Figaro: An Object-Oriented Probabilistic Programming Language

Avi Pfeffer
Charles River Analytics
apfeffer@cra.com

Abstract

We introduce an object-oriented paradigm for probabilistic programming, embodied in the Figaro language. Models in Figaro are objects, and may have properties such as conditions, constraints and relationships to other objects. Figaro model classes are created by inheriting functionality from existing classes. Figaro provides a modular, compositional Metropolis-Hastings algorithm, and gives the modeler optional control over it. Figaro is tightly integrated into the Scala programming language, giving it access to a rich type system, control abstractions and libraries.

1 Introduction

Probabilistic models are ever growing in richness and diversity. Models may be hierarchical, relational, spatio-temporal, recursive and infinite, among others. Developing the representation, inference and learning algorithms for a new model is a significant task. Probabilistic programming has the potential to make this task much easier, by making it easy to represent rich, complex probabilistic models using the full power of programming languages, including rich data structures, control mechanisms and abstraction. Inference and learning algorithms come for free, at least to an extent. Most importantly, a probabilistic programming language provides the modeler with the language with which to think of and formulate new models.

A number of flavors of probabilistic programming languages have been developed. Some languages are logic-based, such as PRISM [10], BLOG [6] and Markov logic [1]. Others are based on functional programming, such as IBAL [8] and Church [2]. FACTORIE [5] is a recent language that uses an imperative style. This paper introduces a new, object-oriented paradigm for probabilistic programming, embodied in the Figaro probabilistic programming language. The object-oriented approach has several advantages.

First, models in Figaro are built by composing models into something more complex, much in the same way that expressions in IBAL or Church are combined into more complex expressions. A model in Figaro is more than just an expression, however. It is an object, and as such, it can have properties. These properties include conditions, making it possible to specify evidence from observations in a clear, modular way. They may also include constraints, which give Figaro the ability to express undirected models as well as purely generative ones. Properties also make it easy to express situations in which objects are related to each other in various ways, which may be hard to express in a purely functional language.

Second, one of the key features of object-orientation is inheritance. In Figaro, more specific model classes can be derived from more basic classes, inheriting the implementation of the basic classes as far as possible. As we will describe, two important model classes in Figaro are `Apply`, which allows a function to be applied to the result of a model to obtain a new model, and `Chain`, which allows models to be chained using conditional probability. Many different classes of models, including most of Figaro's standard model classes, can be derived from them.

Third, Figaro supports modular, compositional control of inference using methods of model classes. Markov chain Monte Carlo (MCMC), and in particular the Metropolis-Hastings (MH) algorithm[3], has emerged as an inference algorithm of choice for probabilistic programming, used in BLOG, Markov logic, Church and FACTORIE. Some languages, such as Church, hide all details of the inference algorithm from the programmer. Others, such as BLOG and FACTORIE, require the programmer to exert some control over the inference process by specifying a proposal distribution. Figaro provides the best of both worlds. On the one hand it provides a sophisticated MH algorithm that will be sufficient for many applications, without the modeler having to worry about inference. On the other hand, for a modeler who wants more control, it provides the means to specify a proposal method for a particular model class. The modeler specifies a *proposal scheme* for the model class, that relies on existing proposal schemes for other classes. Furthermore, the proposal scheme is a high-level specification of the proposal process. The Figaro system provides a number of methods that facilitate writing proposal schemes, and handle all the details of the MH algorithm.

Another important feature of Figaro is that it is fully integrated into the Scala programming language [7]. Scala is a language that combines functional and object-oriented styles, and is fully interoperable with Java. Figaro model classes are Scala classes, and Figaro models are Scala instances. This integration makes it possible to fully utilize the features of the host programming language, such as a very rich type system and control abstractions. It allows the use of native Scala functions within models. Figaro models can use all Scala and Java libraries. In particular, all I/O functionality is available, making it easy to interact with data. Compared to existing languages, Figaro supports higher-order functions in a manner similar to IBAL and Church, and it is possible to write purely functional programs in Figaro, so its expressive power is a superset of those languages. Perhaps the most closely related languages are FACTORIE and Infer.NET [12], which are both integrated with an object-oriented programming language (Scala in FACTORIE's case), but do not explicitly make object-orientation a central focus of the language. FACTORIE works explicitly with factor graphs and sits at a lower level of abstraction than Figaro. Infer.NET does not support general function application so is somewhat less expressive than other probabilistic programming languages.

