The State of Probabilistic Programming

For two weeks last July, I cocooned myself in a hotel in Portland, OR, living and breathing probabilistic programming as a "student" in the probabilistic programming summer school run by DARPA. The school is part of the broader DARPA program on Probabilistic Programming for Advanced Machine Learning (PPAML), which has resulted in a great infusion of energy (and funding) into the probabilistic programming space. Last year was the inaugural one for the summer school, one that is meant to introduce and disseminate the languages and tools being developed to the broader scientific and technology communities. The school was graciously hosted by Galois Inc., which did a terrific job of organizing the event. Thankfully, they're hosting the summer school again this year (there's still time to apply!), which made me think that now is a good time to reflect on last year's program and provide a snapshot of the state of the field. I will also take some liberty in prognosticating on the future of this space. Note that I am by no means a probabilistic programming expert, merely a curious outsider with a problem or two to solve.

What is probabilistic programming?

Before I get on with describing my impressions, I will give a very quick primer on probabilistic programming. This is not meant to be in-depth as there are many excellent existing resources. In particular, if you find my discussion too brief then I suggest this introduction, which provides a more substantial treatment. Other accessible resources include an in-depth walk-through using the Church language here, and a portal on probabilistic programming systems (PPSs) here. Incidentally, PPS is the preferred terminology for DARPA, meant I believe to emphasize the fact that PPSs are more than "just" programming languages, as they typically include an inference engine among other things.

At their most basic level, probabilistic programming languages differ from deterministic ones by allowing language primitives to be stochastic. In other words, instead of being restricted to deterministic assignments such as:

```
x = 5
```

one can also specify a probability distribution from which x is drawn, e.g.:

```
x ~ normal(mu =
                                0, sigma = 1)
```

Depending on the expressiveness of the language, the distributions from which primitives are drawn can be quite complex (e.g. distributions over functions). The power of probabilistic programming doesn't come from merely specifying probabilistic primitives however, as that can be easily done with standard programming languages (e.g. rand). Instead it is their ability to condition on observations (or functions thereof). For example, we can write something like the following program:

```
mu ~ uniform(-10, 10)
```

```
x ~ normal(mu, sigma = 1)

observe(x = 5)
```

The observed value of x is used to constrain the space of possible program executions to a specific subset, namely ones in which x = 5. This in turn makes it possible to perform *inference* . E.g. we can then add:

```
infer(mu)
```

And the PPS will provide an estimated value of mu that correspond to the conditional distribution of mu given x = 5. Such estimates are typically derived either through sampling approaches like MCMC or through optimization approaches that provide the maximum a posteriori estimate . This makes it possible to construct reasonably complex probabilistic models with relative ease, and in principle, enables machine learning practitioners to explore novel models rapidly. An impressive demonstration of the breadth of models that can be coded using a tiny amount of code can be found here . It includes things like nested Dirichlet processes, infinite probabilistic context-free grammars, and more pedestrian models like Latent Dirichlet Allocation. The "point" of probabilistic programming systems is something that I will expound on below, but it is important to note from the get-go that they are not meant to compete with ML frameworks like scikit-learn . Unlike the latter set, which typically provide a fixed set of models and algorithms, PPSs enable the construction of entirely novel models, and in some cases, inference algorithms. As a natural consequence of this, PPSs are unlikely to be as efficient in running well-trod models as specialized algorithms.

The Landscape

An impressive number of PPSs were on display at the PPAML summer school. Three languages, Venture (MIT) , Figaro (Charles River Analytics) , and BLOG (Berkeley) , were given the lion's share of attention due to their (relative) maturity, but other languages present included Church (Stanford) , Haraku (University of Indiana) , and Chimple/Dimple (Gamalon) . This is by no means a full accounting of the space of PPSs, which includes Stan (Columbia) , Infer.NET (Microsoft) , and very many others; the DARPA summer school naturally focused on the DARPA-funded PPSs.

There are many ways to classify and think about all these languages. The space is fragmented right now, with different groups experimenting with widely divergent approaches. The result is that there's a great deal of diversity and uncertainty, and in some ways this is one of the major challenges facing the field, as I describe later. For now, I found the following criteria to be helpful in thinking about PPSs:

- Expressivity
- Scalability
- Maturity
- Programming Paradigm
- Use Case / Niche

I will go through each in turn.

