15

Probabilistic Programming Languages: Independent Choices and Deterministic Systems

David Poole

Pearl [2000, p. 26] attributes to Laplace [1814] the idea of a probabilistic model as a deterministic system with stochastic inputs. Pearl defines causal models in terms of deterministic systems with stochastic inputs. In this paper, I show how deterministic systems with (independent) probabilistic inputs can also be seen as the basis of modern probabilistic programming languages. Probabilistic programs can be seen as consisting of independent choices (over which there are probability distributions) and deterministic programs that give the consequences of these choices. The work on developing such languages has gone in parallel with the development of causal models, and many of the foundations are remarkably similar. Most of the work in probabilistic programming languages has been in the context of specific languages. This paper abstracts the work on probabilistic programming languages from specific languages and explains some design choices in the design of these languages.

Probabilistic programming languages have a rich history starting from the use of simulation languages such as Simula [Dahl and Nygaard 1966]. Simula was designed for discrete event simulations, and the built-in random number generator allowed for stochastic simulations. Modern probabilistic programming languages bring three extra features:

conditioning: the ability to make observations about some variables in the simulation and to compute the posterior probability of arbitrary propositions given these observations. The semantics can be seen in terms of rejection sampling: accept only the simulations that produce the observed values, but there are other (equivalent) semantics that have been developed.

inference: more efficient inference for determining posterior probabilities than rejection sampling.

learning: the ability to learn probabilities from data.

In this paper, I explain how we can get from Bayesian networks [Pearl 1988] to independent choices plus a deterministic system (by augmenting the set of variables). I explain the results from [Poole 1991; Poole 1993b], abstracted to be language independent, and show how they can form the foundations for a diverse set of probabilistic programming languages.

Consider how to represent a probabilistic model in terms of a deterministic system with independent inputs. In essence, given the probabilistic model, we construct a random variable for each free parameter of the original model. A deterministic system can be used to obtain the original variables from the new variables. There are two possible worlds structures, the original concise set of possible worlds in terms of the original variables, and the augmented set of possible worlds in terms of the new random variables. The dimensionality of the augmented space is the number of free parameters which is greater than the dimensionality of the original space (unless all variables were independent). However, the variables in the augmented worlds can be assumed to be independent, which makes them practical to use in a programming environment. The original worlds can be obtained using abduction.

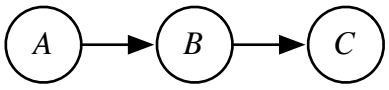
Independent choices with a deterministic programming language can be seen as the basis for most of the probabilistic programming languages, where the deterministic system can be a logic program [Poole 1993b; Sato and Kameya 1997; De Raedt, Kimmig, and Toivonen 2007], a functional program [Koller, McAllester, and Pfeffer 1997; Pfeffer 2001; Goodman, Mansinghka, Roy, Bonawitz, and Tenenbaum 2008], or even a low-level language like C [Thrun 2000].

There had been parallel developments in the development of causality [Pearl 2000], with causal models being deterministic systems with stochastic inputs. The augmented variables in the probabilistic programming languages are the variables needed for counterfactual reasoning.

1 Probabilistic Models and Deterministic Systems

In order to understand probabilistic programming languages, it is instructive to see how a probabilistic model in terms of a Bayesian network [Pearl 1988] can be represented as a deterministic system with probabilistic inputs.

Consider the following simple belief network, with Boolean random variables:



There are 5 free parameters to be assigned for this model; for concreteness assume the following values (where $A = true$ is written as a, and similarly for the other variables):

$$\begin{aligned} P(a) &= 0.1 \\ P(b|a) &= 0.8 \\ p(b|\neg a) &= 0.3 \\ P(c|b) &= 0.4 \\ p(c|\neg b) &= 0.75 \end{aligned}$$

To represent such a belief network in a probabilistic programming language, there are probabilistic inputs corresponding to the free parameters, and the programming

language specifies what follows from them. For example, in Simula [Dahl and Nygaard 1966], this could be represented as:

```
begin
   Boolean a,b,c;
   a := draw(0.1);
   if a then
      b := draw(0.8);
   else
      b := draw(0.3);
   if b then
      c := draw(0.4);
   else
      c := draw(0.75);
end
```

where $draw(p)$ is a Simula system predicate that returns true with probability p; each time it is called, there is an independent drawing.

Suppose c was observed, and we want the posterior probability of b. The conditional probability $P(b|c)$ is the proportion of those runs with c true that have b true. This could be computed using the Simula compiler by doing rejection sampling: running the program many times, and rejecting those runs that do not assign c to true. Out of the non-rejected runs, it returns the proportion that have b true. Of course, conditioning does not need to implemented that way; much of the development of probabilistic programming languages over the last twenty years is in devising more efficient ways to implement conditioning.

