Musings about Language Design

In this lesson we've got no code for you; instead, here are some musings about language design informed by our discussion so far.

WE'LL COVER THE FOLLOWING

Balance Between Generality and Specificity of a Feature

One of the most important questions to ask when designing a language feature is: what should the balance be between generality and specificity of a feature?

Our go-to example is LINQ, and what we've been talking about for the last lessons illustrates the problem nicely.

As we've seen many times in this course, query comprehensions allow us to express projection (`Select`) concisely, binding (`SelectMany`) and use of the zero of an additive monad (`Where`) to express a condition. Because C# uses a syntactic transformation to implement LINQ, we can show these concepts on an instance of any monad that has the right methods. The compiler does not know or care whether the thing being selected is a sequence, a probability distribution, whatever, doesn't matter. All that matters is that a method with the right signature is available as an instance or extension method.

The language designers for C# 3.0 were therefore faced with a choice: how general should the LINQ feature be? There's a spectrum of possibilities; a few points on that spectrum are:

- Monad is just a *monoid* in the category of *endofunctors*, which implies that there are other patterns like “monoid” and “category” and “functor” that we could describe in the *type system*; what’s so special about monads? We could create a whole “higher-kinded” type system to describe generic type patterns; “monad” is just one possible pattern.

- We could embbed monads as a special kind of thing right into the language, using the vocabulary of monads: bind, unit and so on. In this language, sequences are just a special case of the more general monad feature.

- The feature could be made specifically about operations on sequences: not just `Select`, `SelectMany` and `Where` which make sense on multiple monads, but also operations that do not apply across monads, like `GroupBy`, `Join` and `OrderBy`. The operations could apply to any data source that supports those operations, whether XML documents or database tables or plain old lists. In this variation, the concepts in the API are tailored specifically to fit the domain of sequences of data.

- The feature could be narrowly tailored to the needs of a particular data technology; LINQ could have been merely allowing SQL Server query syntax to be embedded directly in the language, and it could have been made to work only with SQL Server and no other dialects of SQL or other databases back ends.

The designers of C# chose a point on the spectrum slightly more general than the third point: the feature is not written in the jargon of monads, but it is more general than simple operations on sequences. The designers of *Haskell* chose maximum generality, and so on.

Other Options

These are, of course, not the only options; there were many numbers of points on that spectrum where the language design could have fallen, and different language designers have chosen different points. Think about list comprehensions in Python, for instance; they are similar to LINQ in many respects, but you’re not going to be marshaling a list comprehension to a database or using it as a generalized monadic bind operation any time soon.

There's nothing wrong with that; it was a reasonable choice for Python.

Our exploration of representing distributions as a monad and combining them using LINQ operators and comprehensions illustrates the pros and cons of C#'s choice. Though I think it is delightful that we can use Select, SelectMany and Where on probability distributions as though they were infinite sequences, and pleased that we can optimize the results using the laws of probability for discrete distributions, the language designer part of me feels like this is somehow wrong. We're using the vocabulary of sequences, not the vocabulary of probability, and it feels out of place.

Revisiting the Monty Hall Problem

This, in particular, was brought into focus by our Monty Hall Problem. Though it was in parts deliberately obtuse and misleading, there was no real joke there; instead, we wanted to illustrate a serious point with a silly example. Consider these two queries:

```
from car in doors
from you in doors
from monty in (
    from m in doors
    where m != car
    where m != you
    select m)
select (car, you, monty)
```

and

```
from car in doors
from you in doors
from monty in doors
where monty != car
where monty != you
select (car, you, monty)
```

We can refactor the first into the second if `doors` is a sequence; this program transformation preserves semantics for sequences. But if `doors` is a *distribution* then the refactoring preserves the support but changes the weights.

This is some evidence that query comprehensions are perhaps not the best

This is some evidence that query comprehensions are perhaps not the best way to express operations on distributions: our intuitions about what

refactorings are correct, is heavily influenced by our understanding of sequence semantics, and this could lead to bugs.

Design of C#

Moreover, think about how the design of C# has evolved with respect to monadic types:

- C# 2 added nullable types and embedded nullable arithmetic and conversions in the language, but did not provide a general mechanism for lifting function calls and other operations to nullables until C# 6 added.
- C# 2 added statement-based sequence workflows in the form of yield return.
- C# 3 added query comprehensions for composing operations on sequences in an expression context.
- *Reactive extensions* `(Rx)` leveraged LINQ to support observables — push-based sequences — but did not add any new syntax; it could do this because the duality between pull- and push-based sequences are strong and our intuitions largely still give correct outcomes.
- C# 5 added statement-based asynchronous workflows; though you can make async lambdas and use the await operator in the middle of an expression, asynchronous workflows are fundamentally collections of statements, not fluent compositions of expressions like query comprehensions.

You'll notice that there is no overarching philosophy here; instead, as the needs of real-world developers evolve, the language evolves to represent different concepts at different levels of generality.

Async workflows could have been made to look just like `Rx` query expressions; after all, a task is logically an observable that pushes a single value instead of multiple values. But the designers of C# figured that it would be more natural for developers to mix asynchrony into their statement-based workflows. Similarly, nullables could be treated as sequences that contain zero or one value, but they’re not.

Here is the lesson for C#: when you’re trying to represent a new concept, try creating a domain-specific subset of the language that solves the problem using the vocabulary of the problem, but with enough generality to be extensible in the future.

Could we do the same for stochastic workflows? Using LINQ as our representation has been fun and educational, but as language designers, we can recognize that real-world users would likely find this choice confusing and weird. Distributions are like sequences in many ways, but they are also dissimilar in a lot of ways.

Thought Process

Can we come up with a domain-specific language subset that better matches the domain of probabilistic programming, but preserves (or improves upon) the nice properties that we’ve seen so far? We’ve noticed that we can express a discrete-distribution workflow and automatically get an inferred distribution out the other end; could we do the same thing in a statement-based workflow language, akin to async/await?

In the next lesson, we are going to sketch out a super-stripped down version of C# and add two new statements for probabilistic workflows. We’ll then show how that stripped-down version could be lowered to ordinary C# 7.