2 Model Composition

Intuitively, a model defines a process that stochastically produces a value. Models are constructed through *model composition*. A model is an instance of a model class, which is a subclass of `Model`, the abstract superclass of all model classes. Each model has an element of inherent randomness. In addition, models take other models as arguments. The model class of a model specifies how the value of a model is produced *deterministically* given the model's randomness and the values of its arguments. Model classes are parameterized by the type of values they produce. This allows Scala's type system to ensure that models are combined in safe ways. We notate a model parameterized by the type `T` by `Model[T]`. [1]

For example, one of the basic built-in model classes, in Scala notation, is `Flip(probModel: Model[Double]) extends Model[Boolean]`. This notation means that `Flip` takes one argument named `probModel` which is a `Model[Double]`, and `Flip` itself is a `Model[Boolean]`. `Flip` has a `Double` as its inherent randomness, uniformly distributed between 0 and 1. The rule for `Flip` says that the produced value is `true` if the randomness is less than the value of `probModel`, and `false` otherwise. Another built-in model class is `If(test: Model[Boolean], thenClause: Model[T], elseClause: Model[T]) extends Model[T]`. `If` produces values of the same type as `thenClause` and `elseClause`. `If` has no inherent randomness; more precisely, its inherent randomness is null. Its rule for producing values is obvious. Other basic model classes include:

```
Constant(value: T) extends Model[T]
Pair(first: Model[T], second: Model[U]) extends Model[(T,U)]
First(pair: Model[(T,U)]) extends Model[T]
Second(pair: Model[(T,U)]) extends Model[U]
Eq(arg1: Model[T], arg2: Model[T]) extends Model[Boolean]
Dist(clauses: (Model[Double], Model[T])*) extends Model[T]
```

[1] Models are *covariant*, which means that if `T1` is a subtype of `T2`, then `Model[T1]` is a subtype of `Model[T2]`. This property enhances the flexibility of combining models.

In the latter, the `*` after the type of `clauses` indicates that `Dist` can take any number of clauses as arguments. `Dist` specifies stochastic choice over the clauses. Each clause has an (unnormalized) probability (specified by the `Model[Double]`) and a result value (specified by the `Model[T]`). The result value is chosen with probability proportional to the probability associated with the clause.

Figaro has no model class for binding a variable to a value. Instead, it piggybacks on top of the variable binding mechanism of the host Scala language. For example, one can write

```
val x = Flip(0.3)
If(x, x, Flip(0.6))
```

The meaning of this is that the Scala field `x` is defined to be a `Flip(0.3)` model. The value of the model will always be the same wherever it appears. Thus the probability that the `If` produces `true` is $0.3 * 1 + 0.7 * 0.6 = 0.72$.

2.1 Apply and Chain

Figaro models define a process that generates Scala values. It would be nice to be able to specify the application of a native Scala function to the value produced by a model, and construct a new model whose value is the result of the function application. The `Apply` model class provides this capability. More specifically, there are two `Apply` classes, one for functions of a single argument and one for functions of two arguments. These `Apply` classes are defined as follows:

```
Apply1(arg: Model[T], f: T => U) extends Model[U]
Apply2(arg1: Model[T1], arg2: Model[T2], f: (T1,T2) => U)
  extends Model[U]
```