Another equivalent model to the Simula program can be given in terms of logic. There can be 5 random variables, corresponding to the independent draws, let's call them A, $Bifa$, $Bifna$, $Cifb$ $Cifnb$. These are independent with $P(a) = 0.1$, $P(bifa) = 0.8$, $P(bifna) = 0.3$, $P(cifb) = 0.4$, and $P(cifnb) = 0.75$. The other variables can be defined in terms of these:

$$b \iff (a \wedge bifa) \vee (\neg a \wedge bifna) \tag{1}$$

$$c \iff (b \wedge cifb) \vee (\neg b \wedge cifnbc) \tag{2}$$

These two formulations are essentially the same, they differ in how the deterministic system is specified, whether it is in Simula or in logic.

Any discrete belief network can be represented as a deterministic system with independent inputs. This was proven by Poole [1991, 1993b] and Druzdzel and Simon [1993]. These papers used different languages for the deterministic systems, but gave essentially the same construction.

2 Possible Worlds Semantics

A probabilistic programming language needs a specification of a deterministic system (given in some programming language) and a way to specify distributions over (independent) probabilistic inputs, or a syntactic variant of this. We will also assume that there are some observations, and that there are some query proposition for which we want the posterior probability.

In developing the semantics of a probabilistic programming language, we first define the set of possible worlds, and then a probability measure over sets of possible worlds [Halpern 2003].

In probabilistic programming, there are (at least) two sets of possible world that interact semantically. It is easiest to see these in terms of the above example. In the above belief network, there were three random variables A, B and C, which had complex inter-dependencies amongst them. With three binary random variables, there are 8 possible worlds. These eight possible worlds give a concise characterization of the probability distribution over these variables. I will call this the *concise* space of possible worlds.

In the corresponding probabilistic program, there is an augmented space with five inputs, each of which can be considered a random variable (these are A, $Bifa$, $Bifna$, $Cifb$ and $Cifnb$ in the logic representation). With five binary random variables, there are 32 possible worlds. The reason to increase the number of variables, and thus possible worlds, is that in this the *augmented* space, the random variables can be independent.

Note that the variables in the augmented space do not *have* to be independent. For example, $P(bifna|a)$ can be assigned arbitrarily since, when a is true, no other variable depends on $bifna$. In the augmented space, there is enough freedom to make the variables independent. Thus, we can arbitrarily set $P(bifna|a) = P(bifna|\neg a)$, which will be the same as $P(b|\neg a)$. The independence assumption makes the semantics and the computation simpler.

There are three semantics that could be given to a probabilistic program:

- The rejection sampling semantics; running the program with a random number generator, removing those runs that do not predict the observations, the posterior probability of a proposition is the limit, as the number of runs increases, of the proportion of the non-rejected runs that have the proposition true.

- The independent-choice semantics, where a possible world specifies the outcome of all possible draws. Each of these draws is considered to be independent. Given a world, the (deterministic) program would specify what follows. In this semantics, a possible world would select values for all five of the input variables in the example above, and thus gives rise to the augmented space of the above program with 32 worlds.

- The program-trace semantics, where a possible world specifies a sequence of outcomes for the draws encountered in one run of the program. In this semantics, a world would specify the values for three of the draws in the above program, as only three draws are encountered in any run of the program, and thus there would be 8 worlds.

In the logical definition of the belief network (or in the Simula definition if the draws are named), there are 32 worlds in the independent choice semantics:

World	A	$Bifa$	$Bifna$	$Cifb$	$Cifnb$	Probability
w_0	false	false	false	false	false	$0.9 \times 0.2 \times 0.7 \times 0.6 \times 0.25$
w_1	false	false	false	false	true	$0.9 \times 0.2 \times 0.7 \times 0.6 \times 0.75$
...						
w_{30}	true	true	true	true	false	$0.1 \times 0.8 \times 0.3 \times 0.4 \times 0.75$
w_{31}	true	true	true	true	true	$0.1 \times 0.8 \times 0.3 \times 0.4 \times 0.75$

The probability of each world is the product of the probability of each variable (as each of these variables is assumed to be independent). Note that in worlds w_{30} and w_{31}, the original variables A, B and C are all true; the value of $Cifnb$ is not used when B is true. These variables are also all true in the worlds that only differ in the value of $Bifna$, as again, $Bifna$ is not used when A is true.

In the program-trace semantics there are 8 worlds for this example, but not all of the augmented variables are defined in all worlds.

