific value in any possible world. We can create a deterministic element representing the value of `favorite.quality`, and write its `genValue` function to first look at the value of `favorite` which is a particular movie, and then get the value of the `quality` of that specific movie. This is implemented by the `get` method of an element collection; we use `actor.get ("favorite.quality")` to create the appropriate element. The relevant part of the example is as follows:

```
1 class Actor extends ElementCollection {
2   var movies: List[Movie] = List()
3   val favorite =
4     Uniform(movies)("favorite", this)
5   val happy =
6     If(get("favorite.quality"),
7        Flip(0.9),
8        Flip(0.3))
9 }
10 class Movie extends ElementCollection {
11  var actors: List[Actor] = List()
12  val quality =
13    Chain(actors map skill,
14          (l: List[Boolean]) =>
15            Flip(fraction(l))
16       ("quality", this)
17}
```

Lines 1 and 10 define instances of `Actor` and `Movie` to be element collections. Now, every element has a name and belongs to an element collection, but there are default arguments for these so they do not usually need to be specified. In previous examples, we used the default name and element collection. In this example, we supply the name and element collection for `favorite` and `quality`. In both cases, the element collection is defined to be "`this`", which means the Scala object in which these elements are contained (i.e., the instances of `Actor` and `Movie`, respectively.) This means that within the actor class we can say `get("favorite.quality")`, which means get the element named `favorite` defined in the actor, get its value, which is a movie, and then get the element named `quality` in that movie.

4.4 UNIVERSES

Figaro defines the notion of *universe*, which is a set of elements that are operated on by an inference algorithm. A universe is an element collection; there is always a default universe (which can be changed). The default universe is also the default element collection, as described in the previous section. It is possible to work with multiple universes at the same time.

Besides the general usefulness of being able to create, store, and manipulate multiple probabilistic programs at once, working with multiple universes has a specific use in allowing different algorithms to be used for different parts of a program, as represented by different universes. One example of this principle is Rao-Blackwellization (Doucet et al., 2000), in which sampling is used for one part of a model while a dependent part is marginalized out exactly. This can easily be represented by putting the variables to be marginalized in a separate universe.

An example of the opposite flavor is implemented in Figaro. If a model is generally tractable and amenable to variable elimination, but it involves some difficult evidential parts, it is possible to put the evidential parts in dependent universes and use a sampling algorithm for those parts, while using variable elimination for the core model. The sampling is conditioned on possible values of variables in the main universe that influence the dependent universe directly. The result of sampling is the probability of the evidence in the dependent universe, and is used to condition the elements in the main universe on which the dependent universe depends. If there are multiple dependent universes, the algorithm can sample

them independently and appropriately condition elements in the main universe.

4.5 DYNAMIC MODELS

Most previous probabilistic programming languages do not support dynamic reasoning explicitly. Thanks to the concept of a universe, this is quite easy in Figaro. A dynamic model in Figaro consists of an initial model, which is a universe, and a transition model, which is a function that takes a universe (representing the previous time step) and returns a universe (representing the current time step). Beginning with the initial model and iterating the transition model results in a stream of universes, one for every time step.

Because a universe is an element collection, we can refer to elements within a universe by their name. This is important, because when the transition model is written we don't know the specific elements on which it is operating, only the universe itself, and we have to get the elements through the universe. Referring to elements by name is also useful for setting evidence by applying conditions and constraints to elements. The following code shows a simple example of a dynamic model:

```
1 val u1 = Universe.createNew()
2 val f = Flip(0.2)("f", u1)
3 def trans(previousUniverse: Universe):
4  Universe = {
5   val u2 = Universe.createNew()
6   val b: Element[Boolean] =
7     previousUniverse.get("f")
8   val i =
9     If(b, Flip(0.8), Flip(0.3))
10      ("f", u2)
11  u2
12}
```

Line 1 creates a new universe for the initial state, and line 2 adds an element named "f" to it. Lines 3 to 12 define the transition function, which takes the previous universe as input and returns a universe. First, it creates the new universe. Lines 6-7 get at the element named "f". For the dynamic model to work, it is necessary that an element named "f" appear both in the initial universe and the result of the transition model. In the transition model, this element is created in lines 8-10. Finally, the transition function returns the new universe in line 11.

4.6 FIGARO AS A COMPILATION TARGET

Since the goal of probabilistic programming is to make it easier to construct new probabilistic models and to reason about them, it is natural to use a probabilistic programming language as a compilation target for another probabilistic modeling language. Figaro's data-structures approach to probabilistic programming makes it particularly well-suited to this task. It is not necessary to generate a probabilistic program from whole cloth; instead, a piecemeal approach where a program is constructed bit by bit can be used.

