A Final Word

In this lesson, we will first summarize all the concepts we learned, then describe a whole lot of interesting stuff that we did not get to and lastly, mention a few reference papers and further reading that inspired this course.

WE'LL COVER THE FOLLOWING

All right, let's finish this course off!

Goal of the Course

If you come away with nothing else from this course, the key message is: probabilistic programming is essential, it is too hard today, and we can do a lot better than we are doing. We need to build better tools that leverage the type system and support line-of-business programmers who need to do probabilistic work, the same way that we built better tools for programmers who needed to use sequences, or databases, or asynchrony.

Course Summary

- We started this course with us complaining about `System.Random`, hence the name of the course. Even though some of the implementation details have finally improved after only some decades of misleading developers, we are still dealing with random numbers like it was 1972.

- The abstraction that we're missing is to make "value drawn from a random distribution" a part of the type system, the same way that "function that returns a value", or "sequence of values", or "value that

might be null" is part of the type system.

- The type we want is something like a sequence enumeration, but instead of a "move next" operation, we've got a "sample" operation.
- If we stick to simple distributions where the support is finite and small and the "weights" are integers, and restrict ourselves to pure functions, we can create new distributions from old ones using the same operations that we use to create new sequences from old ones: the LINQ operators `Select`, `SelectMany` and `Where`.
- Moreover, we can compute distributions exactly at runtime, without doing rejection sampling or other clever techniques.
- And we can even use query comprehension syntax, which is a pleasant and interesting way to write a probabilistic workflow.
- From this we can see that probability distributions are monads; $P(A)$ is just `IDistribution<A>`.
- We also see that conditional probabilities $P(B \; given \; A)$ are just `Func<A, IDistribution<B>>` — they are likelihood functions.
- The `SelectMany` operation on the probability monad lets us combine a likelihood function with a prior probability.
- **The `Where` operation lets us compute a posterior given a prior, a likelihood and an observation**.
- This is a beneficial result; even domain experts like doctors often do not have good intuitions about how an observation should change our opinions about the probability of an event has occurred. A positive result on a test of a rare disease may be only weak evidence if the test failure rate is close to the disease rate.
- Can we put these features in the language as well as the type system? Abusing LINQ is clever but maybe not the best from a usability perspective.
- We could embed `sample` and `condition` operators in the C# language, just as we embedded `await`. We can then write an imperative workflow, and

have the compiler generate a method that returns the correct probability

distribution, just as `await` allows us to write an asynchronous workflow and the compiler generates a method that returns a task!

- Unfortunately, real-world probability problems are seldom discrete, small, and completely analyzable; they're more often continuous and approximated.
- Implementing `sample` efficiently on arbitrary distributions turns out to be a hard problem.
- We can use our tools to generate Markov chains.
- We can use Markov chains to implement `sample` using the Metropolis Algorithm.
- If we have a continuous prior (like a mint that produces coins of a certain quality with a certain probability) and a discrete likelihood (like the probability of a coin flip coming up heads), we can use this technique to compute a continuous posterior given an observation of one or more flips.
- This is a very useful technique in many real-world applications.
- Computing, the expected value of a distribution given a function, is a tricky problem in of itself, but there are a variety of techniques we can use.
- And if we can do that, we can solve some high-dimensional integral calculus problems!

Connection between `await` and `sample`

By far the biggest thing that we did not get to, and that we may return to in another course is: **the connection between `await` as an operator and `sample` as an operator is deeper than you might think**.

As we noted above, you can put `sample` and `condition` operations in a language and have the compiler build a method that when run, generates a simple, discrete distribution. But it turns out **we can do a pretty good job of**

dealing with sample operators on non-discrete distributions as well, by

having the compiler be smart about using some of the techniques that we discussed in this course for continuous distributions.

What you really need is the ability to pick a sample, run a little bit of a routine, remember the result, back up a bit and try a different sample, and so on; from this **we can build a distribution of program traces**, and from that we can build an approximation of the distribution of the output of a probabilistic method!

This kind of control flow is tricky; it's sort of a generalization of coroutines where you're allowed to re-run code that you've run before, but with different values for the variables to see what happens.

It is crucially important that the methods be pure! It's also crucially important that you spend most of your time exploring high-likelihood control flows because the number of unlikely control flows is nigh-infinite. If this sounds a bit like training up an ML model and then using that model in production later, that's because it is the same thing, but applied to programs.

References

Finally, here's an incomplete list of papers and web sites that inspired writing this course.

- The idea that we can treat probability distributions like LINQ queries was given by **Erik Meijer**; his fun and accessible paper Making Money Using Math hits many of the same points we did in this course, but we did it with a lot more code.
- The design of coroutines in Kotlin was a great inspiration to us; they've done a great job of making features that you would normally think of as being part of the language proper, like `yield` and `await`, into library methods.
- An introduction to probabilistic programming is a book-length work that uses a Lisp-like language with samples and observes primitives.
- Church is another Lisp-like language often used in academic studies of

PPLs.

- The webppl.org web site has a Javascript-based implementation of a probabilistic programming language and a lot of great supporting information at dippl.org.
- HackPPL implements many of these ideas; Hack is a statically-typed variant of PHP.

The next few papers are more technical.

- Build Your Own Probability Monads is a good overview for the Haskell programmers out there, as is Practical Probabilistic Programming With Monads and Stochastic Lambda Calculus and Monads of Probability Distributions.
- Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation is a good overview of how you can use MCMC techniques on program traces. A provably correct sampler for probabilistic programs goes into some of the correctness problems faced by PPL implementations, and Generating Efficient MCMC Kernels goes into some of the performance problems.
- Another fascinating area that we wanted to explore in this course is: we know it is bad enough to try to debug programs that have yields and awaits in them; how on earth do you debug programs that are actually running the same code paths maybe thousands of times when exploring the sample space of a workflow? Here's an interesting paper on debugging probabilistic workflows.

And these are some good ones for the math:

- These MIT course notes on Bayesian updating of continuous priors was a good primer for our lessons on "coin-flipping". And the lecture notes MC Methods and Importance Sampling are what it says on the tin.
- Understanding the Metropolis-Hastings Algorithm gives a bunch of the underlying math.
- The enormous book Information Theory, Inference and Learning Algorithms was invaluable in getting up to speed on math.