Expressivity

How expressive is the language? There are in fact two questions baked into this. One is the theoretical expressivity of the language, i.e. the set of probabilistic programs that it can represent. The second is the practical expressivity of the language, i.e. how easy it is to write programs of increasing complexity. Two languages may in principle be capable of representing any samplable probability distribution, but in practice one language may really be designed for a certain subset of problems, which makes it awkward to express more general distributions. Almost all languages on display were in principle capable of representing arbitrary distributions. The exceptions to this were Dimple, which is restricted to graphical models, specifically factor graphs, and BLOG, which is focused on logic models. Outside of PPAML, Stan and Infer.NET are also languages that are not capable of representing arbitrary probabilistic programs. In some instances claims of expressiveness can be contentious, for example see this blog post on whether Stan is Turing complete. My criteria is based on whether it's possible to represent arbitrary probability distributions using the language's built-in constructs.

Venture, Figaro, Church, and Chimple can all in principle represent arbitrary probability distributions. In practice however, the expressivity of these languages varies a great deal. Venture, and its antecedent Church, have the cleanest language design. They were intended for unconstrained probabilistic programming and so their designers took (and are taking, as much of the language design for Venture is still in flux) great care in providing clear and concise semantics for representing probabilistic programs. Figaro and Chimple focus on certain problem domains, for example graphical models in the case of Figaro, which makes them somewhat awkward to use for more general applications. Many of the language's constructs and general idioms are geared specifically toward graphical models.

Naturally, expressiveness comes at a cost. More expressive languages may not be able to carry out inference as efficiently as the languages with more specialized constructs, since they can't assume as much about the model. Another potential drawback to expressivity is the ease with which users can write ill-defined probabilistic programs and ones in which efficient sampling is nearly impossible due to the complexity of their resulting models. More expressive languages provide longer proverbial ropes. Finally an issue that all languages face, but in particular the "clean" ones, is the level of abstraction that should be provided to the user. Ideally a handful of powerful constructs would be sufficient to build most probabilistic programs of interest. The design of Venture/Church accomplishes this quite well. For example a Dirichlet process (DP) can be written from scratch in a few lines of code. Unfortunately, such clean implementations written in the language itself end up not being terribly efficient, necessitating that ready-made constructs for commonly used distributions like DPs are built into the language. The need to provide higher-level black box implementations for commonly used distributions detracts from the cleanliness and unification of probabilistic programming. These issues arise in programming languages in general and scientific computing in particular. For example Matlab provides a host of specialized functions for performing linear algebra, and doing the naïve but clean implementation can often prove sub-optimal. These challenging design decisions are some of the more difficult problems facing the PPS ecosystem right now.

Performance

The scale at which PPSs can carry out inference, i.e. the size of data to condition on, and their general performance characteristics was another area in which languages varied considerably. Some, like Church, are focused primarily on language design and the ability to express very complex probabilistic programs (more on this later). While they may not necessarily be inefficient, sampling efficiency is clearly not their priority. Others, like Figaro and the Chimple/Dimple pair, provide efficient sampling for certain subsets of probabilistic programs, for example graphical models. [Ch/D]imple chose an interesting design route where Chimple is a general-purpose PPS that uses Metropolis-Hastings for everything, while Dimple is a separate language that's meant specifically for graphical models and strives for high performance in that domain.

In the scalability/expressiveness space, Venture is the language that is attempting to have its cake and eat it too, by being simultaneously highly expressive and capable of carrying out serious, industrial-strength, inference. Its approach is two-pronged. First, unlike other languages which only provide one or two sampling techniques, Venture provides a plethora of built-in algorithms, including less common techniques like Slice sampling . Second, Venture exposes the capability to program the inference procedure itself through a built-in inference programming language. For the version at the summer school, what this meant is the ability to tag different random variables as belonging to distinct scopes, and then to define a meta sampling procedure, constructed out of the existing sampling primitives, that specifies how and when the different scopes are sampled. Ultimately their vision is for the inference procedure itself to be an arbitrary probabilistic program.

Figaro also has some inference programming capabilities, although they did not appear to be as extensive as Venture's, but I suspect this will be an area of future focus for the language. In addition, although Church is not focused on high-performance inference, its programs are converted to JavaScript. This means its performance is actually quite competitive, benefiting from the stupendous efforts that browser developers have put into optimizing compilers.

