Proposed Stripped-Down C#

In this lesson we are going to propose a stripped-down version of C# and introduce a new operator.

WE'LL COVER THE FOLLOWING

Stripped Down C#

Without further ado, here is our proposed stripped-down C# that could be a domain-specific language (DSL) for `probabilistic` workflows; as we'll see, it is quite similar to both enumerator blocks from C# 2, and `async`/`await` from C# 5.

Workflow

- The workflow must be a function with a `return` type of `IDiscreteDistribution<T>` for some `T`.
- Just as methods that use `await` must be marked `async`, our DSL methods must be marked `probabilistic`.
- The function has an ordinary function body: a block of statements.

We are using C# 5 asynchronous workflows as an example of how to add a new mechanism to C#. The statements and the expressions in a `probabilistic` method are restricted as follows:

1. Declaration statements must have an initializer.
2. All the locals must have unique names throughout the method.

3. Assignments can only be in declarations or as statements; no use of assignments in the middle of an expression is allowed.

4. No compound assignments, increments or decrements.

5. No `await`, `yield`, `try`, `throw`, `switch`, `while`, `do`, `break`, `continue`, `goto`, `fixed`, `lock`, `using` or labeled statements allowed. What's left? `if` and `return` statements are fine, as well as blocks `{ }`. We told you we are stripping things down!

6. No `lambdas`, anonymous functions, or query comprehensions.

7. No use of `in`, `out` or `ref`.

8. All the other perfectly normal operations are allowed — function calls, object initializers, member access, indexers, arithmetic, that's all just fine.

9. However, all function calls and whatnot must be pure; there can be no side effects, and nothing should throw.

Basically what we are doing here is making a little super-simple subset of C# that doesn't have any of the weird, hard stuff that keeps compiler writers busy. In this pleasant world, locals are always assigned, there are no closures, there are no worries about what happens when an exception is thrown and all that sort of stuff.

We'll sketch out how to soften some of those restrictions in later lessons. We want to show that we can describe how to implement a simple core of a language; harder features can come later.

New Unary Operator and Statement

To this little language, we are going to add a new unary operator and a new statement. The new operator is `sample`, and it may appear only as of the right operand of an assignment, or the initializer of a declaration:

```
int x = sample WeightedInteger.Distribution(8, 2, 7, 3);
```

The operand must be of type `IDiscreteDistribution<T>`, and the type of the

expression is `T`. Again, this should feel familiar if you're used to `await`.

The new statement is a condition, and it has the form

```
condition expression;
```

The expression must be convertible to `bool`. The meaning of this thing is much the same as `Where`: if the Boolean condition is not met, then a value is filtered out from the distribution by setting its weight to zero.

> Aside: In the previous lesson we learned a bit about how language designers must choose how general or specific a language element is, and that this must be reflected in syntax; this is an excellent example of such a choice.

We have chosen `condition` because we can think of this operation as creating a conditional distribution; we could have said `where` instead of `condition` and had it mean basically the same thing, but that would be moving towards the established C# jargon for sequences, which we are explicitly trying to avoid.

However, as we've seen, a primary usage case for a conditional distribution is computing the posterior distribution when given a prior and a likelihood. But why do we wish to compute the posterior at all? Typically because we have observed something. That is, the development cycle of the `probabilistic` program is:

- Before the program is written, data scientists and developers compute priors, like "What percentage of emails are spam?"
- They also compute likelihood functions: "what word combinations are likely, given that an email is spam? What word combinations are likely given that an email is not spam?"
- A spam detection program must now answer the question: given that we have observed an email with certain word combinations, what is the posterior probability that it is spam? This is where the `probabilistic` workflow comes in. For that reason, we could reasonably choose `observation` or `observe` as our keyword here, instead of `condition`, emphasizing to the developer "the reason you use this feature is to

express the computation of a posterior distribution for a given observation." That would make the feature feel a little bit less general but might help more clearly express the desired semantics in the mind of the typical programmer designing a workflow that computes a posterior.

We are going to stick with `condition`, but let's point out that this is a choice that has user experience consequences, were we actually to implement this feature in a language like C#.