Scala's lazy evaluation mechanism is helpful here. In Scala, a field can be defined using `lazy val`. Instead of its value being computed as soon as it is encountered, it is not computed until it is needed. The benefit of this in a compiler is that we can define fields for all the program structures to be lazy, begin construction of the program at any point, and automatically make sure that all the necessary structures are created.

We are currently taking this approach to developing a compiler for PRMs into Figaro. The compiler first works in "PRM world," resolving classes and inheritance and generating all the instances that appear in the model. It then lazily defines all the needed elements, corresponding to all the attributes of all the instances.

In a different project, we have used Figaro as a compilation target for probabilistic models for intelligence analysis. These models, which could be applied to a variety of situations, depended on the specific features of each situation. So, for example, a situation involving a certain kind of aircraft would have elements corresponding to that aircraft. While each situation model was simply a Bayesian network, there was much sharing of knowledge, both between situations (the same aircraft appearing in multiple situations) and within situations (different aircraft using similar logic). Figaro's ability to represent model elements as data structures made constructing the network for each situation easy. In particular, Figaro elements for each of the objects that can appear were stored in a hash table that could be accessed by name of the object, making it simple to assemble the right elements according to the configuration of a situation. As this example shows, Figaro's approach makes it easy to embed Figaro models in a program that does more than probabilistic reasoning.

5. INFERENCE

Inference in probabilistic programming languages is challenging because of the ability the languages provide to create models of significant complexity. Previous languages have often taken the approach of optimizing a particular inference algorithm, with the design of the Figaro being guided by the needs of the algorithm. In reality, many algorithms and variations thereof can be appropriate in different circumstances.

In Figaro, the main goal behind the inference engine is *extensibility*. Ideally, Figaro would include as wide a range of algorithms as possible, and we plan to continue building its repertoire. For this to be accomplished, it should be relatively easy to add new algorithms. Figaro provides a rich class hierarchy of algorithms. Algorithms are distinguished along a number of lines, such as the goal of the algorithm (e.g. computing the marginal probabilities of query variables or computing the probability of evidence—of course, an algorithm can have multiple goals); whether the algorithm is one-shot or

anytime; and the approach taken by the algorithm, such as factor-based or sampling.

To create a new algorithm, you inherit from the appropriate Scala interfaces, and much of the mechanics of the algorithm is automatically taken care of. For example, if you create an anytime algorithm, Figaro automatically takes care of spawning a thread and listening for queries. To create a new MCMC algorithm, you only have to describe how to generate the next state from the previous state; everything else is handled automatically, and you get both a one-shot and anytime algorithm for free. Similarly, mechanics for constructing factor graphs, creating elimination orders, and eliminating variables using appropriate sum and product operations is provided and can be used by any algorithm.

Currently, Figaro's algorithm library includes variable elimination for both conditional probability and most probable explanation, importance sampling, Metropolis-Hastings (MH), dependent universe reasoning, reasoning with abstraction and discretization, and particle filtering for dynamic models. MH in particular can be tricky to get right for a probabilistic program. While we have not solved the problem of automatically generating a good proposal distribution, Figaro's MH algorithm is designed to help the modeler by providing a rich language for specifying proposal distributions and tools for debugging the MH process. One tool we have found to be particularly useful allows you to set any state and observe the probabilities of properties of subsequent states reached by a single MH step. This can help detect when certain important state transitions are very unlikely. Another tool shows you exactly which elements are being proposed during the MH process. As far as we know, there is not a large literature on debugging MH (there are statistical tests for correctness (Geweke, 2004)). R (http://www.r-project.org/) also includes tests that verify the correctness of MH. However, an MH process is often correct but extremely slow, and these tests do not help make your MH algorithm correct and fast. We hope that our tools can make a practical difference.

6. SEMANTICS

Our insistence that Figaro models be side-effect free is crucial to the semantics. While side-effects are allowed in the Scala program that creates the Figaro model, Figaro's semantics is defined at the point of inference. When an inference algorithm is invoked, the Figaro model must already have been constructed. The process of constructing this model can be an arbitrary complex execution in Scala, but it must terminate to reach the point of inference. Therefore, at the time of inference, the model consists of a finite set of elements.

One issue that complicates the semantics is that it is possible in Scala to create cyclic models in which Figaro elements mutually influence each other. This is a novel feature of Figaro, and it may be useful, but we are not claiming it as a key feature because we are unsure about its benefits. Nevertheless, it is important to make sure the semantics cleanly handles such models. Since Figaro is designed to support many inference algorithms, we want to develop a *declarative* semantics that is not dependent on a particular inference algorithm. Since the entire Figaro library can be defined using atomic elements and chains, we focus on those.

