Logic + Probabilistic Programming + Causal Laws

Vaishak Belle
University of Edinburgh & Alan Turing Institute
vaishak@ed.ac.uk

Automated planning is a major topic of research in artificial intelligence, and enjoys a long and distinguished history. The classical paradigm assumes a distinguished initial state, comprised of a set of facts, and is defined over a set of actions which change that state in one way or another. Planning in many real-world settings, however, is much more involved: an agent's knowledge is almost never simply a set of facts that are true, and actions that the agent intends to execute never operate the way they are supposed to. Thus, probabilistic planning attempts to incorporate stochastic models directly into the planning process. In this article, we briefly report on probabilistic planning through the lens of probabilistic (logic) programming: a programming paradigm that aims to ease the specification of structured probability distributions using first-order logic. In particular, we provide an overview of the features of two systems, HYPE and ALLEGRO, which emphasise different strengths of probabilistic programming that are particularly useful for complex modelling issues raised in probabilistic planning. Among other things, with these systems, one can instantiate planning problems with growing and shrinking state spaces, discrete and continuous probability distributions, and non-unique prior distributions in a first-order setting.

1 Introduction

Automated planning is a major topic of research in artificial intelligence, and enjoys a long and distinguished history [18]. The classical paradigm assumes a distinguished initial state, comprised of a set of facts, and is defined over a set of actions which change that state in one way or another. Actions are further characterised in terms of their applicability conditions, that is, things that must be true for the agent to be able to execute it, and effects, which procedurally amounts to adding new facts to a state while removing others. The scientific agenda is then to design algorithms that synthesise a sequence of actions that takes the agent from an initial state to a desired goal state.

From the early days, automated planning was motivated by robotics applications. But it was observed that the real world – or more precisely, the robot's knowledge about the world – is almost never simply a set of facts that are true, and actions that the agent intends to execute never operate the way they are supposed to. One way to make sense of this complication is to separate the "high-level reasoning," in our case the planner's search space, from the low-level sensor-motor details. On the positive side, such a move allows the plan representation to be finite, discrete and simple. On the negative side, significant expert knowledge has to go into materialising this separation of concerns, possibly at the loss of clarity on the behaviour of the system as a whole.

Incidentally, by testing the robot's effectors repeatedly in a controlled environment, one can approximate the uncertain effects of an action in terms of a probability distribution. Similarly, based on minimalistic assumptions about the environment, expressed as a probabilistic prior, by repeated sampling, the robot can update its prior to converge on a reasonable posterior that approximates the environment [60]. To that end, probabilistic planning attempts to incorporate such models directly into the planning process. There are to-date numerous languages and algorithmic frameworks for probabilistic planning, e.g., [15, 9, 27, 46].

In this article, we briefly report on probabilistic planning through the lens of *probabilistic (logic) programming* [20]. Probabilistic logic programming is a programming paradigm that aims to ease the specification of structured probability distributions using first-order logic. In general, probabilistic programming languages developed so as to enable modularity and re-use in probabilistic machine learning applications. Their atomic building blocks incorporate stochastic primitives, and the formal representation also allows for compositionality. Here, we specifically provide an overview of the features of two kinds of systems, both of which have their roots in logic programming:

- HYPE [45]: a planning framework based on *distributional clauses* [22]; and
- ALLEGRO [3]: a high-level control programming framework that extends GOLOG [53].

These two systems emphasise different strengths of probabilistic programming, which we think are particularly useful for complex modelling issues raised in probabilistic planning. HYPE can easily describe growing and shrinking state spaces owing to uncertainty about the existence of objects, and thus is closely related to BLOG models [36, 58]. Since HYPE is an extension of PROBLOG [51], it stands to benefit from a wide range of applications and machine learning models explored with PROBLOG.[1] The dynamical aspects of the domain are instantiated by reifying time as an argument in the predicates, and so it is perhaps most appropriate for finite horizon planning problems.

ALLEGRO treats actions as first-class citizens and is built on a rich model of dynamics and subjective probabilities, which allows it to handle context-sensitive effect axioms, and non-unique probability measures placed on first-order formulas. GOLOG has also been widely used for a range of applications that apply structured knowledge (e.g., ontologies) in dynamical settings [30], and ALLEGRO stands to inherit these developments. GOLOG has also been shown as a way to structure search in large plan spaces [2]. Finally, since there are constructs for iteration and loops, such programs are most appropriate for modelling non-terminating behaviour [13].

