The Principles and Practice of Probabilistic Programming

Noah D. Goodman
Stanford University
ngoodman@stanford.edu

Categories and Subject Descriptors D [*3*]: m

Keywords probabilistic models, probabilistic programs

Probabilities describe degrees of belief, and probabilistic inference describes rational reasoning under uncertainty. It is no wonder, then, that probabilistic models have exploded onto the scene of modern artificial intelligence, cognitive science, and applied statistics: these are all sciences of inference under uncertainty. But as probabilistic models have become more sophisticated, the tools to formally describe them and to perform probabilistic inference have wrestled with new complexity. Just as programming beyond the simplest algorithms requires tools for abstraction and composition, complex probabilistic modeling requires new progress in model representation—*probabilistic* programming languages. These languages provide compositional means for describing complex probability distributions; implementations of these languages provide *generic* inference engines: tools for performing efficient probabilistic inference over an arbitrary program.

In their simplest form, probabilistic programming languages extend a well-specified deterministic programming language with primitive constructs for random choice. This is a relatively old idea, with foundational work by Giry, Kozen, Jones, Moggi, Saheb-Djahromi, Plotkin, and others [see e.g. 7]. Yet it has seen a resurgence thanks to new tools for probabilistic inference and new complexity of probabilistic modeling applications. There are a number of recent probabilistic programming languages [e.g. 8, 9, 11–17], embodying different tradeoffs in expressivity, efficiency, and perspicuity. We will focus on the probabilistic programming language Church [6] for simplicity, but the design of probabilistic languages to best support complex model representation and efficient inference is an active and important topic.

Church extends (the purely functional subset of) Scheme with elementary random primitives, such as flip (a bernoulli), multinomial, and gaussian. In addition, Church includes language constructs that simplify modeling. For instance, mem, a higher-order procedure that memoizes its input function, is useful for describing persistent random properties and lazy model construction. (Interestingly, memoization has a semantic effect in probabilistic languages.) If we view the semantics of the underlying deterministic language as a map from programs to executions of the program, the semantics of the probabilistic language will be a map from programs to distributions over executions. When the program halts with probability one, this induces a proper distribution over return values. Indeed, *any* computable distribution can be represented as the distribution induced by a Church program in this way (see [3, §6], [1, §11], and citations therein).

```
(define (rejection-query thunk condition)
  (let ((val (thunk)))
    (if (condition val)
        val
        (rejection-query thunk condition))))
```

```
(query
  ...defines...
  query-expression
  condition-expression)
```

```
(define (thunk)
  ...defines...
  (pair query-expression
        condition-expression))
(define condition rest)
```

Figure 1. (Top) Defining conditional inference in Church as a stochastic recursion: rejection sampling represents the conditional probability of the thunk conditioned on the condition predicate being true. We typically use special query syntax (Bottom, left), which can be desugared into a query thunk (Bottom, right).

Probabilistic graphical models [10], aka Bayes nets, are one of the most important ideas of modern AI. Probabilistic programs extend probabilistic graphical models, leveraging concepts from programming language research. Indeed, graphical models can be seen as flow diagrams for probabilistic programs—and just as flow diagrams for deterministic programs are useful but not powerful enough to represent general computation, graphical models are a useful but incomplete approach to probabilistic modeling. For an example of this, we need look no further than the fundamental operation for inference, probabilistic conditioning, which forms a posterior distribution over executions from the prior distribution specified by the program. Conditioning is typically viewed as a special operation that happens *to* a probabilistic model (capturing observations or assumptions), not one that can be expressed *as* a model. However, because probabilistic programs allow *stochastic recursion*, conditioning can be defined as an ordinary probabilistic function (Fig. 1, Top). (However see [1] for complications in the case of continuous values.)