My impression is that DARPA, and the three highlighted PPSs (Venture, Figaro, and BLOG), all consider high-performance inference to be a major objective. I think all groups are wary of overpromising on this account, as there remain fundamental open questions regarding how doable general inference is, even in principle. Several negative results have already been established about the viability of inference in all samplable distributions, but I suspect as is often the case with such negative results, they are extremely pessimistic and in practice the subset of problems we care about may prove to be much more tractable (the halting problem hasn't stopped people from building perfectly functional software). For now high-performance inference remains an objective and not a reality, particularly when considering general probabilistic programs. All examples shown were basically toyish in scale, if not in model complexity, and no language currently comes close to competing with custom C/C++ code. For specific subclasses of systems, the situation is different, and languages like Figaro and BLOG can offer competitive performance. This is a fundamental design consideration moving forward, as more expressive languages may end up sacrificing performance because of their inability to assume as much about the underlying set of possible programs.

Maturity

For a field as nascent as probabilistic programming, one might assume that all existing languages are equally (im)mature. This is in fact untrue for several reasons. First, older modeling languages are jumping on the probabilistic programming bandwagon by rebranding themselves. Second, many PPSs started out as tools for constructing graphical models and evolved into more general-purpose modeling languages. Finally, even within the world of genuine probabilistic programming, some languages have had a few years head start on others.

Broadly speaking, the new languages tend to be cleaner and well designed, while the older ones have usually made some questionable design choices. On the other hand, because of the immaturity of the newer languages, they are less optimized from a performance standpoint and suffer from the usual glitches and instabilities of alpha/beta software.

At PPAML the most mature PPS on display was BLOG. There was a competition of sorts (DARPA doesn't like calling it that) to evaluate the PPSs on "Challenge Problems" that have been designed to test out the expressivity and performance of PPSs. BLOG was the only language able to run the Challenge Problems at the largest scale, including the million birds version of one of the Challenge Problems. This is undoubtedly thanks to the fact that BLOG has been in development for nearly 10 years and is now a well-tested and well-optimized system.

The intermediate case was Figaro, which had more of the trappings of a modern PPS system, including integration into Scala and the expressivity to represent nonparametric distributions, but is perhaps not quite as mature as BLOG given that it's been in development for only a few years. Venture represented the frontier, a very modern language with substantial potential, but one that is very much a work-in-progress with frequent crashes and spartan support for even basic functional programming constructs. Its developers were forthright about it being alpha software, and I am hopeful that it will rapidly mature over the course of the next year or two.

Since the summer school, Haraku, which was not officially released yet, has since become available. I suspect we will see more languages emerge as well. The fact that there's such a broad distribution of maturity levels makes choosing a language today somewhat tricky. Investing in a solid language like BLOG or Stan will yield immediate dividends in terms of (relatively) high-performance, reliable environments, and complete documentation. On the other hand, the specter of the new wave of languages arriving soon suggests that committing prematurely to a language now may result in regret down the road. My impression after the summer school is that if one has a "traditional" statistical modeling problem that does not involve nonparametrics, then Stan or BLOG is the way to go. For graphical models, Figaro and other graphical model-specific languages are probably the right approach. For everything else, namely Bayesian nonparametrics , I suggest either taking the plunge now with alpha-level software or waiting until the dust settles.

Programming Paradigm

Like regular programming languages, the question of programming paradigm takes center stage in PPSs. The full zoo of paradigms is already represented by existing PPSs: Figaro is object-oriented,

Venture, Church, and Hakaru are functional, while BLOG is logic-based. Having been rescued from the clutches of object-orientation by early exposure to Mathematica (now Wolfram Language), my inclination is strongly functional, and given the mathematical nature of the programs written in PPSs, I think the functional paradigm is particularly well suited. On the other hand, these are *modeling* languages, and object-orientation can make reasoning about different types of objects and their inter-relationships very natural. I suspect that this difference, perhaps more than any other, will make it possible for multiple languages to survive.

Beyond the programming paradigm, another important issue is whether a PPS exists as an independent programming language or as a domain-specific language (DSL) built on top of a general-purpose language. Here again there is a divide, one that happens to parallel whether the language has academic or industrial roots. Venture/Church, while being dialects of Scheme, are independent languages. Similarly for Haraku and BLOG. Figaro on the other hand is built on top of Scala, while [Ch/D]imple can be used within Matlab or Java.