Let's look at some examples:

```
probabilistic IDiscreteDistribution<bool> Flip()
{
  int x = sample Bernoulli.Distribution(1, 1);
  return x == 0;
}
```

What we would like is for this method to have the same semantics as if we had written:

```
 IDiscreteDistribution<bool> Flip() =>
   Bernoulli.Distribution(1, 1).Select(x => x == 0);
```

What do you think about these two implementations? The former looks a lot more like the straightforward, imperative code that we're used to.

Let's look at another:

```
probabilistic IDiscreteDistribution<int> TwoDSix()
{
  var d = SDU.Distribution(1, 6);
  int x = sample d;
  int y = sample d;
  return x + y;
}
```

Again, it seems that this is more clear than

```
IDiscreteDistribution<int> TwoDSix()
{
  var d = SDU.Distribution(1, 6);
  return from x in d from y in d select x + y;
}
```

And it certainly seems more evident, particularly to the novice, than

```
IDiscreteDistribution<int> TwoDSix()
{
  var d = SDU.Distribution(1, 6);
  return d.SelectMany(x => d, (x, y) => x + y);
}
```

LINQ is incredible for sequences, but the statement-based workflow is much easier to understand for distributions.

Now let's look at a complicated workflow, this one with conditioning:

```
probabilistic IDiscreteDistribution<string> Workflow(int z)
{
  int ii = sample TwoDSix();
  if (ii == 2)
    return "two";
  condition ii != z;
  bool b = sample Flip();
  return b ? "heads" : ii.ToString();
}
```

The first two were easy to see how they corresponded to query syntax, but what even is the distribution represented by this workflow? It depends on the value of the parameter `z`, for one thing.

What we want here is: when you call this method, you get a distribution back. When you `Sample()` that distribution, it should logically have the same effect as all the sample operations `Sample()` their operand, and the returned value is, well, the returned value. However, if any condition is false, then we abandon the current attempt and start over.

The trick is implementing those semantics without actually running the body in a loop!

> **Exercise:** Pick a value for `z`; let's say 3. See if you can work out what the distribution of strings is that should come out the other end of this thing. We will provide the answer to this exercise in the next lesson.

> **Exercise:** If you had to represent this workflow as a query, how would you do it?

Implementation

The code snippet below is non-executable because this is our very own proposed implementation. We will make it executable in the next lesson.

Program.cs
Bernoulli.cs
BetterRandom.cs
Combined.cs
Distribution.cs
Empty.cs
Episode20.cs
Extensions.cs
IDiscreteDistribution.cs
IDistribution.cs
Projected.cs
Pseudorandom.cs
Singleton.cs
StandardCont.cs
StandardDiscrete.cs

```
using System;
namespace Probability
{
    using SDU = StandardDiscreteUniform;
    static class Episode20
    {
#if false
        probabilistic static IDiscreteDistribution<bool> Flip()
        {
```

```
            int x = sample Bernoulli.Distribution(1, 1);
            return x == 0;
        }

#endif
        // The above should have the same semantics as:

        static IDiscreteDistribution<bool> Flip() =>
            Bernoulli.Distribution(1, 1).Select(x => x == 0);

#if false

        probabilistic static IDiscreteDistribution<int> TwoDSix()
        {
            var d = SDU.Distribution(1, 6);
            int x = sample d;
            int y = sample d;
            return x + y;
        }
#endif
        // The above should have the same semantics as:

        static IDiscreteDistribution<int> TwoDSix()
        {
            var d = SDU.Distribution(1, 6);
            return d.SelectMany(x => d, (x, y) => x + y);
        }

#if false

        probabilistic static IDiscreteDistribution<string> DoIt(int z)
        {
            int ii = sample TwoDSix();
            if (ii == 2) return "two";
            condition ii != z;
            bool b = sample Flip();
            return b ? "heads" : ii.ToString();
        }
#endif
        // We'll see what this has the same semantics as in Episode 21.

    }
}
```

Suppose we were writing a compiler for this feature. We know that LINQ works by turning query comprehensions into function calls; we know that the compiler turns `async` workflows and iterator blocks build state-machine coroutines. How might we lower this new kind of code into ordinary C# 7 code? Probably somehow using our existing query operators but what exactly would work? Let's have a look at this in the next lesson.