A wide variety of probabilistic models are useful for diverse tasks, including unsupervised learning, vision, planning, and statistical model selection. Due to space limitations, we only mention two characteristic examples here. The first, Fig. 2, captures key concepts for commonsense reasoning about the game tug-of-war. This "conceptual library" of probabilistic functions can be used to reason about many patterns of evidence, via different queries, without needing to specify ahead of time the set of people, the teams, or the matches. The program thus enables a very large numbers of different inferences to be modeled, and the model predictions match human intuitions quite well (a correlation of 0.98 between

POPL'13, January 23–25, 2013, Rome, Italy.
ACM 978-1-4503-1832-7/13/01.

```
;;Strength is a persistent property of each person,
    hence memoized:
(define strength
  (mem (lambda (person) (gaussian 1.0 1.0))))

;;Laziness varies from match to match:
(define (lazy person) (flip 0.3))

;;When a person is lazy they pull less:
(define (pulling person) (if (lazy person)
                             (/ (strength person) 2)
                             (strength person)))

;;Total pulling of the team is the sum:
(define (total-pulling team) (sum (map pulling team)))

;;The winning team pulls hardest:
(define (winner team1 team2)
  (if (< (total-pulling team1) (total-pulling team2))
      team2
      team1))
```

Figure 2. Modeling intuitive concepts in the tug-of-war domain. While this model is simple, probabilistic queries can explain human reasoning from diverse evidence with high quantitative accuracy [4].

```
;;The basic listener: how is the world, given that the
    utterance is true?
(define (literal-listener utterance)
  (query
    (define world (world-prior))
    world
    ((meaning utterance) world)))

;;The speaker: what should I say, so that the listener
    infer the right world?
(define (speaker world)
  (query
    (define utterance (language-prior))
    utterance
    (equal? world (literal-listener utterance))))

;;The smart listener: how is the world, given that a
    speaker chose this utterance to express it?
(define (listener utterance)
  (query
    (define world (world-prior))
    world
    (equal? utterance (speaker world))))
```

Figure 3. A probabilistic model of natural language pragmatics using nested-query to model reasoning-about-reasoning.

model and human judgements in the experiments of [4]). The ability to model many inferences is inherited from the productivity of the underlying programming language, while the ability to capture complex, graded commonsense reasoning is inherited from probabilistic inference.

A more subtle model is shown in Fig. 3. Here we model pragmatic inference in natural language following [2, 5, 18]. Critically, because query is an ordinary function that may be nested in itself, we are able to model a listener reasoning about a speaker, who reasons about a naive listener. This model formalizes the idea that a listener is trying to infer what the speaker intended, while a speaker is trying to make the listener form a particular belief. Versions of this model again predict human judgements with high quantitative accuracy [2, 5].

The critical obstacle to probabilistic programming as a practical tool is efficient implementation of the inference operator. It is clear that the expected runtime for the rejection-query operator in Fig. 1 increases as the probability of satisfying the condition decreases. Hence, while rejection is useful as a definition, it is impractical for real modeling tasks, where the condition is typically very unlikely. An obvious alternative that does not take longer when the condition is less likely is to enumerate all execution paths of the program, for instance by using co-routines. Unfortunately, naive enumeration takes time proportional to the number of executions, which grows exponentially in the number of random choices. This can be ameliorated in some cases by using dynamic programming. Indeed it is possible to do inference for the simple pragmatics model in Fig. 3 by using a sophisticated dynamic programming approach [18].

For other classes of program, such as the tug-of-war model, Fig. 2, it can be better to switch to an approximate inference technique based on Markov chain Monte Carlo. One such algorithm [6] can be derived from the Metropolis Hastings recipe: a random change to a single random choice is made on each step, leading to a random walk in the space of program executions. This walk is guaranteed to visit executions in proportion to their probability. We individuate random choices using their position in the execution—this can be achieved by a code transformation, resulting in a very lightweight inference process [20].

Each step of this MCMC process requires re-executing the program, in order to compute updated probabilities. Several orders of magnitude in speed increases are seen by just-in-time compiling this probability update [21]. To do so, we separate structural random choices (those that can influence control flow) from non-structural. When we encounter a new setting of the structural choices we partially-evaluate the program to straight-line code, which is aggressively optimized. Structural choices can sometimes be found automatically by flow analysis, but in other instances must be annotated by hand.