This is a big distinction and dramatically alters the use cases of the language. The main advantage for independent languages is the cleanliness of language design afforded by the ability to completely rethink what a probabilistic programming language should be. The advantage of DSLs is their ability to leverage the richness of existing libraries of established programming languages. The integration is more than skin deep. Figaro allows any Scala object to be treated probabilistically, enabling some very advanced scenarios, e.g. the ability to reason and manipulate visual graphical objects in a scene probabilistically. The fact that industrial languages have opted for the DSL route is indicative of their more immediate practicality. Venture does try to bridge this gap through tight integration with Python, but its interface is far from seamless.

Use Case / Niche

All the above brings me to what in some sense really matters, and that is the PPSs target use case and niche. Depending on its objective, the capabilities that a PPS must achieve can vary a great deal. Without a clear target market, it is difficult to conjure up the appropriate language design, and to prioritize the features that need to be included.

Many use cases were proffered at the summer school. The biggest and most obvious is for domain experts—engineers, scientists, etc.—to explore and prototype specialized models. This may include machine learning researchers and advanced practitioners. The market for this area is not unlike that for scientific computing platforms such as Mathematica and Matlab, and just like these systems, certain problem sizes may be tractable without having to step outside the PPS.

Another set target specialized domains. Church for example is increasingly focused on cognitive science and psychology, and its team intends to push the boundaries of model expressiveness to capture increasingly subtle models, such as this one . Their focus is not on scale or efficient inference, not even model expressivity *per se* , but on a certain class of models that make use of self-referential behavior.

Then there are even more specialized use cases, such as the ability to communicate novel models in conference papers using formal probabilistic programs instead of the current mixture of prose, math,

and plate notation . Almost all groups mentioned this as a possibility. Naturally the extent to which any language can realistically assume this role will depend a great deal on its readability

Not surprisingly, the focus of any given language tended to correlate with its origin as an academic or commercial project. A language like Figaro, which is designed and commercialized by Charles River Analytics, is putting a strong emphasis on scalability and inference, and is already being used by paying customers. The fact that it may not push the envelope of language design is beside the point. It is also focused on playing nice with existing ecosystems, integrating with Scala and thus Java, and being generally accessible to industry users. Languages with academic origins like Venture are much less concerned, at least currently, on acquiring paying customers and are thus freer to explore bolder language design choices, at the expense of being practical tools that integrate well with a researcher's existing toolbox.

Challenges

There is a great deal of uncertainty in the field, which presents a challenge in its own way. However, there are also areas of clear difficulty that I see as the key hurdles to be overcome in the coming years.

General Inference

First is the question of how complex a program can get before inference becomes hopeless. We already know that inference in the arbitrarily general case is intractable. The more pertinent question is whether there is a class of problems that is large enough to warrant a programming language to describe it, yet small enough to be amenable to efficient inference. Clearly there are families of programs, for example Bayesian networks or hierarchical DPs (HDPs), that are useful and in which tractable inference is possible. However if the space of such programs is sufficiently fragmented, a high-level language may not be a terribly useful paradigm for specifying probabilistic programs, as they would all be effectively *ad hoc* and might as well be programmed from the ground up by the researchers who invent them. In some sense, the success of languages like Stan already dispels this notion, as they prove the existence of a useful space of structured probability distributions that can be specified programmatically. But Stan only addresses a small subspace of problems, and in particular for models involving discrete or infinite quantities, this remains very much an open question.

Scalability

Related to general inference but quite distinct from it is the question of scalability. This is not so much about whether a given class of programs is amenable to efficient inference. Rather, assuming a class of problems can be sampled efficiently, can its (generic) description in a programming language be reasoned about in an automated fashion to yield a sampling algorithm that is as efficient (at least asymptotically) as would be designed by a human programmer? Obviously in the limit of AI, the answer is yes, but in the near-term horizon of 1 to 5 years, this will be an important determinant of the success of PPSs. Languages like Figaro don't really count, because they have specialty constructs that deal with the "design patterns" of probabilistic programming. Similarly my experience with Venture so

far made it clear that a naïve generic implementation of an HDP, while correct, will not be sampled from efficiently. The problem is only interesting in the case of programs that are not obviously reducible to an existing construct, and thus the language designers can't cheat by looking up the right algorithm from a pre-existing library. David Blei's work on black box variational inference seemed to garner some hope in this area.