World	A	$Bifa$	$Bifna$	$Cifb$	$Cifnb$	Probability
w_0	false	$\bot$	false	$\bot$	false	$0.9 \times 0.7 \times 0.25$
w_1	false	$\bot$	false	$\bot$	true	$0.9 \times 0.7 \times 0.75$
...						
w_7	true	true	$\bot$	false	$\bot$	$0.1 \times 0.8 \times 0.6$
w_8	true	true	$\bot$	true	$\bot$	$0.1 \times 0.8 \times 0.4$

where $\bot$ means the variable is not defined in this world. These worlds cover all 8 cases of truth values for the original worlds that give values for A, B and C. The values of A, B and C can be obtained from the program. The idea is that a run of the program is never going to encounter an undefined value. The augmented worlds can be obtained from the worlds defined by the program trace by splitting the worlds on each value of the undefined variables. Thus each augmented world corresponds to a set of possible worlds, where the distinctions that are ignored do not make a difference in the probability of the original variables.

While it may seem that we have not made any progress, after all this is just a simple Bayesian network, we can do the same thing for any program with probabilistic inputs. We just need to define the independent inputs (often these are called *noise inputs*), and a deterministic program that gives the consequences of the choices of values for these inputs. It is reasonably easy to see that any belief network can be represented in this way, where the number of independent inputs is

equal to the number of free parameters of the belief network. However, we are not restricted to belief networks. The programs can be arbitrarily complex. We also do not need special "original variables", but can define the augmented worlds with respect to any variables of interest. Observations and queries (about which we want the posterior probability) can be propositions about the behavior of the program (e.g., that some assignment of the program variables becomes true).

When the language is Turing-equivalent, the worlds can be countably infinite, and thus there can be uncountably many worlds. A typical assumption is that the program eventually infers the observations and the query, that is, each run of the program will eventually (with probability 1) assign a truth value to any given observation and query. This is not always the case, such as when the query is to determine the fixed point of a Markov chain (see e.g., Pfeffer and Koller [2000]). We could also have non-discrete choices using continuous variables, which complicates but does not invalidate the discussion here.

A probability measure is over sets of possible worlds that form an algebra or a σ-algebra, depending on whether we want finite additivity or countable additivity [Halpern 2003]. For a programming language, we typically want countable additivity, as this allows us to not have a bound on the number of steps it takes to prove a query. For example, consider a person who plays the lottery until they win. The person will win eventually. This case is easy to represent as a probabilistic program, but requires reasoning about an infinite set of worlds.

The typical σ-algebra is the set of worlds that can be finitely described, and their (countable) union. Finitely describable means there is a finite set of draws that have their outcomes specified. Thus the probability measure is over sets of worlds that all have the same outcomes for a finite set of draws, and the union of such sets of worlds. We have a measure over such sets by treating the draws to be independent.

3 Abductive Characterization

Abduction is a form of reasoning characterized by "reasoning to the best explanation". It is typically characterized by finding a minimal consistent set of assumables that imply some observation. This set of assumables is called an *explanation* of the observation.

Poole [1991, 1993b] gave an abductive characterization of a probabilistic programming language, which gave a mapping between the independent possible world structure, and the descriptions of the worlds produced by abduction. This notion of abduction lets us construct a concise set of sets of possible worlds that is adequate to infer the posterior probability of a query.

The idea is that the the independent inputs become assumables. Given a probabilistic program, a particular observation obs and a query q, we characterize a (minimal) partition of possible worlds, where

- in each partition either $\neg obs$, $obs \wedge q$ or $obs \wedge \neg q$ can be inferred, and

- in each partition the same (finite) set of choices for the values of some of the inputs is made.

This is similar to the program-trace semantics, but will only need to make distinctions relevant to computing $P(q|obs)$. Given a probabilistic program, an observation and a query, the "explanations" of the observation conjoined with the query or its negation, produces such a partition of possible worlds.

In the example above, if the observation was $C = true$, and the query was B, we want the minimal set of assignments of values to the independent choices that gives $C = true \wedge B = true$ or $C = true \wedge B = false$. There are 4 such explanations:

- $A = true$, $Bifa = true$, $Cifb = true$
- $A = true$, $Bifa = false$, $Cifnb = true$
- $A = false$, $Bifna = true$, $Cifb = true$
- $A = false$, $Bifna = false$, $Cifnb = true$

The probability of each of these explanations is the product of the choices made, as these choices are independent. The posterior probability $P(B|C = true)$ can be easily computed by the weighted sum of the explanations in which B is true. Note also that the same explanations would be true even if C has unobserved descendants. As the number of descendants could be infinite if they were generated by a program, it is better to construct the finite relevant parts than prune the infinite irrelevant parts.