Here `f: T => U` means that `f` is a function from type `T` to type `U`. `Apply1` also takes an optional *condition inverter* argument. The condition inverter is a function that takes an optional condition on the result of the application, and inverts it through the function to produce an optional condition on the argument. The condition inverter is useful for inference. Similarly, `Apply2` takes two condition inverters, one for each argument. Many of the basic model classes presented above are actually implemented using `Apply`, through inheritance. For example, `First` is implemented as follows:

```
class First[T,U](arg: Model[(T,U)]) extends Apply1[(T,U),T](
  arg,
  (pair: (T,U)) => pair._1,
  (copt: Option[Condition[T]]) =>
    copt match {
      case None => None
      case Some(condition) =>
        Some(PairCondition(Some(condition), None)) } )
```

The first line says that `First` is a class that takes a model over pairs `(T,U)` as argument, and inherits from `Apply1`, in which the argument is of type `(T,U)` and the result is of type `T`. The remaining lines present the arguments to `Apply1`. The first argument is just the argument to `First`, i.e. the pair whose first component is being extracted. The second argument is a function that takes a pair and returns the first component. The final argument is the condition inverter. It takes an optional condition on the result type `T`, and produces an optional condition on the pair argument. The option can have the value `None`, meaning that no value is present, or it can be `Some(condition)`. In the first case, no condition is passed to the argument. In the second case, a `PairCondition` is produced which applies `condition` to the first element of the pair and nothing to the second element. Meanwhile, `Pair` and `Eq` are implemented in terms of `Apply2`.

Another fundamental model class used in Figaro is `Chain`. `Chain` implements the probability monad, introduced by Ramsey and Pfeffer [9]. Monads are fundamental data structures in functional programming, used to thread state through functions. Monads are represented by a class `M` parameterized by a type `T`, and are defined in terms of two functions: `unit`, which takes a value of type `T` and produces an `M[T]`, and `bind`, which takes an `M[T]` and a function `T => M[U]`, and produces an `M[U]`. `unit` is trivial in Figaro: it is the function that turns a value `v` into the model `Constant(v)`. As Ramsey and Pfeffer showed, `bind` corresponds to chaining using conditional

probability. We can think of the first argument of `bind` as a probability distribution over values of type `T`, and the second argument as a function from values of type `T` to distributions over type `U`, which is nothing more than a conditional distribution over `U` given `T`. The result of chaining these two arguments is a distribution over `U`. This chaining is implemented in the `Chain` model class. Like `Apply`, `Chain` takes an optional condition inverter argument. As we will see, the implementation of chain requires careful thought. The payoff to implementing chaining once and for all is large. The basic model classes `If` and `Dist` are both implemented by inheriting from `Chain`.

3 Models as Objects

Thus far, Figaro models are much like expressions is IBAL or Church. However, because they are objects, they can have properties. One kind of property a model can have is a *condition*. A condition can be any function from the type of values of a model to a `Boolean`. The most common kind of condition is simply an observation of a value. For example, one can modify the previous program to

```
val x = Flip(0.3)
val y = If(x, x, Flip(0.6))
y.condition(true)
```

The probability that `x` is `true` can now be determined to be $0.3 * 1/(0.3 * 1 + 0.7 * 0.6) \approx 0.42$.

Another property a model can have is a *constraint*, which is a function from the type of values of the model to a `Double`. A constraint is like a potential in an undirected model. A model may have multiple constraints. Constraints make it possible for Figaro to represent undirected models such as Markov logic networks. For example, the smokers and friends example from Markov logic (simplified for brevity) can be represented as follows:

```
class Person { val smokes = Flip(0.5) }
val alice, bob, clara = new Person
alice.smokes.condition(true)
val friends = List((alice, bob), (bob, clara))
def friendsConstraint(pair: (Boolean, Boolean)): Double =
  if (pair._1 == pair._2) 3.0; else 1.0
for {(p1,p2) <- friends} {
  Pair(p1.smokes, p2.smokes).constrain(friendsConstraint)
}
```

The first line defines a `Person` class with a `smokes` field which is defined to be a `Flip(0.5)` model. The second line defines three instances of `Person`. They are all different instances, so they all have their own unique `smokes` field. The third line conditions Alice's `smokes` model to have value `true`. The fourth line defines `friends` to be a list of pairs of people. This is a standard Scala `List`, not a special-purpose Figaro construct. It could easily have been read from a file. The fifth and sixth lines define the constraint, `Double`. which says that all else being equal, friends are three times as likely to have the same smoking habit than different. The last three lines go through the pairs of friends. For each pair of friends, a `Pair` model consisting of the `smokes` model of each person is created and constrained.