Our semantics is specified by associating a function $P^E(v \mid \mathbf{Q})$ with each element E where v is a possible value of the element and $\mathbf{Q}$ is the set of free variables in E**.** After constructing the function, we will show that, under the right assumptions, it naturally defines a conditional probability distribution (CPD) over values of E given $\mathbf{Q}$.

Atomic elements have no free variables and the function for an atomic element A is given by

$$P^A(v) = \sum_{r:A.\text{genValue}(r)=v} P(A.\text{genRand}() = r) \quad (1)$$

Thus, atomic elements naturally correspond to unconditional distributions with the expected semantics.

6.1 CPDS FOR CHAINS

Consider a chain C with parent R and function f. Generating the value of the chain involves applying f to R's value to produce an element E, and then getting E's value. Applying f might involve the generation of a number of other elements $S_1,\ldots,S_n$, in that order, which might be used by E. These elements $S_1,\ldots,S_n$ are bound in C. These elements and E might also use free variables $\mathbf{Q}$. Marginalizing over temporary variables, P^C is given by

$$P^C(v \mid R, \mathbf{Q}) = \sum_{s_1} P^{S_1}(s_1 \mid \mathbf{Q}) \sum_{s_2} P^{S_2}(s_2 \mid s_1, \mathbf{Q}) \ldots P^E(v \mid s_1, \ldots, s_n, \mathbf{Q}) \quad (2)$$

Elements created by applying the function f of a chain are called *temporary*. This is in contrast with elements that are created before inference is begun, which are called *permanent*. As we stipulated earlier, f can have no side effects. As a result of this restriction, any elements created by f cannot have conditions or constraints, and there can be no cycles in the above equations for temporary elements. Therefore, expanding C through Equation (2) leads to a sequence of recursive definitions that sometimes terminate in atomic elements and contain no cycles. A complete expansion can be fully characterized by the outcomes of its atomic elements.

We stipulate that the process of expansions terminate with probability 1 for the program to be well defined. This is a standard assumption that has been made by all previous functional PPLs. This assumption implies that with probability 1, the resulting value v of C will be finite. Now consider the sample space $\mathcal{F}$ of all possible expansions of C, equipped with the σ-algebra generated

by finite sub-expansions. For fixed R and $\mathbf{Q}$, the above equations define a probability measure over $\mathcal{F}$. Note that R and $\mathbf{Q}$ must be permanent elements, since temporary elements are not in scope outside the chain that created them. Therefore the number of elements in $\mathbf{Q}$ must be finite. We can therefore interpret $P^C(v \mid R, \mathbf{Q})$ as a conditional probability distribution over the value of C given the parent R and free variables $\mathbf{Q}$ that appear inside elements generated by the function.

6.2 DEFINING THE JOINT DISTRIBUTION

So far, we have shown that all permanent elementsare associated with a CPD. Unlike temporary elements, the number of permanent elements is guaranteed to be finite. However, also unlike temporary elements, permanent elements can have conditions and constraints, and there can be cycles among the permanent elements. To start with, we will ignore the issue of cycles and deal with conditions and constraints. Let the permanent elements be $E_1,\ldots,E_N$. Let the parents of E_i be $\mathbf{Q}_i$. $P^{E_i}(v_i \mid \mathbf{Q}_i)$ is the local CPD of E_i as defined previously, not taking into account conditions and constraints. We define

$$P_0(E_1 = v_1,\ldots,E_N = v_N) = \prod_{i=1}^{N} P^{E_i}(v_i \mid \mathbf{Q}_i) \quad (3)$$

We stipulate that the conditions be satisfiable by some assignment of values to the elements that has non-zero probability under P_0 and that the constraints be positive everywhere. We now define

$$P_1(E_i = v_i \mid \mathbf{Q}_i) = P^{E_i}(v_i \mid \mathbf{Q}_i)\,\mathbb{1}_{\phi_i}(v_i)\,\psi_i(v_i) \quad (4)$$

where $\mathbb{1}_{\phi_i}(v_i)$ denotes the indicator function of the conditions on E_i and $\psi_i(v_i)$ denotes the product of values of the constraints. In the acyclic case, the probability of a joint assignment to all the elements is given by

$$P(E_1 = v_1,\ldots,E_N = v_N) \propto \prod_{i=1}^{N} P_1(E_i = v_i \mid \mathbf{Q}_i) \quad (5)$$