In an analogous way to how the probability of a real-variable is defined as a limit of discretizations, we can compute the posterior probability of a query given a probabilistic programming language. This may seem unremarkable until it is realized that for programs that are guaranteed to halt, there can be countably many possible worlds, and so there are uncountably many sets of possible worlds, over which to place a measure. For programs that are not guaranteed to halt, such as a sequence of lotteries, there are uncountably many possible worlds, and even more sets of possible worlds upon which to place a measure. Abduction gives us the sets of possible worlds in which to answer a conditional probability query. When the programs are not guaranteed to halt, the posterior probability of a query can be defined as the limit of the sets of possible worlds created by abduction, as long as the query can be derived in finite time for all but a set of worlds with measure zero.

In terms of the Simula program, explanations correspond to execution paths. In particular, an explanation corresponds to the outcomes of the draws in one trace of the program that infers the observations and a query or its negation. The set of traces of the program gives a set of possible worlds from which to compute probabilities.

When the program is a logic program, it isn't obvious what the program-trace semantics is. However, the semantics in terms of independent choices and abduction is well-defined. Thus it seems like the semantics in terms of abduction is more general than the program-trace semantics, as it is more generally applicable. It is also possible to define the abductive characterization independently of the details of the programming language, whereas defining a trace or run of a program depends on the details of the programming language.

Note that this abductive characterization is unrelated to MAP or MPE queries; we are defining the marginal posterior probability distribution over the query variables.

4 Inference

Earlier algorithms (e.g. Poole [1993a]) extract the minimal explanations and compute conditional probabilities from these. Later algorithms, such as used in IBAL [Pfeffer 2001], use sophisticated variable elimination to carry out inference in this space. IBAL's computation graph corresponds to a graphical representation of the explanations. Problog [De Raedt, Kimmig, and Toivonen 2007] compiles the computation graph into BDDs.

In algorithms that exploit the conditional independent structure, like variable elimination or recursive conditioning, the order that variables are summed out or split on makes a big difference to efficiency. In the independent choice semantics, there are more options available for summing out variables, thus there are more options available for making inference efficient. For example, consider the following fragment of a Simula program:

```
begin
   Boolean x;
   x := draw(0.1);
   if x then
      begin
        Boolean y := draw(0.2);
        ...
      end
   else
      begin
        Boolean z := draw(0.7);
        ...
      end
   ...
end
```

Here y is only defined when x is true and z is only defined when x is false. In the program-trace semantics, y and z are never both defined in any world. In

the independent-choice semantics, y and z are defined in all worlds. Efficiency considerations may mean that we want to sum out X first. In the independent-choice semantics, there is no problem, the joint probability on X and Y makes perfect sense. However, in the program trace semantics, it isn't clear what the joint probability of X and Y means. In order to allow for flexible elimination orderings in variable elimination or splitting ordering in recursive conditioning, the independent choice semantics is the natural choice.

Another possible way to implement probabilistic programming is to use MCMC [Milch, Marthi, Russell, Sontag, Ong, and Kolobov 2005; Goodman, Mansinghka, Roy, Bonawitz, and Tenenbaum 2008; McCallum, Schultz, and Singh 2009]. It is possible to do MCMC in either of the spaces of worlds above. The difference arises in conditionals. In the augmented space, for the example above, an MCMC state would include values for all of X, Y and Z. In the program-trace semantics, it would contain values for X and Y when $X = true$, and values for X and Z when $X = false$, as Y and Z are never simultaneously defined. Suppose X's value changes from $true$ to $false$. In the augmented space, it would just use the remembered values for Z. In the program-trace semantics, Z was not defined when Z was true, thus changing X from $true$ to $false$ means re-sampling all of the variables defined in that branch, including Z.

BLOG [Milch, Marthi, Russell, Sontag, Ong, and Kolobov 2005] and Church [Goodman, Mansinghka, Roy, Bonawitz, and Tenenbaum 2008] assign values to all of the variables in the augmented space. FACTORIE [McCallum, Schultz, and Singh 2009] works in what we have called the abductive space. Which is these is more efficient is an empirical question.

5 Learning Probabilities

The other aspect of modern probabilistic programming languages is the ability to learn the probabilities. As the input variables are rarely observed, the standard way to learn the probabilities is to use EM. Learning probabilities using EM in probabilistic programming languages is described by Sato [1995] and Koller and Pfeffer [1997]. In terms of available programming languages, EM forms the basis for learning in Prism [Sato and Kameya 1997; Sato and Kameya 2001], IBAL [Pfeffer 2001; Pfeffer 2007] and many subsequent languages.