We can give semantics to models, conditions and constraints as follows. A *configuration* is an assignment to all the inherent randomness elements of all models in a program. Given the randomness elements, the values of all the models are deterministic. A configuration satisfies the conditions if the value of every model satisfies the condition (if any) of that model. Configurations that do not satisfy the conditions have probability zero. For those that do satisfy the conditions, the probability is proportional to the product of probabilities of the randomness elements and, for every model, the product of the values of the constraints on that model given the value of the model.

This ability to represent both generative and non-generative aspects within the same model is a powerful combination. We illustrate with a simple example. A number of firms are bidding for a contract. Some firms are efficient, while others are not, which affects their bid. Any of the firms may be selected, but the winning bid is likely to be low. This situation can be expressed in Figaro by

```
class Firm {
  val efficient = Flip(0.3)
  val bid = If(efficient, Uniform(0.0,10.0), Uniform(5.0,20.0))
}
val firms = Array.fromFunction(i => new Firm)(20)
def clause(firm: Firm) = (Constant(1.0), Constant(firm))
val winner = Dist(firms map clause)
val winningBid = Chain(winner, (f: Firm) => f.bid)
winningBid.constrain((bid: Double) => 20.0 - bid)
```

The first four lines define a `Firm` class with a Boolean `efficient` field and a continuous (`Double`) `bid` field. The fifth line defines `firms` to be an array of 20 instances of class `Firm`. The sixth line defines a function that takes a firm and produces a clause for the `Dist` expression in the seventh line. `firms map clause` means that we apply the function `clause` to each member of firms to produce an array of clauses to pass into `Dist`. The `winner` is defined to be the result of the `Dist`, and the `winningBid` is defined to be the `bid` field of the winner. The winning bid is then constrained by a constraint that says that lower bids are more likely.

Despite its simplicity, this example is hard to capture cleanly in other probabilistic programming languages. For one thing, it requires continuous variables, which many existing probabilistic programming languages do not allow. Markov logic has been recently extended to hybrid domains [11], but it does not allow general expressions in the constraint weights, so it cannot represent this constraint. In addition, the generative languages have difficulty with the constraint.

In addition to conditions and constraints, Figaro models can have more general properties. One use of this feature is to define relationships between different model objects. This allows Figaro to express probabilistic relational models [4] in full generality. While IBAL and Church are able to represent a large subset of PRMs, they have difficulty representing situations in which different objects are mutually dependent on each other. For example, we may have a `Battalion` class and a `Battery` class that depend on each other as follows:

```
class Battalion {
  val mission = Dist((0.3, attack), (0.7, defend))
  val underAttack = Flip(0.2)
  var batteries: List[Battery] = Nil
  lazy val battStatus = ListModel(batteries map (b => b.statusOk))
  def chooseNextMission(status: List[Boolean]) =
    if (status contains false) attack; else mission
  lazy val nextMission = Chain(battStatus, chooseNextMission)
}
class Battery(inBattalion: Battalion) {
  inBattalion.batteries = this :: inBattalion.batteries
  val hit = If(inBattalion.underAttack, Flip(0.6), Flip(0.01))
  val statusOk = If(hit, Flip(0.2), Flip(0.9))
}
val battalion1 = new Battalion
val battery1 = new Battery(battalion1)
val battery2 = new Battery(battalion1)
battalion1.nextMission
```

The third line of the definition of `Battalion` defines the `batteries` variable. This is a mutable variable with state that can be changed. The next line defines a list model containing the status of the batteries. It is important that it be lazy; if it were not, it would be evaluated at the time of definition, with an empty battery list. This way, its evaluation is delayed until it is used, which is after batteries have been added to the list. The following three lines define a function for choosing the next mission, and the final line defines the nextMission model, which is also lazy. The first line of the definition of `Battery` makes sure that when a battery is constructed, it is added to the `batteries` list of the model that contains it. After defining the classes, we create a battalion containing two batteries, and then get the `nextMission` model of the battalion. At this point, the lazy fields are evaluated, and `nextMission` correctly depends on the two batteries.

