Converting to Ordinary C# 7

In this lesson, we will convert the code from the previous lesson, to ordinary C# 7.

WE'LL COVER THE FOLLOWING

In the previous lesson, we proposed a stripped-down DSL for probabilistic workflows. In this lesson, let's see how we could "lower" it to ordinary C# 7 code. We will assume that we have all of the types and extension methods that we've developed so far.

Naive Approach

The first thing I'm going to do is describe a possible but very problematic way to do it.

We know how C# lowers methods that have `yield` return statements: it generates a class that implements `IEnumerable`, rewrites the method body as the `MoveNext` method of that class, and makes the method return an instance of the class. We could do the same thing here.

Recall that last time we gave three examples, the longest of which was:

```
probabilistic static 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();
}
```

We could lower this to:

```
static IDiscreteDistribution<string> Workflow(int z) =>
  new WorkflowDistribution(z);

sealed class WorkflowDistribution :
  IDiscreteDistribution<string>
{
  int z;
  public WorkflowDistribution(int z)
  {
    this.z = z;
  }
  public string Sample()
  {
    start:
    int ii = TwoDSix().Sample();
    if (ii == 2)
      return "two";
    if (!(ii != z))
      goto start;
    bool b = Flip().Sample();
    return b ? "heads" : ii.ToString();
  }
  public IEnumerable<string> Support() => ???
  public int Weight(string t) => ???
}
```

We'd need to make sure that we did the right thing if `z` got modified during the workflow and the condition was not met of course, but that's a small detail.

We could similarly lower the other two.

The good news is: we get the right distribution out of this thing:

```
Console.WriteLine(Workflow(3).Histogram());
```

```
    4|***
```

```
    5|****
    6|*****
    7|*******
    8|******
    9|****
   10|***
   11|**
   12|*
  two|**
heads|************************************
```

The bad news is:

- We haven't computed the support or the weights, as required.
- We are once again in a situation where a `condition` causes a potentially long-running loop to execute; we could do hundreds or thousands of iterations of going back to "start" for every successful sample if the `condition` was rarely met.

Basically, we've lost the great property that we had before: that we automatically get the right discrete distribution object inferred.

Our Goal

So let's state unequivocally right now what our goal here is.

> The goal of this feature is to take a method body that describes a probabilistic workflow that includes conditions and produces a semantically equivalent distribution that does not actually contain any loop-causing condition, the same as we did for query-based workflows.

In our exploration of creating a posterior, we've seen that we can take a query that contains `Where` clauses and produce a projected weighted integer distribution that matches the desired distribution. Though there is some cost to producing that distribution object, once we've paid that cost there is no additional per-sample cost. We can do the same here, by leveraging the work we've already done.

Improved Approach

The technique we are going to use is as follows:

1. Each statement in the method will be numbered starting from zero; statements nested inside `if` statements are of course statements.
2. Each statement will be replaced by a local method named `Sn`, for `n` the number of the statement.
3. Each local method will take as its arguments all the locals of the method. (If there are any assignments to formal parameters then they will be considered locals for this purpose.)
4. Each local method will return a probability distribution.

At this point, we could give the general rule for each kind of statement in our stripped-down language, but let's not bother. Let's just lower the methods and it will become obvious what we're doing.

We'll start with the easy one.

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

we rewrite this as:

```
static IDiscreteDistribution<bool> Flip()
{
    Func<int, IDiscreteDistribution<bool>> S1 = x =>
        Singleton<bool>.Distribution(x == 0);
    Func<int, IDiscreteDistribution<bool>> S0 = x =>
        Bernoulli.Distribution(1, 1).SelectMany(_x => S1(_x));
    return S0(0);
}
```

Read that over carefully and convince yourself that this implements the correct logic, just in a much more convoluted fashion.

> Notice that we are taking advantage of the monad laws here regarding the relationship between binding and the function that produces a `Singleton`. It might not have been clear before why this is an important

> monad law; hopefully, it is now clear.