Now, once we introduce cycles, we have a cyclic model defined by CPDs, which is just like a dependency network, so the obvious approach is to base our semantics on dependency networks. Heckerman et al. (2000) defined the semantics of dependency networks through a pseudo-Gibbs sampling method that visits each node in turn and samples a value for the node using the local conditional distribution of the node. We could use such a pseudo-Gibbs method that samples each element E_i from $P^{E_i}(v_i \mid \mathbf{Q}_i)$. Heckerman et al. showed that under the assumption that the distributions are positive, the pseudo-Gibbs sampler converges to a unique stationary distribution, and used that distribution to define the meaning of the network. Unfortunately, the distribution to which this process converges depends on the order in which nodes are sampled. Furthermore, in the stationary distribution, the local CPD of a node given its parents is not necessarily equal to the local CPD that was used to sample the node. In addition, in our case the CPDs for deterministic elements are generally not positive. Finally, we seek to develop a declarative semantics for Figaro that is not tied to a particular inference algorithm.

Our approach, instead, is to define the semantics of such a model as a Markov network in which the CPD of each node specifies a potential over that node. We use Equation (4) to define the potential at each node, and use Equation (5) to define the semantics of the model as a whole, with no change from the acyclic case.

There are subtleties, which are handled well by our semantics. First, it is possible using cycles to define contradictory models. For example, consider an element X whose parent is X and is defined to be the negation of its parent. In such a case, all assignments will have zero probability. This case is ruled out by the requirement that there exists a satisfying assignment that has positive probability under P_0. Second, it is possible for a particular setting of the randomness of all elements to have more than one consistent assignment of values. For example, suppose X is its own parent and is the same as its parent. In this case, the semantics is exactly as in Equation (5). Because P_1 is uniquely defined using Equation (4), Equation (5) always has a unique solution. In this example, all values of X would have uniform probability.

We note that although we have defined semantics for a general class of models, a particular inference algorithm may only work for a subclass. This is consistent with Figaro's approach of employing a library of algorithms, each of which is suitable for particular programs.

7. CONCLUSION AND FUTURE WORK

We have described the probabilistic programming language Figaro, whose most fundamental contribution is to introduce the data-structures approach to probabilistic programming. We have shown a number of benefits of this approach, and also described Figaro's extensible inference framework and presented a formal semantics. As we have described, Figaro is already being used in real applications and its benefits are becoming apparent.

The main design of the language is complete, and Figaro has been released open source. However, there are a number of areas in which we are seeking to improve Figaro, most importantly in developing algorithms to learn parameters and structure of programs. We are also seeking to introduce a wider variety of algorithms and to identify design patterns for probabilistic programming.

ACKNOWLEDGEMENTS

This work was supported by DARPA contract W31P4Q-11-C-0083, with thanks to Dr. Robert Kohout and Dr. Anthony Falcone. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

De Raedt, L., Kimmig, A., & Toivonen, H. (2007). ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *Proceedings of International Joint Conference on Artificial Intelligence*.

Domingos, P. & Richardson, M. (2007). Markov Logic: A Unifying Framework for Statistical Relational Learning. In L. Getoor & B. Taskar (Eds.), *Introduction to Statistical Relational Learning* (pp. 339-371). Cambridge, MA: MIT Press.

Doucet, A., de Freitas, N., Murphy, K., & Russell, S. (2000). Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks. In *Proceedings of Uncertainty in Artificial Intelligence*.

Geweke, J. (2004). Getting It Right. *Journal of the American Statistical Association*, 99(467), 799-804.

Goodman, N. D., Mansinghka, V. K., Roy, D., Bonawitz, K., & Tenenbaum, J. B. (2008). Church: a Language for Generative Models. In *Proceedings of Uncertainty in Artificial Intelligence*.

Koller, D. & Pfeffer, A. (1998). Probabilistic Frame-Based Systems. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*.

McCallum, A., Rohanemanesh, K., Wick, M., Schultz, K., & Singh, S. (2008). FACTORIE: Efficient Probabilistic Programming for Relational Factor Graphs Via Imperative Declarations of Structure, Inference and Learning. In *Proceedings of NIPS Workshop on Probabilistic Programming*.

Pfeffer, A. (2007). The Design and Implementation of IBAL: A General-Purpose Probabilistic Language. In L. Getoor & B. Taskar (Eds.), *Statistical Relational Learning*. MIT Press.

Sato, T. (2008). A Glimpse of Symbolic-Statistical Modeling in PRISM. *Journal of Intelligent Information Systems*.