4 Model Extension

Figaro uses a Metropolis-Hastings (MH) algorithm for inference. Each iteration has two steps: (1) propose the randomness elements of some models, and (2) update the values of the remaining models. The values need to be updated because the remaining models may share models with those whose randomness has been changed, so their values have changed. However, as an optimization, we keep track of which models have had their randomness changed, and only update models that use them. FACTORIE uses a similar technique of keeping track of changed lists.

The algorithm is modular. For each model, the algorithm goes through a sequence of steps, which may include proposing the randomness of the model, proposing its arguments, or updating its arguments. These steps are controlled by a *proposal scheme*. The Figaro implementation takes a proposal scheme and turns it into a proposal algorithm, keeping track of the MH probabilities, the constraints, the models whose randomness has changed, the models that have been updated in any way (a superset of the changed models), and whether the values of models satisfy their conditions. The Figaro implementation is also responsible for accepting or rejecting proposals and making sure all values and randomness elements correctly reflect the most recently accepted proposal.

Proposals take an optional condition as an argument. This condition is a merging of extrinsic conditions, inferred for the model by models that use it, and the intrinsic condition associated with the model. An example of an extrinsic condition arises when there is a `true` condition on an equality. Given such a condition, and a value of 7 for the second argument, we can infer that the first argument must equal 7. This is passed to the first argument as an extrinsic condition. The condition is enforced when the randomness element of a model is proposed. Instead of proposing an arbitrary value, which will often lead to rejection, a value consistent with the condition is proposed. In addition to the steps of the algorithm, the proposal scheme controls how conditions are passed around.

Creating a new model class requires defining a proposal scheme and several other methods. This process is called *model extension*. In many cases, these methods can be inherited from other classes; for example, `First` inherits the methods from `Apply1`. Altogether, a model class needs to implement the following elements:

- `Randomness`, the type of the random element.
- `initRand`, which produces an initial value for the random element.
- `densityRand(d)`, which returns the density of the random element being equal to `d`.
- `proposeRand(condition)`, which proposes a value for the random element, given the optional condition `condition`, and returns both the new value and the ratio of the reverse proposal probability to the proposal probability.
- `computeValue`, which deterministically computes the value of the model given its randomness and the values of its arguments.
- `activeArgs`, a list of the arguments which are currently active. This list may depend on the values of some arguments. For example, for a model `If(test, thenClause, elseClause)`, if `test` is `true`, `activeArgs` includes `thenClause`, otherwise it includes `elseClause`.
- `proposalScheme(condition, alreadyUpdated)`, where `condition` is an optional condition, and `updatedState` is the set of models that have already been updated during the current MH iteration together with their values.
- `forward(updatedState)`, which runs a forward importance sampling algorithm on this model, given the values of models that have already been updated. The reason this is needed is explained in Section 5.

For many models, the inherent randomness is null. Those models can inherit from `NoRandModel`, and they do not have to implement `Randomness`, `initRand`, `densityRand` and `proposeRand`. Thus the basic methods that need to be specified are `computeValue`, `activeArgs`, `proposalScheme` and `forward`. All except `proposalScheme` are usually very simple. We illustrate a typical proposal scheme by that used for the `Select` model class, which is used in the implementation of `Dist`. `Select` takes a list of arguments representing models of probabilities, and implements the random choice of an index into the list.

```
def proposalScheme(condition, updatedState) = {
  if (random.nextDouble() < 0.5) {
    proposeThis(condition)(updatedState)
  } else {
    val choice = random.nextInt(len)
    beginProposal(updatedState) followedBy {
      assessOldDensity()
    } followedBy {
      proposeArg(probArgs(choice), None)
    } withUpdatesTo {
      probArgs.slice(0, choice) :: probArgs.slice(choice+1, len)
    } followedBy {
      assessNewDensity() } } }
```