> Recall that in the jargon of monads, the unit operation that creates a new wrapper around a single instance of the underlying type is sometimes called return. It is no coincidence that our lowering turns return statements into singletons!

Now let's lower the next one:

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

This becomes this mess:

```
static IDiscreteDistribution<int> TwoDSix()
{
    Func<IDiscreteDistribution<int>, int, int, IDiscreteDistribution<int>>
 S3 = (d, x, y) =>
        Singleton<int>.Distribution(x + y);
    Func<IDiscreteDistribution<int>, int, int, IDiscreteDistribution<int>>
 S2 = (d, x, y) =>
        d.SelectMany(_y => S3(d, x, _y));
    Func<IDiscreteDistribution<int>, int, int, IDiscreteDistribution<int>>
 S1 = (d, x, y) =>
        d.SelectMany(_x => S2(d, _x, y));
    Func<IDiscreteDistribution<int>, int, int, IDiscreteDistribution<int>>
 S0 = (d, x, y) =>
        S1(StandardDiscreteUniform.Distribution(1, 6), x, y);
    return S0(null, 0, 0);
}
```

Again, walk through that and convince yourself that it's doing the right thing. This implementation is ridiculously overcomplicated for what it does, but you should have confidence that it is doing the right thing. Remember, we're trying to prove that the proposed language feature is in theory possible first, and then we'll think about ways to make it better.

Finally, let's do one with an `if` and a `condition`:

```
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();
}
```

This lowers to:

```
static IDiscreteDistribution<string> Workflow(int z)
{
  Func<int, bool, IDiscreteDistribution<string>> S5 = (ii, b) =>
    Singleton<string>.Distribution(b ? "heads" : ii.ToString());
  Func<int, bool, IDiscreteDistribution<string>> S4 = (ii, b) =>
    Flip().SelectMany(_b => S5(ii, _b));
  Func<int, bool, IDiscreteDistribution<string>> S3 = (ii, b) =>
    ii != z ? S4(ii, b) : Empty<string>.Distribution;
  Func<int, bool, IDiscreteDistribution<string>> S2 = (ii, b) =>
    Singleton<string>.Distribution("two");
  Func<int, bool, IDiscreteDistribution<string>> S1 = (ii, b) =>
    ii == 2 ? S2(ii, b) : S3(ii, b);
  Func<int, bool, IDiscreteDistribution<string>> S0 = (ii, b) =>
    TwoDSix().SelectMany(_ii => S1(_ii, b));
  return S0(0, false);
}
```

And at last we can find out what the distribution of this workflow is:

```
Console.WriteLine(Workflow(3).ShowWeights());
```

gives us:

```
  two:2
heads:33
    4:3
    5:4
    6:5
    7:6
    8:5
```

```
 9:4
10:3

11:2
12:1
```

> We can mechanically rewrite our little probabilistic workflow language into a program that automatically computes the desired probability distribution.

It should now be pretty clear what the rewrite rules are for our simple DSL:

- Normal assignment and declaration statements invoke the "next statement" method with new values for the locals.
- Assignment or declaration statements with sample become `SelectMany`.
- The `condition` statement evaluates their condition and then either continue to the next statement if it is met or produces an empty distribution if it is not.
- `if` statements evaluate their condition and then invoke the consequence or the alternative helper method.
- `return` statements evaluate an expression and return a `Singleton` of it.

> **Exercise:** We severely restricted the kinds of statements and expressions that could appear in this language. Can you see how to implement:
>
> - `while`, `do`, `for`, `break`, `continue`, `switch` and `goto` labeled statements
> - compound assignment, increment, and decrement as statements

> **Exercise:** What about assignments, increments, and decrements as expressions; any issues there?

> **Exercise:** What about lambdas? Any issues with adding them in? Again, we're assuming that all functions called in this system are pure, and that includes lambdas.