In what follows, we describe the essential formal and algorithmic contributions of these systems before concluding with open computational issues. Clearly, these two systems are not the only languages with elements of logic and probability for planning: relational MDPs [56], first-order POMDPs [57], and to a much lesser extent dynamic Bayesian networks over action models [55] are part of the larger family. However, these systems are built on powerful and general logical foundations: HYPE is an extension of PROBLOG, and so allows the specification of probabilistic assertions with logic programming. ALLEGRO, on the other hand, is an extension of GOLOG, which is a dialect of first-order logic with some second-order logical features. Thus, they are indicative of what is possible when probabilities and first-order logic are unified in a dynamic setting. This sets our agenda apart from proposals such as BLOG and its dynamic version too [37, 58], which support some first-order features but do not permit arbitrary logical connectives and quantification.

To prepare for our discussion, let us briefly reflect on why probabilistic logic programming is a powerful paradigm.

2 Why probabilistic programming + logic?

Probabilistic programming is a growing field that involves representing probabilistic models as executable code [21, 33, 26, 32, 47, 38, 19]. This approach has enabled researchers to formalize, automate, and scale up many aspects of modeling and inference, making them more accessible to a broader audience of developers and domain experts. By integrating modeling and inference approaches from multiple domains, this technology has also led to the development of new programmable AI systems. Probabilistic programming is widely used for formulating and solving problems in statistics and data analysis. Stochastic programming languages, such as STAN and BUGS [29], provide a formal language that gives simple, uniform, and reusable descriptions of a wide range of models, supporting generic inference techniques. Pyro, a probabilistic programming language written in Python and supported by PyTorch, enables flexible and expressive deep probabilistic modeling, unifying deep learning and Bayesian modeling [7].

Much of the mainstream discussion falls into two camps. First, for "conventional" programming languages such as Python, the effort centers around support for wrapping neural computations in program code, as in the case of Pyro (and STAN to some extent) [8]. On the other extreme, support for functional programming and higher-order programming languages, and how a denotational semantics should be assigned to operational constructs in such languages, remains a challenge, especially constructs such as "sampling" from a, say, normal distribution [59]. The correct sampling strategy for programmatic code, especially with loops and higher-order functions, and how these samples converge for robust inference, is also a challenging problem [32].

Amidst this, an interesting entrant is probabilistic logic programming (PLP) [23, 31, 54, 40, 41, 42, 48, 50]. This concept involves the integration of probabilities into the rules of a logic program. (Although we largely focus on PROBLOG here, there's a history of interesting proposals for combining logic programming and probability, as hinted above; see [14].) Initially, the idea was to generate all possible proofs for each random choice and determine the probabilities of these "worlds" [52]. However, recent advancements have led to the development of encoding strategies of programs to the task of model counting [17]. Interestingly, this problem task can be applied to many other representations, including factor graphs, relational Bayesian networks, and Markov logic networks [61]. All this instantiates an incredibly powerful pipeline that offers exact inference.

[1] dtai.cs.kuleuven.be/problog

In our view, four key features set probabilistic logic programming apart from other probabilistic programming techniques. Firstly, probabilistic logic programming exhibits elaboration tolerance, which means that it can easily incorporate new knowledge without having to change the entire program. Secondly, it allows for the incorporation of relational knowledge, which enables modeling of random objects and their properties. Thirdly, it can handle constraints, which makes it useful for solving optimization problems. Lastly, it supports modular non-probabilistic computation, which makes it easier to develop larger and more complex systems.

We do not mean to suggest that PLPs alleviates the problems or the representational challenges of classical probabilistic programs (PPs). To a large extent, the foci and applications of classical probabilistic programming are orthogonal to those of probabilistic logic programming (PLP). While it is possible to reconstruct some simple examples from probabilistic programs to PLPs, a deeper study is clearly needed. Additionally, the integration of functional artifacts and higher-order programming in the context of PLPs requires further study, despite existing work on integrating logic programming and higher-order programming [39].

Let us make these aforementioned features concrete using a few examples. Recall that declarative programming is a programming paradigm that expresses the logic of a computation without explicitly defining how the individual steps are to be executed. We describe the "what" of the program rather than the "how to accomplish it". What might this paradigm mean in a probabilistic context? We take this to mean that (a) the instructions and logic for decision making are fixed, but (b) the probabilities, which are learned from data anyway, can be updated without needing to manipulate the logical instructions. Additionally, (c) computing with these probabilities can be done without requiring input from the user. As we will show, a language like PROBLOG (but also other probabilistic logical languages) can accomplish this easily.