As this example shows, Figaro provides many methods to facilitate writing proposal schemes. `beginProposal`, `followedBy` and `withUpdatesTo` thread proposal steps together and keep track of all the bookkeeping. `proposeThis` proposes the randomness element of the given model. In the proposal scheme shown here, with probability $1/2$ the randomness element of the `Select` model itself is proposed. With the remaining probability, a randomly chosen argument is proposed (`probArgs` are the probability arguments of the `Select` model). `proposeArg` recursively proposes a given argument while passing it an optional condition; in this case, there is no condition. After one argument has been proposed, the others must be updated in case they share models with the proposed argument. This is accomplished by `withUpdatesTo`, which takes a list of models to be updated. In this case, the list is all the `probArgs` except the proposed one. As a result of these steps, some of the probability values may have changed. Therefore, even though the selection itself is not changed, its density may be changed. This is captured by calling `assessOldDensity` before the argument is proposed, and `assessNewDensity` afterwards. The final probability returned by the proposal scheme will incorporate the new density divided by the old density.

5 Implementation of `Chain`

The implementation of `Apply` is fairly straightforward. However, the implementation of `Chain` involves subtleties. Recall that `Chain` takes two inputs, an argument model, and a function from values of the argument to result models. When the argument equals a certain value, we need to apply the function to obtain the result model. If we did this every time, we would create a new result model every time, and therefore we would start a new MCMC process for the result model each time. To avoid this, and make sure the state of the result models is persistent over time, we cache the function, so that the same result model is produced every time for a given argument value.

In developing the proposal scheme, the first suggestion we might try is to randomly choose between proposing the result given the current value of the argument, and proposing the argument. However, if we propose the argument and its value changes, we will at least sometimes need to propose the new result immediately. If we do not, and there is a condition on the result, it may be unlikely that the initial value of the new result model satisfies the condition. In that case, the proposal will almost certainly be rejected. On the other hand, we don't want to have to propose both the argument and the result all the time, especially since the value of the argument will often be one that has appeared before. Therefore, we use the following scheme: with probability $1/2$ we propose the result given the current value of the argument, with probability $1/4$ we propose the argument only, and with probability $1/4$ we propose both the argument and the new result.

However, if we just propose the argument and the new result, we will have a non-reversible proposal. Suppose, for example, we have a model `If(Flip(0.3), Flip(0.2), Flip(0.6))` (recall that `If` is implemented using `Chain`). There are three Boolean randomnesses. Suppose all three are currently `true`, and we propose the test to get `false` and the second consequent to get `false`. This proposal cannot be reversed, because if the test is flipped to `true`, the second consequent cannot be changed. Therefore, in the case where we propose both the argument and the result model, we also propose the previous result model. A further consideration is that when the argument and one of the result models share models, the first case where we propose the result model may also result in the argument being changed. This proposal will also not be reversible, because the

old result model will not be the result model given the new value of the argument. Therefore we disallow this first case when the argument and result share models. However, there is then a further complication. If, during a proposal, we have gone from a situation where the argument shares variables with the old result, but not the new result, that means that we would have entered this sequence of proposals (propose the argument, then the old and new result) with probability $1/2$, because the first sequence (propose the result only) is disallowed. On the other hand, for the reverse direction the first sequence is allowed, so this third sequence is only entered with probability $1/4$. Therefore the forward proposal is twice as likely as the reverse proposal. To compensate for this, we introduce a factor of $1/2$. Likewise, if the opposite situation holds, we introduce a factor of 2.

