Discrete Probability Distribution Type as a Monad

In this lesson, we will learn about additive monads.

WE'LL COVER THE FOLLOWING

- Additive Monad
- Implementation

Before we get going on this lesson, you might want to refresh your memory of what an **additive monad** is.

Additive Monad

Briefly, an additive monad is a monad where there is a *"zero value"*; like the number zero, "multiplying" by zero produces a zero, and "adding" a zero is an identity.

For example, the sequence monad, `IEnumerable<T>`, has a zero: the empty sequence. If we `Select` or `SelectMany` from the empty sequence — our analog of "multiplying" — we get an empty sequence. If we concatenate an empty sequence onto another sequence — the sequence analog of "adding" — we get the original sequence.

All additive monads can have a `Where` function defined on them; if we wanted to implement `Where` for sequences and didn't care about performance, we could implement it like this:

```
public static IEnumerable<T> Single<T>(T t)
{
  yield return t;
}
public static IEnumerable<T> Zero<T>()
{
  yield break;
```

```
}

// Non-standard Where:
public static IEnumerable<T> Where<T>(this IEnumerable<T> items, Func<T, b
ool> predicate) =>
  from a in items
  from b in predicate(a) ? Single(a) : Zero<T>()
  select b;
```

That's slow and produces hideous collection pressure, but it works; our actual implementation of `Where` is just an optimization.

What about the converse? Our probability monad `IDiscreteDistribution<T>` has a `Where` function defined. We have a `Singleton<T>` type. But our implementation of the distribution monad does not appear to have a zero value. It seems plausible that there should be a way to express `Where` on distributions as we did with the sequence monad: as a `SelectMany` that produces either the single or zero distributions based on the predicate.

Give that some thought, and then look at the answer.

Show Hint

We never implemented this value because every distribution class we've created already throws when you try to create an empty distribution:

1. `StandardDiscreteInteger` throws if the range is empty.
2. `Bernoulli` and `WeightedInteger` both throw if you give them all zero weights.

In our current implementation a `Where` clause, where the predicate is false for everything in the support of the underlying collection, will eventually throw.

In our original implementation, a `Where` clause where the predicate is always false hangs when sampled, but does not throw. Our implementation of `Select` throws if the support is empty.

We have learned the following facts:

1. The zero value of the discrete distribution monad is the empty distribution.

2. The joint distribution produced by `SelectMany` is the analog of multiplication of two distributions.
3. Concatenation is the "addition" of the sequence monad. (The two sequences have to be of the same element type.)

Doing a `SelectMany` on an empty distribution has to produce an empty distribution. But we still have a mystery to solve: what is the addition operator on two discrete distributions? They have to be of the same element type. The addition operator has to have the property that adding zero to any distribution is an identity, but what does it mean to add together two non-zero distributions?

It turns out that there are some uses for an explicit empty distribution; we'll discover what the specific benefits of it are in a later lesson.

What are the costs? We don't mean implementation costs, but rather, what are the downsides to developers of having this feature? In short: if we go down this road, what new opportunities for bugs are we producing?

One interesting cost is that we will defer an operation that can throw; this can be very confusing! A classic source of Stack Overflow questions is when someone writes an enumerator block:

```
static IEnumerable<int> Foo(string bar)
{
  if (bar == null)
    throw new ArgumentNullException();
  yield return bar.Length;
  ...
}
```

and then calls it:

```
var foo = Foo(accidentallyNullThing); // no throw
[...]
foreach (int x in foo) // throw!
```

The source of the problem is that the throw is delayed. If you look at the proper, industrial-strength implementations of `Where`, `Select` and so on,

you'll notice that each one is written in a style where it validates its

arguments first, and then returns a call to a helper method that does the iteration. That way the exception is thrown close to the point of the mistake.

However, that doesn't fix other common variations on the problem. For example, you might have some buggy code that produces an empty sequence sometimes, and then a thousand lines later you call first on the sequence and it blows up, but the bug is where the sequence is produced.

And of course this is really no different than nullable types that blow up when we forget that they can be null; a nullable `T` is logically a sequence of `T` where the sequence length is either zero or one, and if we forget that it can be "zero-length", we get into trouble.

The empty distribution will have the same property: it will be easy to create it by accident in a buggy program and it will not blow up until it is sampled, just as nullable reference types do not blow up until they are dereferenced.

That being said, we're going to do it because the benefits are pretty compelling, oddly enough.

Implementation

The code for this lesson is as follows:

Program.cs

Bernoulli.cs

BetterRandom.cs

Combined.cs

Distribution.cs

Episode17.cs

Extensions.cs

IDiscreteDistribution.cs

IDistribution.cs

Projected.cs

Pseudorandom.cs

Singleton.cs

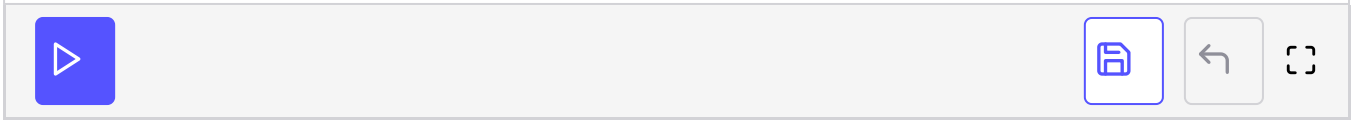
StandardCont.cs

StandardDiscrete.cs

WeightedInteger.cs

```
using System;
using System.Collections.Generic;

// Let's show how we can create Where out of SelectMany on sequences.
namespace Weird
{
    static class MyLinq
    {
        // Standard implementation of Select:
        public static IEnumerable<R> Select<A, R>(
            this IEnumerable<A> items,
            Func<A, R> projection)
        {
            foreach (A item in items)
                yield return projection(item);
        }

        // Standard implementation of SelectMany:
        public static IEnumerable<R> SelectMany<A, B, R>(
            this IEnumerable<A> items,
            Func<A, IEnumerable<B>> selection,
            Func<A, B, R> projection)
        {
            foreach (A a in items)
                foreach (B b in selection(a))
                    yield return projection(a, b);
        }

        public static IEnumerable<T> Single<T>(T t)
        {
            yield return t;
        }

        public static IEnumerable<T> Zero<T>()
        {
            yield break;
        }

        // Non-standard Where:
        public static IEnumerable<T> Where<T>(
                this IEnumerable<T> items,
                Func<T, bool> predicate) =>
            from a in items
            from b in predicate(a) ? Single(a) : Zero<T>()
            select b;
    }
}

namespace Probability
{
```

```
    // No using System.Linq.
    using Weird;
    static class Episode17
    {
        public static void DoIt()
        {
            Console.WriteLine("Episode 17");
            Console.WriteLine("Custom Where using only SelectMany");
            Console.WriteLine("aBcDe".Where(char.IsLower).CommaSeparated());
            Console.WriteLine("Delayed throw in enumerator block");
            var foo = Foo(null);
            Console.WriteLine("No throw yet!");
            try
            {
                foreach (int x in foo)
                    Console.WriteLine(x);
            }
            catch
            {
                Console.WriteLine("Now we throw.");
            }
        }

        static IEnumerable<int> Foo(string bar)
        {
            if (bar == null)
                throw new ArgumentNullException();
            yield return bar.Length;
        }
    }
}
```



An implementation of the empty distribution is straightforward, but it will require fixing up some of our existing code to use it. Before we get to that though, let's test your understanding of what we've seen so far by looking at a famously counterintuitive problem in probabilistic reasoning.