Suppose there is an infectious disease spreading through a population. If two people are in regular contact and one is infected, there is a 0.6 probability that the other person will also become infected. The goal is to predict the spread of the disease given a set of initially infected people and a graph of connections between individuals in the population. We might use the following program:

```
person(a).
person(b).
0.1::inf(X) :- person(X).
0.1::contact(X,Y) :- person(X), person(Y).
0.6::inf(X) :- contact(X, Y), inf(Y).
query(inf(_)).
```

This statement indicates that a person chosen at random has a 0.1 probability of being infected. Two people chosen at random have a 0.1 probability of being in contact with each other. Finally, given a randomly chosen infected person and another person in contact with that infected person, there is a 0.6 probability that the second person will become infected.

It is clear that if we need to update the probabilities or make any of them deterministic, the logical rules are unaffected. We would only need to update the numerical values. For example, to make the contagion deterministic, we may replace the fifth line above with:

```
inf(X) :- contact(X, Y), inf(Y).
```

Indeed, McCarthy's notion of elaboration tolerance was defined as a property of a formalism that makes it convenient to modify a set of facts to accommodate new phenomena or changed circumstances. This means that the formalism can easily incorporate new knowledge without having to change the entire program. He envisoned that there are different kinds of elaboration, with the simplest being the addition of new formulas, which he called "additive elaborations." Additionally, a second type of elaboration is changing the values of parameters. Although he ultimately explored these notions using a different set of examples, it is easy to see that in a very concrete sense, elaboration tolerance is seen to be supported in PLPs. Updating the probabilities or even making probabilistic assertions deterministic is easily accomplished, as seen below (example adapted from [34, 35]):

```
0.2::inf(X) :- person(X).
0.2::contact(X,Y) :- person(X), person(Y).
inf(X) :- contact(X, Y), inf(Y).
```

In this case, the priors on infection and contact were increased to 0.2 - likely due to new data - and the spread of infection among people in contact with each other is categorical.

If we are now to incorporate multiple sources of infection (so-called inhibition effects [34]):

```
0.1::inf(X) :- person(X).
```

```
0.1::contact(X,Y) :- person(X), person(Y).
0.1::groundzero(X) :- person(X).
0.6::inf(X) :- contact(X, Y), inf(Y).
0.2::inf(X) :- groundzero(X).
```

We need then a noisy-or structure where the parents independently influence a joint effect. All of this is accomplished without manipulating the remaining rules, and the computing of the probabilities is completely hidden from the user.

Finally, we can further contextualize influence, by allowing for, say, a distinguished group of vulnerable people, while not adjusting the weights or the rules previously constructed.

```
0.1::inf(X) :- person(X).
0.1::contact(X,Y) :- person(X), person(Y).
0.1::groundzero(X) :- person(X).
0.1::susceptible(X) :- person(X).
0.6::inf(X) :- contact(X, Y), inf(Y), \+susceptible(X).
0.8::inf(X) :- contact(X, Y), inf(Y), susceptible(X).
0.2::inf(X) :- groundzero(X).
```

The program now additionally says that a vulnerable person has an increased chance of catching the infection.

It is worth noting that such languages could serve as an interface for other AI systems, including natural language interactions as well as deep learning models. For example, in [16], questions of the following sort can be parsed into PROBLOG programs.

*You roll a **fair six-sided die twice**. What is the probability that the first roll shows a five and the second roll shows a six?*

Here, the bold text is clearly a probabilistic model, and the rest a query. Correspondingly, the text is parsed and tokenized into a PROBLOG program of the following sort (with some additional syntax for probability and combinatorics constructs, including taking with replacement):

```
group(die).size(die, 6).
given(exactly(1, die, one)).
...
given(exactly(1, die, six)).
take_wr(die, rolls, 2).
probability(and(nth(1, rolls, five), nth(2, rolls, six))).
property(outcome(0), [one, two, three, four, five, six]).
```

Running this program then returns the answer to the question.

Conversely, in [31, 24], logical reasoning is used to signal deep learning models to learn distributions that respect constraints. Underlying these models is an extremely simple but ubiquitous computational task called *weighted model counting* [10]. An extension to that task for continuous models [4] has enabled a logic-based solver strategy for a range of "non-logical" (i.e., classical) probabilistic programming languages in the recent years [11, 1, 25].