Another issue can also be illustrated with the model `If(Flip(0.3), Flip(0.2), Flip(0.6))`. Suppose we have a condition `true` on this model. This condition will be passed to the two possible consequents. During sampling, because they have the condition, the two consequents will always produce `true`. We need to take into account the fact that this value only happens with probability 0.2 for the first consequent and 0.6 for the second one. To accomplish this, we introduce a factor equal to the probability that the new result model satisfies the given extrinsic condition divided by the probability that the previous result model satisfies the extrinsic condition with which it was previously sampled. Computing these probabilities presents a challenge. One might think that we could simply use our MCMC sampler to estimate these probabilities. However, the MH state of the result models may be conditioned on the very condition whose probability we are trying to estimate. To estimate the probability correctly, we would have to use a completely fresh MH process for every model, which is prohibitive.

Our solution to this problem is to use a forward importance sampler to get a quick and dirty sample of the value of a model, conditioned only on its intrinsic conditions, and not on any extrinsic conditions. The samples are accumulated over time so that the estimates gradually improve, and the same set of samples is used for estimating the probability of different extrinsic conditions. The obvious question, though, is if we need an importance sampler, what have we gained from using MCMC at all? Isn't MCMC usually a preferred method to importance sampling? The answer is that importance sampling is a fine method when there is not too much evidence or constraints. The importance sampler is only used for an individual model and its recursive arguments, which in many cases will not contain a large amount of evidence or constraints. A program can consist of many, many models, with a large amount of evidence spread across them, but the importance sampler does not need to take it into account. The breadth of evidence and constraints is handled by the MCMC algorithm, which is much better suited to dealing with large amounts of evidence. For example, in the smoking domain, a single model may consist of the smoking habits of a pair of friends, with only a single constraint. The model as a whole contains many pairs of friends with many constraints.

6 Conclusion

In this paper, we have presented Figaro, a probabilistic programming language that takes an object-oriented approach, and demonstrated that this approach has several advantages. Ultimately, the are at least three major reasons to use probabilistic programming. One is flexibility. You can say exactly what you want to say in the way you want to say it. Figaro facilitates this capability by providing many ways to say things. Another is that probabilistic programming can be the glue that binds many reasoning paradigms together, for example for multi-intelligence reasoning or for modular learning. Again, Figaro helps by being flexible enough to express a variety of modes of reasoning. A third is that probabilistic inference is only a small fraction of what needs to be done in an application, and probabilistic programming, and particular Figaro with its tight integration with Scala, make it possible for the probabilistic inference to interact seamlessly with the rest of the application. Figaro is a real system; we have tested it for correctness on many programs and we have run it on examples with hundreds of thousands of models taking millions of samples. We do not claim that it is the fastest system for any given application, but with its expressivity, flexibility and full integration with Scala, it goes a long way towards making probabilistic programming practical for the real world.

References

[1] Pedro Domingos and Matthew Richardson. Markov logic: A unifying framework for statistical relational learning. In Lise Getoor and Ben Taskar, editors, *Statistical Relational Learning*.

MIT Press, 2007.

[2] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *Conference on Uncertainty in Artificial Intelligence*, 2008.

[3] W.K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 1970.

[4] Daphne Koller and Avi Pfeffer. Probabilistic frame-based systems. In *National Conference on Artificial Intelligence (AAAI)*, 1998.

[5] Andrew McCallum, Khashayar Rohanemanesh, Michael Wick, Karl Schultz, and Sameer Singh. FACTORIE: Efficient probabilistic programming for relational factor graphs via imperative declarations of structure, inference and learning. In *NIPS 2008 Workshop on Probabilistic Programming*, 2008.

[6] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: Probabilistic models with unknown objects. In Lise Getoor and Ben Taskar, editors, *Statistical Relational Learning*. MIT Press, 2007.

[7] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2009.

[8] Avi Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In Lise Getoor and Ben Taskar, editors, *Statistical Relational Learning*. MIT Press, 2007.

[9] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Principles of Programming Languages*, 2002.

[10] Taisuke Sato. A glimpse of symbolic-statistical modeling in PRISM. *Journal of Intelligent Information Systems*, 2008.

[11] Jue Wang and Pedro Domingos. Hybrid markov logic networks. In *AAAI Conference on Artificial Intelligence*, 2008.

[12] John Winn. Infer.NET and CSOFT. In *NIPS 2008 Workshop on Probabilistic Programming*, 2008.