Challenges for Improving Imperative Probabilistic Workflows

WE'LL COVER THE FOLLOWING

In the previous lesson, we gave you several challenges for improving our DSL for imperative probabilistic workflows.

Compiler Writer's Approach

Let's solve a few puzzles:

Question One: You are walking down the street when you see a can of gasoline, some matches, a house on fire, a wrench, a fire hose, and a fire hydrant. You consider the house being on fire to be problematic, so what do you do?

Show Hint

Question Two: You are walking down the street when you see a can of gasoline, some matches, a house *not on fire*, a wrench, a fire hose, and a fire hydrant. What do you do?

Show Hint

That's how compiler writers approach problems. We know how to do lowerings in a very simplified version of probabilistic C#; to lower more

lowerings in a very simplified version of probabilistic C#, to lower more

complicated forms, we keep on applying the "lower it to a previously solved problem" technique:

- `goto` is straightforward: you evaluate the "statement method" that is the statement you're going to.
- Once you have `goto` and `if`, it is straightforward to implement `do`, `while`, `break` and `continue`; reduce those to combinations of `if` and `goto`, which are previously solved problems.
- `for` loops are just an irritatingly complicated version of `while`, which is an already solved problem.
- Implementing `throw` and `try` is tricky because it is a non-local goto. Throughout the history of C#, methods involving various combinations of `try`, `catch`, `finally`, `throw`, `yield return`, `yield break` and `await` had restrictions placed on them in C# because of the complexities involved in handling non-local branching in complex workflows. The details of how a compiler team might go about dealing with the combination of `throw` and `sample` in a probabilistic workflow are beyond the scope of this course; if this subject interests you, consider doing research on how the C# compiler implements asynchronous workflows that can throw.
- We have not solved the problem for `try`, which means that we're blocked on `foreach`, `lock` and `using`, all of which are syntactic sugars for `try`.

To allow the `sample` operator in more places, again, reduce the problem to previously solved problems. For example:

```
x = Foo(sample A(), sample B()[sample C()]) + sample D();
```

is equivalent to

```
var a = sample A();
var b = sample B();
var c = sample C();
var f = Foo(a, b[c]);
var d = sample D();
x = f + d;
```

and we already know how to lower all of those.

- You've got to be careful with the operators that conditionally evaluate their operands (`&&`, `||`, `??`, `?:`), but you can lower those into `if` statements.
- Assignments, increment and decrements used in expression locations can be handled with the same trick: turn it into a form that has only statements that we already know how to do.
- For lambdas and anonymous methods, we've previously stated upfront that they need to be pure, which means that they should not be closed over values which change. Given that restriction, lambdas (and therefore also query expressions) can be allowed.
- Moreover, we know how to turn lambdas into ordinary methods; if we apply the same restrictions to a lambda body as we do to probabilistic methods then *we could allow probabilistic lambdas*; that is, lambdas that return probability distributions and contain `sample` operators and `condition` statements. This is much the same as the way C# allows async lambdas.

> If actually implementing this scheme sounds super complicated with lots of details to keep track of and performance pitfalls everywhere, congratulations, you now know what it is like to be a compiler developer!

Compiler developers have to do all of this stuff even without fancy kinds of workflows in the language; everything that we are describing is a problem that has to be solved during the codegen phase of a compiler because IL does not have concepts like `while` or `for` or "*assignment as an expression*". It's all about reducing programs to more straightforward and simpler forms until you get it down to the IL.

Let's assume for now that we can make all those basic enhancements to our imperative DSL, and address the question we left you with last time: *is this sugar for queries just a sugar*, or *is there something we can do with it that we can't do with a query*?

Yes and no.

As we've seen, ultimately *everything* gets desugared into calls to `SelectMany`, so no. Anything you can do with this notation, you can do with a `SelectMany`, and therefore you could write a query comprehension. However, this notation certainly makes it much easier to clearly and concisely represent operations that would look pretty odd in a query.

For example, we don't usually think of queries as being recursive, but recursive workflows are straightforward in this DSL, so maybe there is an argument that yes, there is some more expressiveness in the statement form.

How's that, you say? Let's look at an example of a recursive workflow.

Let's play a game in n rounds; we'll state upfront what n is. You flip a coin. If it is `heads`, I pay you n dollars, and we're done. If it is `tails`, we go to the next round. In the next round, you flip a coin. If it is `heads`, I pay you $n - 1$ dollars, if tails, we go to the next round, and so on. If we get down to zero dollars, the game is over, and you get nothing. (The recursion has to stop somewhere!)

> We do not have to implement this using recursion; we could use a loop. The point is that we can easily implement it with recursion, so recursion is a tool we have available in our toolbox when building probabilistic workflows.

> **Exercise:** what would a fair price be for you to pay me upfront so that the game is on average a net gain for neither you nor me? The answer is below.