3 HYPE

PROBLOG aims to unify logic programming and probabilistic specifications, in the sense of providing a language to specify distributions together with the means to query about the probabilities of events. As a very simple example, to express that the object c is on the table with a certain probability, and that all objects on the table are also in the room, we would write (free variables are assumed to be quantified from the outside):

```
.6::onTable(c).
inRoom(x) :- onTable(x).
```

This then allows us to query the probability of atoms such `inRoom(c)`.

A more recent extension [22] geared for continuous distributions and other infinite event-space distributions allows the head atom of a logical rule to be drawn from a distribution directly, by means of the following syntax:

```
h ~ D :- b1, ..., bn.
```

For example, suppose there is an urn with an unknown number of balls [36]. Suppose we pull a ball at a time and put it back in the urn, and repeat these steps (say) 6 times. Suppose further we have no means of identifying if the balls drawn were distinct from each other. A probabilistic program for this situation might be as follows:

```
n ~ poisson(6).
pos(x) ~ uniform(1,10) :-  between(1,~(n),x).
```

For simplicity, we assume here that the physical form of the urn is a straight line of length 10, and the position of a ball is assumed to be anywhere along this line.

HYPE is based on a dynamic extension that allows us to temporally index the truth of atoms, and so can be used to reason about actions. For example, the program:

```
numBehind(x,t+1) ~ poisson(1) :-  removeObj(x,t).
```

says that on removing the object x at t, we may assume that there are objects – typically one such object – behind x. Such programs can be used in object tracking applications to reason about occluded objects [43].

A common declaration in many robotics applications [60] is to define actions and sensors with an error profile, such as a Gaussian noise model. These can be instantiated in HYPE using:

```
pos(x, t+1) ~ gaussian(~(pos(x,t)) + 1, var) :- move(x,t).
obs(x,t+1) ~ gaussian(~(pos(x,t)), var).
```

The first rule says that the position of x on doing a move action is drawn from a normal distribution whose mean is x's current position incremented by one. The second one says that observing the current position of x is subject to additive Gaussian noise.

As an automated planning system, HYPE instantiates a Markov decision process (MDP) [49]. Recall that MDPs are defined in terms of states, actions, stochastic transitions and reward functions, which can be realised in the above syntax using rules such as:

```
poss(act, t) ~  conditions(t).
reward(num, t) :-  conditions(t).
```

To compute a policy, which is a mapping from states and time points to actions, HYPE combines importance sampling and SLD resolution to effectively bridge the high-level symbolic specification and the probabilistic components of the programming model. HYPE allows states and actions to be discrete or continuous, yielding a very general planning system. Empirical evaluations are reported in [45] and [44].

4 ALLEGRO

The GOLOG language has been successfully used in a wide range of applications involving control and planning [30], and is based on a simple ontology that all changes are a result of named actions [53]. An initial state describes the truth values of properties, and actions may affect these values in non-trivial context-sensitive ways. In particular, GOLOG is a programming model where executing actions are the simplest instructions in the program, upon which more involved constructions for iteration and loops are defined. For example, a program to clear a table containing an unknown number of blocks would be as follows: $([\pi \mathtt{x}\ \mathtt{onTable(x)?}; \mathtt{removeObj(x)}])^*; \neg\exists \mathtt{x}\ \mathtt{onTable(x)?}$

Here, π is the non-deterministic choice of argument, semi-colon denotes sequence, ? allows for test conditions, and $*$ is unbounded iteration. The program terminates successfully because the sub-program before the final test condition removes every object from the table.

As argued in [30], the rich syntax of GOLOG allows us, on the one hand, to represent policies and plans in an obvious fashion; for example: $(\mathtt{a}_1; \ldots; \mathtt{a}_n; \mathtt{P?})$ ensures that the goal P is true on executing the sequence of actions. However, the syntax also allows open-ended search; for example: $(\mathtt{while}\ \neg \mathtt{P}\ \pi \mathtt{a}.\ \mathtt{a})$ tries actions until P is made true. The benefit of GOLOG then is that it allows us to explore plan formulations between these two extremes, including partially specified programs that are completed by a meta-language planner.