One can do EM learning in either of the semantic structures. The difference is whether some data updates the probabilities of parameters that were not involved in computing the data. By making this choice explicit, it is easy to see that one should use the abductive characterization to only update the probabilities of the choices that were used to derive the data.

Structure learning for probabilistic programming languages has really only been explored in the context of logic programs, where the techniques of inductive logic programming can be applied. De Raedt, Frasconi, Kersting, and Muggleton [2008] overview this active research area.

6 Causal Models

It is interesting that the research on causal modelling and probabilistic programming languages have gone on in parallel, with similar foundations, but only recently have researchers started to combine them by adding causal constructs to probabilistic programming languages [Finzi and Lukasiewicz 2003; Baral and Hunsaker 2007; Vennekens, Denecker, and Bruynooghe 2009].

In some sense, the programming languages can be seen as representations for all of the counterfactual situations. A programming language gives a model when some condition is true, but also defines the "else" part of a condition; what happens when the condition is false.

In the future, I expect that programming languages will be the preferred way to specify causal models, and for interventions and counterfactual reasoning to become part of the repertoire of probabilistic programming languages.

7 Observation Languages

These languages can be used to compute conditional probabilities by having an "observer" (either humans or sensors) making observations of the world that are conditioned on. One problem that has long gone unrecognized is that it is often not obvious how to condition when the language allows for multiple individuals and relations among them. There are two main problems:

- The observer has to know what to specify and what vocabulary to use. Unfortunately, we can't expect an observer to "tell us everything that is observed". First, there are an unbounded number of true things one can say about a world. Second, the observer does not know what vocabulary to use to describe their observations of the world. As probabilistic models get more integrated into society, the models have to be able to use observations from multiple people and sensors. Often these observations are historic, or are created asynchronously by people who don't even know the model exists and are unknown when the model is being built.

- When there are are unique names, so that the observer knows which object(s) a model is referring to, an observer can provide a value to the random variable corresponding to the property of the individual. However, models often refer to roles [Poole 2007]. The problem is that the observer does not know which individual in the world fills a role referred to in the program (indeed there is often a probability distribution over which individuals fill a role). There needs to be some other mechanism other than asking for the observed value of a random variable or program variable, or the value of a property of an individual.

The first problem can be solved using ontologies. An ontology is a specification of the meaning of the symbols used in an information system. There are two main

areas that have spun off from the expert systems work of the 1970's and 1980's. One is the probabilistic revolution pioneered by Pearl. The other is often under the umbrella of knowledge representation and reasoning [Brachman and Levesque 2004]. A major aspect of this work is in the representation of ontologies that specify the meaning of symbols. An ontology language that has come to prominence recently is the language OWL [Hitzler, Krötzsch, Parsia, Patel-Schneider, and Rudolph 2009] which is one of the foundations of the semantic web [Berners-Lee, Hendler, and Lassila 2001]. There has recently been work on representing ontologies to integrate with probabilistic inference [da Costa, Laskey, and Laskey 2005; Lukasiewicz 2008; Poole, Smyth, and Sharma 2009]. This is important for Bayesian reasoning, where we need to condition on all available evidence; potentially applicable evidence is (or should be) published all over the world. Finding and using this evidence is a major challenge. This problem in being investigated under the umbrella of semantic science [Poole, Smyth, and Sharma 2008].

To understand the second problem, suppose we want to build a probabilistic program to model what apartments people will like for an online apartment finding service. This is an example where models of what people want and descriptions of the world are built asynchronously. Rather than modelling people's preferences, suppose we want to model whether they would want to move in and be happy there in 6 months time (this is what the landlord cares about, and presumably what the tenant wants too). Suppose Mary is looking for an apartment for her and her daughter, Sam. Whether Mary likes an apartment depends on the existence and the properties of Mary's bedroom and of Sam's bedroom (and whether they are the same room). Whether Mary likes a room depends on whether it is large. Whether Sam likes a room depends on whether it is green. Figure 1 gives one possible probability model, using a belief network, that follows the above story.

If we observe a particular apartment, such as the one on the right of Figure 1, it isn't obvious how to condition on the observations to determine the posterior probability that the apartment is suitable for Mary. The problem is that apartments don't come labelled with Mary's bedroom and Sam's bedroom. We need some role assignment that specifies which bedroom is Mary's and which bedroom is Sam's. However, which room Sam chooses depends on the colour of the room. We may also like to know the probability that a bachelor's apartment (that contains no bedrooms) would be suitable.