> **Exercise:** We severely restricted where the sample operator may appear, just to make it easier on me. Can you see how we might ease that restriction? For example, it would be much nicer to write `TwoDSix` as:

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

How might you modify the lowering regimen we have described to handle that?

We'll discuss possible answers to all of these in future lessons.

Possible Issues with the Solution

As we have noted repeatedly in this course this is not how we'd actually implement the proposed feature. Problems abound:

- We have absurdly over-complicated the code generation to the point where every statement is its local function.
- We are passing around all the locals as parameters; probably some of them could be realized as actual locals.
- Each method is tail recursive, but C# does not guarantee that tail recursions are optimized away, so the stack depth could get quite large.
- Several of the methods take arguments that they do not ever use.
- We could have made some simple inlining optimizations that would decrease the number of helper methods considerably.

That last point bears further exploration. Imagine we applied an inlining optimization over and over again — that is, the optimization where we replace a function call with the body of the function. Since every function call here is expression-bodied, that's pretty easy to do!

What would come out the other end of repeated inlining is:

Show Hint

Is this then how we could implement this feature for real? Lower it to functions, then inline the functions? Though it would work, that's probably not how we'd do it for real.

Our purpose here was to show that it could be done with a straightforward, rules-based transformation; I'm not looking to find the optimal solution. It can be done, which is great!

> We stated a few lessons back that it would become clear why having an explicitly empty distribution is a win; hopefully it is now clear. Having an explicitly empty distribution allows us to implement `condition` using `SelectMany`! It is not at all obvious how to rewrite the workflow above into a query with a `Where` clause, but as we've seen, implementing the `condition` statement as conditionally producing an empty distribution gives us the ability to stick what is logically a Where anywhere in the workflow.

> Many lessons back we gave the challenge of implementing a `Where` that took a probabilistic predicate: `Func<T, IDiscreteDistribution>`. That is no challenge at all in our new workflow language: `bool b = sample predicate(whatever); condition b;`

You would agree that our proposed "imperative style" workflow is an improvement over our use of LINQ to project, `condition` and combine probability distributions; our `Workflow()` method is a lot easier to read as an imperative workflow than its query form, and not just because it uses the jargon of probability rather than that of sequences.

Implementation

Let's have a look at the code:

Program.cs

```
using System;
using System.Collections.Generic;
namespace Probability
{
    using System.Collections.Generic;
    using SDU = StandardDiscreteUniform;
    static class Episode21
    {
        public static void DoIt()
        {
            Console.WriteLine("Episode 21");
            AttemptOne.DoIt();
            AttemptTwo.DoIt();
        }
        static class AttemptOne
        {
            public static void DoIt()
            {
                Console.WriteLine("Attempt One: Be like iterator blocks");
                Console.WriteLine(Workflow(3).Histogram());
            }
            // Attempt #1 at lowering: be like "yield return" workflows.
            // * We generate a new class.
            // * Parameters become fields
            // * The body of the method becomes the Sample() method of the class.
            // * sample operators become Sample() calls
            // * failed conditions become restarts.
            // * Not sure yet how to implement Support and Weight

            static IDiscreteDistribution<bool> Flip() =>
```

```
    new FlipDistribution();

sealed class FlipDistribution : IDiscreteDistribution<bool>

{
    public bool Sample()
    {
        int x = Bernoulli.Distribution(1, 1).Sample();
        return x == 0;
    }

    public IEnumerable<bool> Support() =>
        throw new NotImplementedException();

    public int Weight(bool t) =>
        throw new NotImplementedException();
}

static IDiscreteDistribution<int> TwoDSix() =>
    new TwoDSixDistribution();

sealed class TwoDSixDistribution : IDiscreteDistribution<int>
{
    public int Sample()
    {
        var d = SDU.Distribution(1, 6);
        int x = d.Sample();
        int y = d.Sample();
        return x + y;
    }

    public IEnumerable<int> Support() =>
        throw new NotImplementedException();

    public int Weight(int t) =>
        throw new NotImplementedException();
}

static IDiscreteDistribution<string> Workflow(int z) =>
    new WorkflowDistribution(z);

sealed class WorkflowDistribution : IDiscreteDistribution<string>
{
    int z;

    public WorkflowDistribution(int z)
    {
        this.z = z;
    }

    public string Sample()
    {
    start:
        int ii = TwoDSix().Sample();
        if (ii == 2)
            return "two";
        if (!(ii != z))
            goto start;
        bool b = Flip().Sample();
        return b ? "heads" : ii.ToString();
    }

    public IEnumerable<string> Support() =>
```

```
            throw new NotImplementedException();

        public int Weight(string t) =>
            throw new NotImplementedException();
    }
}

static class AttemptTwo
{
    public static void DoIt()
    {
        Console.WriteLine("Attempt Two: Every statement is a local method");
        Console.WriteLine(Workflow(3).ShowWeights());
        Console.WriteLine("We can apply inlining optimizations to that mess");
        Console.WriteLine(WorkflowInlined(3).ShowWeights());
    }

    static IDiscreteDistribution<bool> Flip()
    {
        Func<int, IDiscreteDistribution<bool>> S1 = x =>
            Singleton<bool>.Distribution(x == 0);
        Func<int, IDiscreteDistribution<bool>> S0 = x =>
            Bernoulli.Distribution(1, 1).SelectMany(_x => S1(_x));
        return S0(0);
    }

    static IDiscreteDistribution<int> TwoDSix()
    {
        Func<IDiscreteDistribution<int>, int, int, IDiscreteDistribution<int>> S3 =
            Singleton<int>.Distribution(x + y);
        Func<IDiscreteDistribution<int>, int, int, IDiscreteDistribution<int>> S2 =
            d.SelectMany(_y => S3(d, x, _y));
        Func<IDiscreteDistribution<int>, int, int, IDiscreteDistribution<int>> S1 =
            d.SelectMany(_x => S2(d, _x, y));
        Func<IDiscreteDistribution<int>, int, int, IDiscreteDistribution<int>> S0 =
            S1(StandardDiscreteUniform.Distribution(1, 6), x, y);
        return S0(null, 0, 0);
    }

    static IDiscreteDistribution<string> Workflow(int z)
    {
        Func<int, bool, IDiscreteDistribution<string>> S5 = (ii, b) =>
            Singleton<string>.Distribution(b ? "heads" : ii.ToString());
        Func<int, bool, IDiscreteDistribution<string>> S4 = (ii, b) =>
            Flip().SelectMany(_b => S5(ii, _b));
        Func<int, bool, IDiscreteDistribution<string>> S3 = (ii, b) =>
            ii != z ? S4(ii, b) : Empty<string>.Distribution;
        Func<int, bool, IDiscreteDistribution<string>> S2 = (ii, b) =>
            Singleton<string>.Distribution("two");
        Func<int, bool, IDiscreteDistribution<string>> S1 = (ii, b) =>
            ii == 2 ? S2(ii, b) : S3(ii, b);
        Func<int, bool, IDiscreteDistribution<string>> S0 = (ii, b) =>
            TwoDSix().SelectMany(_ii => S1(_ii, b));
        return S0(0, false);
    }

    static IDiscreteDistribution<string> WorkflowInlined(int z) =>
        TwoDSix().SelectMany(ii =>
            ii == 2 ?
                Singleton<string>.Distribution("two") :
                ii != z ?
                    Flip().SelectMany(b =>
```

```
                                Singleton<string>.Distribution(b ?
                                    "heads" :
                                    ii.ToString())) :
                                Empty<string>.Distribution);
        }
    }
}
```




It looks like we've come up with a reasonable syntactic sugar, but is it just sugar? Is there anything that our "imperative" workflow can do that our LINQ workflow cannot? Let's discuss this in the next lesson.