Inference Engines

Beyond the above, somewhat theoretical considerations, there is the well-defined task of building the requisite sampling algorithms. Almost all languages provide the basics (MCMC, Gibbs Sampling, etc), but there is a race of sorts to bake in more and more algorithms. Part of the challenge here is not overwhelming the programmer with an overabundance of algorithms, but instead developing intelligent heuristics that remove the guesswork out of this process. I haven't talked much about Stan (see Bonus), but it is one language that has taken an alternative approach, relying on one sampling approach, Hamiltonian Monte Carlo , to do all the heavy lifting. Such simplicity in design may prove prescient.

Language Design

What a probabilistic programming language should look like may end up being very, very different from what regular programming languages currently look like, and my impression is that few people, if any, really have an idea of how this will settle down. A comment made by one of the senior researchers working on a PPS was telling: he said that in regular programming languages, he can write one hundred lines of code and be fairly confident of its behavior. With a PPS, even ten lines of code can lead to programs whose behavior is entirely unpredictable. This has been my experience as well, and in some ways points to a general drawback of probabilistic programming: *it makes things too easy.* There is the old adage about certain programming languages handing the programmer enough rope to hang themselves. This phenomenon appears to be widely true of PPSs.

It may in part be due to the prevailing inexperience of all would-be probabilistic programmers, but it's also clear that PPSs present some fundamentally new challenges. I believe that well-designed languages will provide constructs for concisely writing probabilistic programs in a way that makes it possible to reason about the tradeoffs between complexity and tractability of said programs. Concomitantly, programs should be amenable to automated analysis by compilers, so that general inference becomes possible and the language is not merely reduced to a bag of prepackaged algorithms. Striking the balance between clarity, expressivity, and tractability is a major challenge of this field.

The Future

I will finish by prognosticating on the future of this space. For starters, I do think it is an incredibly exciting area, possibly even the future of machine learning and AI. Deep learning has momentarily sucked the air of the machine learning room—for good reason—and the slowness and generally poor performance of sampling-based approaches hasn't helped the cause of probabilistic programming either. But the richness of what can be achieved with probabilistic programming is only beginning to be

understood, and if / when major breakthroughs in variational and sampling approaches are made, this area will garner a lot more attention than it currently has. For now, the DARPA infusion seems to have really helped, and the program manager running the PPAML program at the time of the summer school (Kathleen Fisher) appears to have done a great job, from the original solicitation which was incredibly well thought-out to the current organization of the program.

For the near term, my suspicion is that the PPS space will experience a lot of growing pains. The right way forward remains unclear, and I don't believe anyone has an unassailable vision. This is not a fault, but merely a reflection of the early research state of the field. Too many things, most of which won't work, still have to be tried. I think that eventually useful PPSs will emerge, but none of the ones in the current crop may end up being the "winner". Many of the people currently involved in PPSs will likely be the same ones that make it work, but there will be a lot of shuffling and the existing languages will largely disappear. There's so much uncertainty right now that the first generation languages will invariably make multiple fatal mistakes. It will take multiple iterations to get this right, as it should.

Horizontal vs. Vertical Integration

One of the keys questions that will have to be addressed is that of horizontal vs. vertical integration. There are two aspects to this. One is whether a PPS is a general programming language that offers everything, including libraries for tooling, visualization, etc, or whether probabilistic constructs are merely added to existing languages. Second is whether a PPS provides the full stack of functionality needed for probabilistic programming, including an inference engine, or whether there will be a decoupling where probabilistic *languages* specify a language design, and pluggable inference engines can be used with different languages. Not being an expert in this field, my impression is that the jury is out, but there are a lot of subtle interactions between how the language is designed and how inference is carried out that makes this unknown territory. There have been several papers for example showing how probabilistic constructs can be added to regular programming languages , but it is unclear in practice whether this approach will ultimately yield a competitive (performance-wise) and elegant solution.

My suspicion is that there will be room for one or two languages that vertically integrate at least the PPS stack and that truly offer a leap in usability over standard programming languages. There is something to be said for this approach because functional PPS code in a language like Venture is very different from regular programs. A 10-line program can specify a very complex inference task, and likely the longest programs for the foreseeable future will be under 100 lines of code (the situation is different for an OO language like Figaro.) Hence thinking of probabilistic programming as regular programming with probability distributions is misguided. On the other hand, even the best PPSs are unlikely to compete with regular programming languages in terms of libraries and APIs, and so it is likely that they will be made to interface with other languages. More broadly, I expect most PPS solutions to adopt a horizontal approach, focusing on one aspect of the PPS stack. In part this is because designing good sampling algorithms is quite different from designing a good language, and the skills and talents of the people involved tend to be quite different. Furthermore, if more and more regular programming languages acquire probabilistic constructs, the value of plug-and-play inference