To solve the second problem, we need a representation of observations. These observations and the programs need to refer to interoperating ontologies. The observations need to refer to the existence of objects, and so would seem to need some subset of the first-order predicate calculus. However, we probably don't want to allow arbitrary first-order predicate calculus descriptions of observations. Arguably, people do not observe arbitrary disjunctions. One simple, yet powerful, observation language, based on RDF [Manola and Miller 2004] was proposed by Sharma, Poole, and Smyth [2010]. It is designed to allow for the specification of observations of

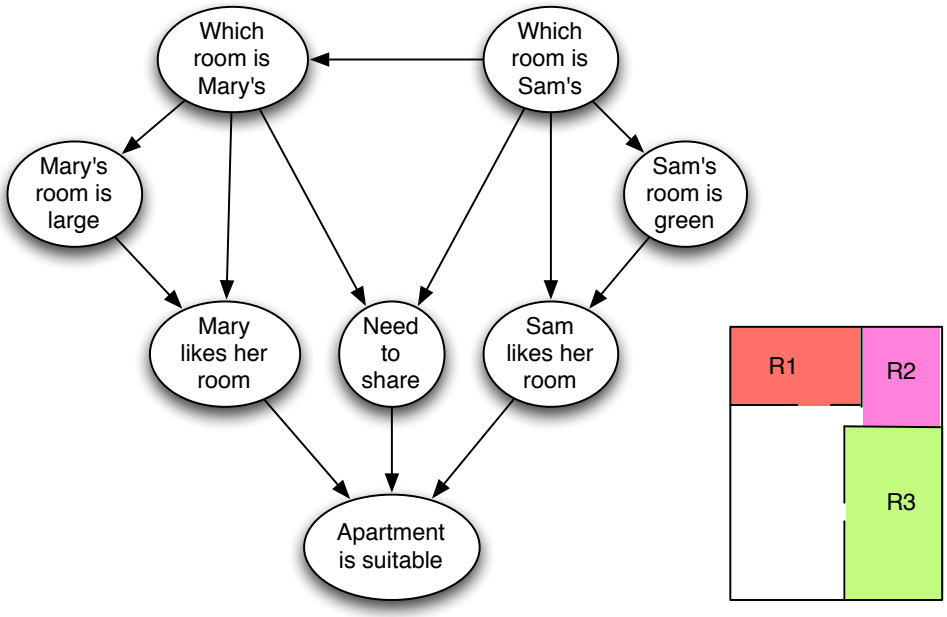


Figure 1. A possible belief network and observed apartment for the apartment example

the existence and non-existence of objects, and the properties and non-properties of objects. Observations are represented in terms of quadruples of the form:

$$\langle object, property, value, truthvalue \rangle$$

When the range of *property* is a fixed set, this quadruple means $property(object, value)$ or $\neg property(object, value)$, depending on the truth value.

When *value* is a world object, this quadruple means $\exists value\ property(object, value)$ or $\neg\exists value\ property(object, value)$, where the other mentions of *value* are in the scope of the quantification. This simple language seems to have the power to represent real observations, without representing arbitrary first-order formulae. It is then part of the program or the programming language to determine the correspondence between the objects in the language and the observed objects.

In the above example, the apartment can be described in terms of the existence of three bedrooms, one medium-sized and red (which we call $R1$), one small and pink (which we call $R2$) one large and green (which we will call $R3$). We also observe

that there is not a fourth bedroom. This can be represented as:

$$\langle apr, hasBedroom, R1, true\rangle$$
$$\langle R1, size, medium, true\rangle$$
$$\langle R1, color, red, true\rangle$$
$$\langle apr, hasBedroom, R2, true\rangle$$
$$\ldots$$
$$\langle apr, hasBedroom, R4, false\rangle$$

Thus this language is analogous to observing conjunctions of propositional atoms. However, it also lets us observe the existence and non-existence of objects, without allowing for representing arbitrary disjunctions.

Such observational languages are an important complement to probabilistic programming languages.

8 Pivotal Probabilistic Programming Language References

Probabilistic Horn abduction [Poole 1991; Poole 1993b] is the first language with a probabilistic semantics that allows for conditioning. Much of the results of this paper were presented there, in the context of logic programs. Probabilistic Horn abduction was refined into the Independent Choice Logic [Poole 1997] that allowed for choices made by multiple agents, and there is a clean integration with negation as failure [Poole 2000]. Prism introduced learning into essentially the same framework [Sato and Kameya 1997; Sato and Kameya 2001]. More recently, Problog [De Raedt, Kimmig, and Toivonen 2007] has become a focus to implement many logical languages into a common framework.