The distinction between structural and non-structural choices can also be used to construct an algorithm that more effectively explores the space of executions for "open universe" models—those that define distributions over a variable number of entities [22]. This class of programs is especially well-suited to complex procedural modeling in computer graphics. For instance, Yeh et al. [22] describe models for layout of virtual worlds. These procedural models typically have many real-valued random choices. In this setting Hamiltonian Monte Carlo, which exploits the gradient of the score to make more well-directed steps in execution space, can be a useful additional tool. The needed gradient can be derived automatically by a non-standard interpretation of the program [19].

The sampling of inference algorithms described above suggests an important generalization: advanced statistical inference techniques can be implemented for arbitrary programs by leveraging methods developed in programming language research (program analysis, non-standard interpretation, partial evaluation, etc.). Existing research only scratches the surface of potential interactions between statistics, AI, and PL. Though technically demanding, this interaction may be the critical ingredient in bringing complex probabilistic modeling to the wider world.

Acknowledgments

Thanks to Dan Roy, Cameron Freer, and Andreas Stuhlmüller for helpful comments. This work was supported by ONR grant N00014-09-0124.

References

[1] N. Ackerman, C. Freer, and D. Roy. Noncomputable conditional distributions. In *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*, pages 107–116. IEEE, 2011.

[2] M. Frank and N. Goodman. Predicting pragmatic reasoning in language games. *Science*, 336(6084):998–998, 2012.

[3] C. E. Freer and D. M. Roy. Computable de Finetti measures. *Annals of Pure and Applied Logic*, 163(5):530–546, 2012. doi: 10.1016/j.apal.2011.06.011.

[4] T. Gerstenberg and N. D. Goodman. Ping pong in Church: Productive use of concepts in human probabilistic inference. In *Proceedings of the 34th annual conference of the cognitive science society*, 2012.

[5] N. Goodman and A. Stuhlmüller. Knowledge and implicature: Modeling language understanding as social cognition. *Topics in Cognitive Science*, 2013.

[6] N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum. Church: A language for generative models. In *In UAI*, 2008.

[7] C. Jones and G. Plotkin. A probabilistic powerdomain of evaluations. In *Logic in Computer Science (LICS), 1989 4th Annual IEEE Symposium on*, pages 186–195, Jun 1989. doi: 10.1109/LICS.1989.39173.

[8] A. Kimmig, B. Demoen, L. De Raedt, V. S. Costa, and R. Rocha. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming*, 11(2-3):235–262, 2011.

[9] O. Kiselyov and C. Shan. Embedded probabilistic programming. In *Domain-Specific Languages*, pages 360–384, 2009.

[10] D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

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

[12] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1352–1359, 2005.

[13] A. Pfeffer. IBAL: A probabilistic rational programming language. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 733–740. Morgan Kaufmann Publ., 2001.

[14] A. Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, 2009.

[15] D. Poole. The independent choice logic and beyond. *Probabilistic inductive logic programming*, pages 222–243, 2008.

[16] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62:107–136, 2006.

[17] T. Sato and Y. Kameya. PRISM: A symbolic-statistical modeling language. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 1997.

[18] A. Stuhlmüller and N. Goodman. A dynamic programming algorithm for inference in recursive probabilistic programs. *arXiv preprint arXiv:1206.3555*, 2012.

[19] D. Wingate, N. Goodman, A. Stuhlmueller, and J. Siskind. Nonstandard interpretations of probabilistic programs for efficient inference. In *Advances in Neural Information Processing Systems 23*, 2011.

[20] D. Wingate, A. Stuhlmueller, and N. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the 14th international conference on Artificial Intelligence and Statistics*, page 131, 2011.

[21] L. Yang, P. Hanrahan, and N. Goodman. Incrementalizing mcmc on probabilistic programs through tracing and slicing. Under review.

[22] Y. Yeh, L. Yang, M. Watson, N. Goodman, and P. Hanrahan. Synthesizing open worlds with constraints using locally annealed reversible jump mcmc. *ACM Transactions on Graphics (TOG)*, 31 (4):56, 2012.