ALLEGRO augments the underlying ontology to reason about probability distributions over state properties, and allow actions with uncertain (stochastic) effects. In logical terms, the semantical foundations rests on a rich logic of belief and actions. Consequently, it can handle partial probabilistic specifications. For example, one can say c is on the table with a certain probability as before: $\mathtt{pr(onTable(c))} = .6$, but it is also possible to express the probability that

there is an object on the table without knowing which one: $\texttt{pr}(\exists \texttt{x}\ \texttt{onTable}(\texttt{x})) = .6$. We can go further and simply say that there is a non-zero probability of that statement: $\texttt{pr}(\exists \texttt{x}\ \texttt{onTable}(\texttt{x})) > 0$, which means that any distribution satisfying the formula is admissible. Such a feature can be very useful: for example, in [28], it is argued that when planning in highly stochastic environments, it is useful to allow a margin of error in the probability distributions defined over state properties.

To model the case of Gaussian error models, actions with uncertain effects are given a general treatment. For one thing, the effects of actions are axiomatised using the notion of successor state axioms which incorporate Reiter's solution to the frame problem [53]. So, for example, changing the position of an object using a move action can be expressed as:

$\texttt{pos}(\texttt{x}, \texttt{do}(\texttt{a}, \texttt{s})) = \texttt{u} \equiv (\texttt{a} = \texttt{move}(\texttt{x}, \texttt{y}) \wedge \texttt{pos}(\texttt{x}, \texttt{s}) = \texttt{u} + \texttt{y}) \vee (\texttt{a} \neq \texttt{move}(\texttt{x}, \texttt{y}) \wedge \texttt{pos}(\texttt{x}, \texttt{s}) = \texttt{u})$.

This says that if the action of moving x was executed, its position (along a straight line) is decremented by y units, and for all other actions, the position is unaffected. To deal with uncertain effects, we will distinguish between what the agent intends and what actually happens. That is, let $move(x, y, z)$ be a new action type, where y is what the agent intends, and z is what happens. Then, the successor state axiom is rewritten as follows:

$\texttt{pos}(\texttt{x}, \texttt{do}(\texttt{a}, \texttt{s})) = \texttt{u} \equiv (\texttt{a} = \texttt{move}(\texttt{x}, \texttt{y}, \texttt{z}) \wedge \texttt{pos}(\texttt{x}, \texttt{s}) = \texttt{u} + \texttt{z}) \vee (\texttt{a} \neq \texttt{move}(\texttt{x}, \texttt{y}, \texttt{z}) \wedge \texttt{pos}(\texttt{x}, \texttt{s}) = \texttt{u})$.

The story remains essentially the same, except that z determines the actual position in the successor state, but it is not in control of the agent. A Gaussian error profile can be accorded to this action by means: $\texttt{l}(\texttt{move}(\texttt{x}, \texttt{y}, \texttt{z}), \texttt{s}) = \texttt{gaussian}(\texttt{z}; \texttt{y}, \texttt{var})$. That is, the actual value is drawn from a Gaussian whose mean is the intended argument y. Analogously, attributing additive Gaussian noise in a sensor for observing the position is defined using: $\texttt{l}(\texttt{obs}(\texttt{x}, \texttt{z}), \texttt{s}) = \texttt{gaussian}(\texttt{z}; \texttt{pos}(\texttt{x}, \texttt{s}), \texttt{var})$. That is, the observation z is drawn from a Gaussian whose mean is the actual position of the object x.

As hinted above, as an extension to GOLOG, the syntax of ALLEGRO is designed to compactly represent full or partial plans and policies in a general way, and on termination, ALLEGRO programs can be tested for any probabilistic or expectation-based criteria. The foundations of ALLEGRO was established in [3] with a discussion on its empirical behaviour against a predecessor based on goal regression.

5 Conclusions

Automated planning is often deployed in an application context, and in highly stochastic and uncertain domains, the planning model may be derived from a complex learning and reasoning pipeline, or otherwise defined over non-trivial state spaces with unknowns. In this article, we reported on two probabilistic programming systems to realise such pipelines. Indeed, combining automated planning and probabilistic programming is receiving considerable attention recently, e.g., [58]. These languages are general purpose, and their first-order expressiveness can not only enable a compact codification of the domain but also achieve computational leverage.