engines will only increase. The situation may end up being somewhat analogous to functional programming. There are a handful of elegant best-in-class functional languages that are great at what they do and integrate the full vertical stack—Mathematica is my favorite example. On the other hand, functional language constructs have been added and are continuing to be added to a wide array of languages. The winners, at least for the time being, seem to be the ugly jack-of-all-trades . One wrinkle to this discussion is the utility of integrating probabilistic constructs in general programming languages. So far, the applications have been somewhat limited, for example intuitive physics or generative computer vision . The question is whether adding "intelligence" to general purpose languages is something that is broadly useful. If it is, then I think the above considerations apply.

Inference

How to do inference and the relationship of the inference engine vis-à-vis the programming language *and the programmer* is another interesting question. Incidentally, the PPAML summer school lectures were set up in a way that presumes the programmer does not need to be exposed to the underlying inference engine. I think this was a mistake, given that these are early days, and the students would have benefited from better exposure to the guts of PPS systems. More fundamentally, I suspect that exposing the inference procedure via high-level sampling primitives will be the key to making probabilistic programming work, at least in the short term. There is no free lunch, generally speaking, but what does exist is the possibility of exposing the important knobs to the aspiring probabilistic programmer. What inference programming enables is the ability for a programmer to exploit her understanding of the specifics of her probabilistic program, such as conjugacy relationships, exchangeability properties, and other collapsible aspects of the model, in an easily iterable fashion to try out different sampling strategies. Just as probabilistic programming promises to make the exploration of different probabilistic models accessible, inference programming may make it possible to explore the space of sampling strategies more easily, which in turn can result in efficient sampling procedures of specialized probabilistic programs. This will be especially true if the underlying sampling primitives that a PPS exposes are themselves implemented efficiently. Hoping for anything more, i.e. for a PPS to figure out everything automatically including the most optimal sampling strategy, is probably too much to ask for, for the time being. But if PPSs are able to expose inference programming in an accessible fashion, then they do not need to solve the broader and more difficult problem in a single swoop. They would provide immediate value to statisticians and machine learning practitioners today.

Want To Dabble?

If this discussion got you curious about probabilistic programming, I suggest you give webppl a try, which requires very little upfront investment to get started.

Bonus (Stan)

Stan, the language developed and maintained by Andrew Gelman 's group at Columbia, was not represented at the PPAML summer school. Fortunately I caught a talk about Stan by Bob Carpenter at

the Open Machine Learning Workshop at MSR in New York. Here are some of my brief thoughts. I should note that I have not personally used Stan (yet!).

Unlike most languages at PPAML, Stan is increasingly a mature platform used by statisticians and data scientists to do real-world modeling. It is maintained by something like a dozen full-time staff members, and more closely resembles a small professional software team than an academic project run by a handful of graduate students (which is an accurate description of most of the other projects). For practical modeling problems that do not involve discrete variables or variable-sized models, Stan is the PPS to beat. In many ways it has already validated the PPS space by virtue of being widely used in real-world contexts.

The caveat to all the above are the words “discrete” and “variable-sized”. The most glaring omission from the Stan toolkit is support for discrete random variables. Interestingly, at the PPAML summer school this was spun by some of Stan’s competitors as a “philosophical” issue, i.e. that Stan’s creators do not perceive discrete random variables to be a meaningful construct for real-world applications. Bob Carpenter dispelled this notion, stating that it’s simply a practical limitation of the way Stan currently does sampling (presumably due to its reliance on HMC). Unfortunately, all indications point to this being a rather fundamental design limitation, and so I am not holding my breath that the problem will be fixed soon. Stan 3 will not have support for discrete variables.

The lack of support for discrete variables also implies the inability to handle variable-sized models like DPs (this is true for even finite variable-sized models), which are all the rage in Bayesian nonparametrics. Regardless of where one stands on the utility of Bayesian nonparametrics (I think they’re very important), the fact that Stan is unable to address the class of models of most interest to ML researchers means its primary target demographic will be limited to ML practitioners and data scientists. This is of course an important demographic, and is the area where Stan has found a lot of success, but it does limit its potential as a future platform for probabilistic programming. It will be interesting to see if Stan is able to overcome its limitations and become the de facto system, or if one of the new PPAML-sponsored languages can mature enough to become a serious competitor to Stan. Regardless, competition is good for the field, and I look forward to seeing it play out.