We are going to assume that we've made all the straightforward improvements to our DSL, including allowing probabilistic expression-bodied methods. We can implement the logic of our game like this:

```
probabilistic IDiscreteDistribution<int> Game(int rounds) =>
  (rounds <= 0 || sample Flip()) ?
    rounds :
    sample Game(rounds - 1);
```

We can do an initial lowering to a simpler form:

```
probabilistic IDiscreteDistribution<int> Game(int rounds)
{
  S0: if (rounds <= 0) goto S5;
  S1: x = sample Flip();
  S2: if (x) goto S5;
  S3: y = sample Game(rounds - 1)
  S4: return y;
  S5: return rounds;
}
```

And then lower that form into actual runnable C# 7. (I'll elide the unnecessary locals and temporaries.)

```
static IDiscreteDistribution<int> Game(int rounds)
{
  Func<IDiscreteDistribution<int>> S5 = () =>
    Singleton<int>.Distribution(rounds);
  Func<int, IDiscreteDistribution<int>> S4 =  y =>
    Singleton<int>.Distribution(y);
  Func<IDiscreteDistribution<int>> S3 = () =>
    Game(rounds - 1).SelectMany(y => S4(y));
  Func<bool, IDiscreteDistribution<int>> S2 = x =>
    x ? S5() : S3();
  Func<IDiscreteDistribution<int>> S1 = () =>
    Flip().SelectMany(x => S2(x));
  Func<IDiscreteDistribution<int>> S0 = () =>
    rounds <= 0 ? S5() : S1();
  return S0();
}
```

> (Inlining that into a query is left as an exercise.)

We asked above what is the fair price to pay; that's the expected value. We can work that out directly from the distribution:

```
public static double ExpectedValue(
  this IDiscreteDistribution<int> d) =>
    d.Support().Select(s =>
      (double)s * d.Weight(s)).Sum() / d.TotalWeight();
```

And if we run it, we get the answers we seek:

```
Console.WriteLine(Game(5).ShowWeights());
Console.WriteLine(Game(5).ExpectedValue());
```

```
5:16
4:8
3:4
2:2
1:1
0:1
4.03125
```

You, who know probability theory, could have predicted the results: out of a large set of games, on average you get paid $54 half the time, 4 a quarter of the time, and so on.

If you paid me $4.00 for the chance to play this game in five rounds, on average, you'd come out ahead by a few cents. Half the time you'd net a dollar, a quarter of the time you'd break even, and the rest of the time you'd lose some or all of your money.

Again the thing that is so exciting about this technique is: we humans didn't need to work out either the weights or the expected value; we wrote a program describing the game and executing that program solved the exact distribution of the results!

What we are showing here is that we don't have to work out what the distribution of this silly game is. Nor do we have to simulate the game, collect a hundred thousand data points, and make a statistical approximation of the distribution. We describe the flow of the game in imperative code, run the method, and the thing that pops out the other end is the exact distribution! It's almost magical.

> 💡 Also, notice that we implemented a new basic statistical operation here: we can compute the expected value of a discrete distribution on numbers with just a few short lines of code. The question of how to solve the more general version of this problem – how to compute the expected value of a continuous distribution – will be the subject of chapter 4 of

this course.

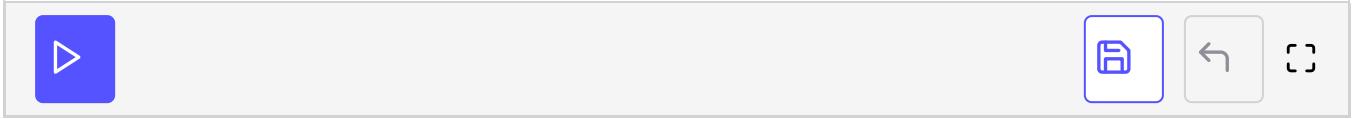
Implementation

Let's have a look at the code:

Program.cs
Bernoulli.cs
BetterRandom.cs
Distribution.cs
Empty.cs
Episode22.cs
Extensions.cs
IDistribution.cs
IDiscreteDistribution.cs
Projected.cs
Pseudorandom.cs
Singleton.cs
StandardCont.cs
StandardDiscrete.cs
WeightedInteger.cs

```
using System;
namespace Probability
{
    static class Episode22
    {
        public static void DoIt()
        {
            Console.WriteLine("Episode 22");
            Console.WriteLine(Game(5).ShowWeights());
            Console.WriteLine(Game(5).ExpectedValue()); // 4.03125
        }

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

        probabilistic IDiscreteDistribution<int> Game(int rounds) =>
```

```
            (rounds <= 0 || sample Flip()) ?
              rounds :
              sample Game(rounds - 1);

        // This lowers to:

        probabilistic IDiscreteDistribution<int> Game(int rounds)
        {
          S0: if (rounds <= 0) goto S5;
          S1: x = sample Flip();
          S2: if (x) goto S5;
          S3: y = sample Game(rounds - 1)
          S4: return y;
          S5: return rounds;
        }

        // Which lowers to:
#endif
        static IDiscreteDistribution<int> Game(int rounds)
        {
            Func<IDiscreteDistribution<int>> S5 = () =>
                Singleton<int>.Distribution(rounds);
            Func<int, IDiscreteDistribution<int>> S4 =  y =>
                Singleton<int>.Distribution(y);
            Func<IDiscreteDistribution<int>> S3 = () =>
                Game(rounds - 1).SelectMany(y => S4(y));
            Func<bool, IDiscreteDistribution<int>> S2 = x =>
                x ? S5() : S3();
            Func<IDiscreteDistribution<int>> S1 = () =>
                Flip().SelectMany(x => S2(x));
            Func<IDiscreteDistribution<int>> S0 = () =>
                rounds <= 0 ? S5() : S1();
            return S0();
        }
    }
}
```



In the next lesson, we will have a look at the problems with Discrete Probability Distributions.