In parallel to the work on probabilistic logic programming languages, has been work on developing probabilistic functional programming languages starting with Stochastic Lisp [Koller, McAllester, and Pfeffer 1997], including IBAL [Pfeffer 2001; Pfeffer 2007], A-Lisp [Andre and Russell 2002] and Church [Goodman, Mansinghka, Roy, Bonawitz, and Tenenbaum 2008].

Other probabilistic programming languages are based on more imperative languages such as CES [Thrun 2000], based on C, and the languages BLOG [Milch, Marthi, Russell, Sontag, Ong, and Kolobov 2005] and FACTORIE [McCallum, Schultz, and Singh 2009] based on object-oriented languages. BLOG concentrates on number and identity uncertainty, where the probabilistic inputs include the number of objects and whether two names refer to the same object or not.

9 Conclusion

This paper has concentrated on similarities, rather than the differences, between the probabilistic programing languages. Much of the research in the area has concentrated on specific languages, and this paper is an attempt to put a unifying structure on this work, in terms of independent choices and abduction.

Unfortunately, it is difficult to implement an efficient learning probabilistic programming language. Most of the languages that exist have just one implementation; the one developed by the designers of the language. As these are typically research code, the various implementations have concentrated on different aspects. For example, the Prism implementation has concentrated on incorporating different learning algorithms, the IBAL implementation has concentrated on efficient inference, my AILog2 implementation of ICL has concentrated on debugging and explanation and use by beginning students[1]. Fortunately, many of the implementations are publicly available and open-source, so that they are available for others to modify.

One of the problems with the current research is that the language and the implementation are often conflated. This means that researchers feel the need to invent a new language in order to investigate a new learning or inference technique. For example, the current IBAL implementation uses exact inference, but it does not need to; different inference procedures could be used with the same language. If we want people to use such languages, they should be able to take advantage of the advances in inference or learning techniques without changing their code. One interesting project is the ProbLog project [De Raedt, Kimmig, and Toivonen 2007], which is building an infrastructure so that many of the different logic programming systems can be combined, and so that the user can use a standard language, and it can incorporate advances in inference and learning.

Probabilistic programming languages have an exciting future. We will want to have rich languages to specify causal mechanisms, processes, and rich models. How to program these models, learn them, and efficiently implement these are challenging research problems.

Acknowledgments: Thanks to Peter Carbonetto, Mark Crowley, Jacek Kisyński and Daphne Koller for comments on earlier versions of this paper. Thanks to Judea Pearl for bringing probabilistic reasoning to the forefront of AI research. This work could not have been done without the foundations he lay. This work was supported by an NSERC discovery grant to the author.

References

Andre, D. and S. Russell (2002). State abstraction for programmable reinforcement learning agents. In *Proc. AAAI-02*.

Baral, C. and M. Hunsaker (2007). Using the probabilistic logic programming language P-log for causal and counterfactual reasoning and non-naive conditioning. In *Proc. IJCAI 2007*, pp. 243–249.

Berners-Lee, T., J. Hendler, and O. Lassila (2001). The semantic web: A new

[1] We actually use it to teach logic programming to beginning students. They use it for assignments before they learn that it can also handle probabilities. The language shields the students from the non-declarative aspects of languages such as Prolog, and has many fewer built-in predicates to encourage students to think about the problem they are trying to solve.

form of web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American May*, 28–37.

Brachman, R. and H. Levesque (2004). *Knowledge Representation and Reasoning*. Morgan Kaufmann.

da Costa, P. C. G., K. B. Laskey, and K. J. Laskey (2005, Nov). PR-OWL: A Bayesian ontology language for the semantic web. In *Proceedings of the ISWC Workshop on Uncertainty Reasoning for the Semantic Web*, Galway, Ireland.

Dahl, O.-J. and K. Nygaard (1966). Simula : an ALGOL-based simulation language. *Communications of the ACM 9*(9), 671–678.

De Raedt, L., P. Frasconi, K. Kersting, and S. H. Muggleton (Eds.) (2008). *Probabilistic Inductive Logic Programming*. Springer.

De Raedt, L., A. Kimmig, and H. Toivonen (2007). ProbLog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-2007)*, pp. 2462–2467.

Druzdzel, M. and H. Simon (1993). Causality in Bayesian belief networks. In *Proc. Ninth Conf. on Uncertainty in Artificial Intelligence (UAI-93)*, Washington, DC, pp. 3–11.