One of the key concerns with the use of probabilistic programming and stochastic specifications generally is that most systems perform inference by Monte Carlo sampling. As is well-known, one is able to only obtain asymptotic guarantees with such methods, and moreover, handling low-probability observations can be challenging. In that regard, there have been recent logical approaches for inferring in mixed discrete-continuous probability spaces with tight bounds on the computed answers [6, 5, 12]. Since HYPE, ALLEGRO and many such systems use probabilistic inference as a fundamental computational backbone, the question then is whether the aforementioned approaches can enable robust planning and programming frameworks in stochastic domains.

References

[1] A. Albarghouthi, L. D'Antoni, S. Drews, and A. V. Nori. Fairsquare: probabilistic verification of program fairness. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

[2] J. A. Baier, C. Fritz, and S. A. McIlraith. Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proc. ICAPS*, pages 26–33, 2007.

[3] V. Belle and H. J. Levesque. Allegro: Belief-based programming in stochastic dynamical domains. In *IJCAI*, 2015.

[4] V. Belle, A. Passerini, and G. Van den Broeck. Probabilistic inference in hybrid domains by weighted model integration. In *IJCAI*, 2015.

[5] V. Belle, A. Passerini, and G. Van den Broeck. Probabilistic inference in hybrid domains by weighted model integration. In *IJCAI*, 2015.

[6] V. Belle, G. Van den Broeck, and A. Passerini. Hashing-based approximate probabilistic inference in hybrid domains. In *UAI*, 2015.

[7] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978, 2019.

[8] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978, 2019.

[9] C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11(1):94, 1999.

[10] M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7):772–799, 2008.

[11] D. Chistikov, R. Dimitrova, and R. Majumdar. Approximate counting in smt and value estimation for probabilistic programs. In *TACAS*, volume 9035, pages 320–334. 2015.

[12] D. Chistikov, R. Dimitrova, and R. Majumdar. Approximate counting in smt and value estimation for probabilistic programs. In *TACAS*, volume 9035, pages 320–334. 2015.

[13] J. Claßen and G. Lakemeyer. A logic for non-terminating golog programs. In *KR*, pages 589–599, 2008.

[14] L. De Raedt and A. Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100:5–47, 2015.

[15] C. Domshlak and J. Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *JAIR*, 30:565–620, 2007.

[16] A. Dries, A. Kimmig, J. Davis, V. Belle, and L. De Raedt. Solving probability problems in natural language. In *IJCAI*, 2017.

[17] D. Fierens, G. V. den Broeck, I. Thon, B. Gutmann, and L. D. Raedt. Inference in probabilistic logic programs using weighted CNF's. In *UAI*, pages 211–220, 2011.

[18] R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. In *Proc. IJCAI*, pages 608–620, 1971.

[19] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Proc. UAI*, pages 220–229, 2008.

[20] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Proc. International Conference on Software Engineering*, 2014.

[21] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Proc. International Conference on Software Engineering*, 2014.

[22] B. Gutmann, I. Thon, A. Kimmig, M. Bruynooghe, and L. De Raedt. The magic of logical inference in probabilistic programming. *TPLP*, 11:663–680, 2011.

[23] B. Gutmann, I. Thon, A. Kimmig, M. Bruynooghe, and L. De Raedt. The magic of logical inference in probabilistic programming. *TPLP*, 11:663–680, 2011.

[24] N. Hoernle, R. M. Karampatsis, V. Belle, and K. Gal. Multiplexnet: Towards fully satisfied logical constraints in neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 5700–5709, 2022.

[25] S. Holtzen, G. Van den Broeck, and T. Millstein. Scaling exact inference for discrete probabilistic programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–31, 2020.

[26] C. Jones and G. Plotkin. A probabilistic powerdomain of evaluations. In *Logic in Computer Science, 1989. LICS '89, Proceedings., Fourth Annual Symposium on*, pages 186–195, Jun 1989.

[27] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99 – 134, 1998.

[28] L. P. Kaelbling and T. Lozano-Pérez. Integrated task and motion planning in belief space. *I. J. Robotic Res.*, 32(9-10):1194–1227, 2013.

[29] F. Korner-Nievergelt, T. Roth, S. Von Felten, J. Guélat, B. Almasi, and P. Korner-Nievergelt. *Bayesian data analysis in ecology using linear models with R, BUGS, and Stan*. Academic Press, 2015.

[30] G. Lakemeyer and H. J. Levesque. Cognitive robotics. In *Handbook of Knowledge Representation*, pages 869–886. Elsevier, 2007.

[31] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, and L. De Raedt. Deepproblog: Neural probabilistic logic programming. *Advances in Neural Information Processing Systems*, 31, 2018.