Finzi, A. and T. Lukasiewicz (2003). Structure-based causes and explanations in the independent choice logic. In *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI 2003)*, Acapulco, Mexico, pp. 225–232.

Goodman, N., V. Mansinghka, D. M. Roy, K. Bonawitz, and J. Tenenbaum (2008). Church: a language for generative models. In *Proc. Uncertainty in Artificial Intelligence (UAI)*.

Halpern, J. Y. (2003). *Reasoning about Uncertainty*. Cambridge, MA: MIT Press.

Hitzler, P., M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph (2009). *OWL 2 Web Ontology Language Primer*. W3C.

Koller, D., D. McAllester, and A. Pfeffer (1997). Effective Bayesian inference for stochastic programs. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI)*, Providence, Rhode Island, pp. 740–747.

Koller, D. and A. Pfeffer (1997). Learning probabilities for noisy first-order rules,. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI)*, Nagoya, Japan, pp. 1316–1321.

Laplace, P. S. (1814). *Essai philosophique sur les probabilities*. Paris: Courcier. Reprinted (1812) in English, F.W. Truscott amd F. L. Emory (Trans.) by Wiley, New York.

Lukasiewicz, T. (2008). Expressive probabilistic description logics. *Artificial Intelligence 172*(6-7), 852–883.

Manola, F. and E. Miller (2004). *RDF Primer*. W3C Recommendation 10 February 2004.

McCallum, A., K. Schultz, and S. Singh (2009). Factorie: Probabilistic programming via imperatively defined factor graphs. In *Neural Information Processing Systems Conference (NIPS)*.

Milch, B., B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov (2005). BLOG: Probabilistic models with unknown objects. In *Proc. 19th International Joint Conf. Artificial Intelligence (IJCAI-05)*, Edinburgh.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA: Morgan Kaufmann.

Pearl, J. (2000). *Causality: Models, Reasoning and Inference*. Cambridge University Press.

Pfeffer, A. (2001). IBAL: A probabilistic rational programming language. In *Proc. 17th International Joint Conf. Artificial Intelligence (IJCAI-01)*.

Pfeffer, A. (2007). The design and implementation of IBAL: A general-purpose probabilistic language. In L. Getoor and B. Taskar (Eds.), *Statistical Relational Learning*. MIT Press.

Pfeffer, A. and D. Koller (2000). Semantics and inference for recursive probability models,. In *National Conference on Artificial Intelligence (AAAI)*.

Poole, D. (1991, July). Representing Bayesian networks within probabilistic Horn abduction. In *Proc. Seventh Conf. on Uncertainty in Artificial Intelligence (UAI-91)*, Los Angeles, pp. 271–278.

Poole, D. (1993a). Logic programming, abduction and probability: A top-down anytime algorithm for computing prior and posterior probabilities. *New Generation Computing 11*(3–4), 377–400.

Poole, D. (1993b). Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence 64*(1), 81–129.

Poole, D. (1997). The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence 94*, 7–56. special issue on economic principles of multi-agent systems.

Poole, D. (2000). Abducing through negation as failure: stable models in the Independent Choice Logic. *Journal of Logic Programming 44*(1–3), 5–35.

Poole, D. (2007, July). Logical generative models for probabilistic reasoning about existence, roles and identity. In *22nd AAAI Conference on AI (AAAI-07)*.

Poole, D., C. Smyth, and R. Sharma (2008). Semantic science: Ontologies, data and probabilistic theories. In P. C. da Costa, C. d'Amato, N. Fanizzi, K. B.

Laskey, K. Laskey, T. Lukasiewicz, M. Nickles, and M. Pool (Eds.), *Uncertainty Reasoning for the Semantic Web I*, LNAI/LNCS. Springer.

Poole, D., C. Smyth, and R. Sharma (2009, Jan/Feb). Ontology design for scientific theories that make probabilistic predictions. *IEEE Intelligent Systems 24*(1), 27–36.

Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. In *Proceedings of the 12th International Conference on Logic Programming (ICLP95)*, Tokyo, pp. 715–729.

Sato, T. and Y. Kameya (1997). PRISM: A symbolic-statistical modeling language. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI97)*, pp. 1330–1335.

Sato, T. and Y. Kameya (2001). Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research (JAIR) 15*, 391–454.

Sharma, R., D. Poole, and C. Smyth (2010). A framework for ontologically-grounded probabilistic matching. *International Journal of Approximate Reasoning In press*.

Thrun, S. (2000). Towards programming tools for robots that integrate probabilistic computation and learning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, San Francisco, CA. IEEE.

Vennekens, J., M. Denecker, and M. Bruynooghe (2009). CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming (TPLP) to appear*.