[32] V. Mansinghka. *Natively Probabilistic Computation*. PhD thesis, Massachusetts Institute of Technology, 2009. MIT/EECS George M. Sprowls Doctoral Dissertation Award.

[33] A. McCallum, K. Schultz, and S. Singh. FACTORIE: probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems 2009. Proceedings of a meeting held 7-10 December 2009, Vancouver, British Columbia, Canada.*, pages 1249–1257, 2009.

[34] W. Meert, J. Struyf, and H. Blockeel. Learning ground cp-logic theories by leveraging bayesian network learning techniques. *Fundamenta Informaticae*, 89(1):131–160, 2008.

[35] W. Meert and J. Vennekens. Inhibited effects in cp-logic. In *Probabilistic Graphical Models: 7th European Workshop, PGM 2014, Utrecht, The Netherlands, September 17-19, 2014. Proceedings 7*, pages 350–365. Springer, 2014.

[36] B. Milch, B. Marthi, S. Russell, D. Sontag, D. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. *Introduction to statistical relational learning*, page 373, 2007.

[37] B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *Proc. IJCAI*, pages 1352–1359, 2005.

[38] B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *Proc. IJCAI*, pages 1352–1359, 2005.

[39] G. Nadathur and D. J. Mitchell. System description: Teyjus-a compiler and abstract machine based implementation of lambda-prolog. In *CADE*, volume 16, pages 287–291. Springer, 1999.

[40] R. Ng and V. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

[41] L. Ngo and P. Haddawy. Probabilistic logic programming and Bayesian networks. *Algorithms, Concurrency and Knowledge*, pages 286–300, 1995.

[42] D. Nitti. *Hybrid Probabilistic Logic Programming*. PhD thesis, KU Leuven, 2016.

[43] D. Nitti. *Hybrid Probabilistic Logic Programming*. PhD thesis, KU Leuven, 2016.

[44] D. Nitti, V. Belle, T. De Laet, and L. De Raedt. Planning in hybrid relational mdps. *Machine Learning*, pages 1–28, 2017.

[45] D. Nitti, V. Belle, and L. D. Raedt. Planning in discrete and continuous markov decision processes by probabilistic programming. In *ECML*, 2015.

[46] S. C. W. Ong, S. W. Png, D. Hsu, and W. S. Lee. Planning under uncertainty for robotic tasks with mixed observability. *Int. J. Rob. Res.*, 29(8):1053–1068, July 2010.

[47] A. Pfeffer. Ibal: A probabilistic rational programming language. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'01, pages 733–740, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[48] D. Poole. Logic, probability and computation: Foundations and issues of statistical relational AI. In *Logic Programming and Nonmonotonic Reasoning*, volume 6645 of *LNCS*, pages 1–9. 2011.

[49] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[50] L. D. Raedt and K. Kersting. Probabilistic inductive logic programming. In *Probabilistic Inductive Logic Programming*, pages 1–27, 2008.

[51] L. D. Raedt, A. Kimmig, and H. Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *Proc. IJCAI*, pages 2462–2467, 2007.

[52] L. D. Raedt, A. Kimmig, and H. Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *Proc. IJCAI*, pages 2462–2467, 2007.

[53] R. Reiter. *Knowledge in action: logical foundations for specifying and implementing dynamical systems*. MIT Press, 2001.

[54] S. Sanghai, P. Domingos, and D. Weld. Relational Dynamic Bayesian Networks. *Journal of Artificial Intelligence Research*, 24:759–797, 2005.

[55] S. Sanner. Relational dynamic influence diagram language (rddl): Language description. Technical report, Australian National University, 2011.

[56] S. Sanner, K. V. Delgado, and L. N. de Barros. Symbolic dynamic programming for discrete and continuous state MDPs. In *Proc. UAI*, pages 643–652, 2011.

[57] S. Sanner and K. Kersting. Symbolic dynamic programming for first-order pomdps. In *Proc. AAAI*, pages 1140–1146, 2010.

[58] S. Srivastava, S. J. Russell, P. Ruan, and X. Cheng. First-order open-universe pomdps. In *UAI*, pages 742–751, 2014.

[59] S. Staton, H. Yang, F. Wood, C. Heunen, and O. Kammar. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 525–534, 2016.

[60] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005.

[61] G. Van den Broeck, W. Meert, and A. Darwiche. Skolemization for weighted first-order model counting. In *KR*, 2014.