Calculating with Infinite Sequences in C#

By Tomas Petricek

My previous article explored *lazy evaluation* and looked at how it can be simulated in C#. We implemented a class Lazy<T>, which represents a value that can be evaluated on demand—this means that the class "knows" how to calculate the result, but doesn't actually calculate it until it is really needed in the program. One very interesting data structure known from functional programming that can be implemented using this lazy cell is a *lazy list*. Each element in a lazy list stores the value associated with the element (for example a number) and knows how to calculate the next element in the list, but it is *lazy* meaning that the next element is not calculated until its value is accessed by the program. The most interesting aspect of lazy lists is that they can be used for representing infinite sequences—this is possible because the elements are calculated only when needed and so the list will always store only a finite number of elements, but it will still be able to calculate the next element if it is needed.

In this article I will show that lazy lists can be implemented using the IEnumerable<T> type and I will also demonstrate how LINQ query operators can be used to manipulate infinite lists. Finally, we will look at an implementation of the Mandelbrot set program (you can see it on the screenshot) using the techniques described in this article.

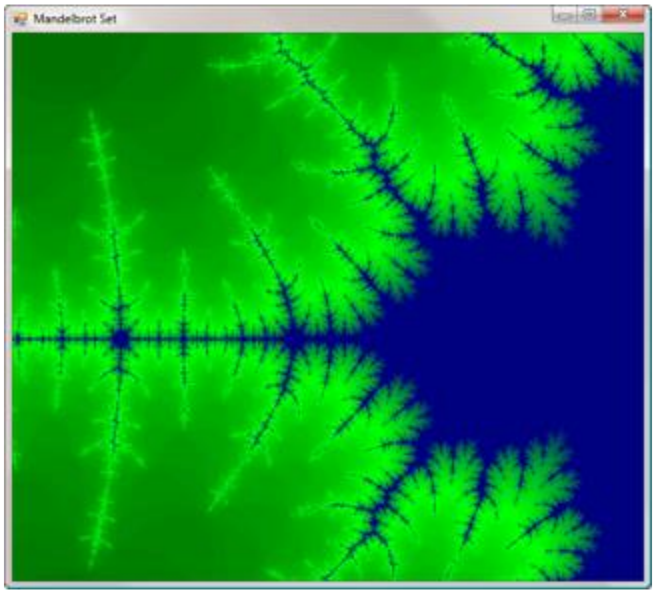

Figure 1. Mandelbrot Set (Click for larger image.)

Introducing IEnumerable

Most of you probably know that .NET sequences can be represented using the IEnumerable<T> interface. It can be used for enumerating items in standard collections such as arrays, lists, dictionaries, and it can also be used in custom collections. This interface is also very often used together with the foreach statement. Let's first review the methods of this interface. (Other methods such as those inherited from IDisposable are not important for our examples):

```
interface IEnumerable<T> : IDisposable<T>
{
  public void Reset();
  public bool MoveNext();
  public T Current { get; }
  // ... other methods omitted ...
}
```

An enumeration over a data structure is done using the three members shown in the previous code example. First the Reset method is called to move the enumerator before the first element. After that the MoveNext method moves the enumerator to the next element in the sequence or returns false if there are no more elements. During the enumeration, the selected element can be accessed using the Current property.

Until version C# 2.0 you had to provide an implementation of this interface when you wanted to allow enumeration over your own collection. This introduced unnecessary complexity, so in C# 2.0 yield return was introduced. Using these keywords, implementing a custom enumerator is just as easy as iterating over all the elements in a loop. Let's look at a very simple example:

```
class MyList {
  int[] elements;
  IEnumerabl<int> GetEnumerable() {
    for(int i=0; i<elements.Length; i++)
      yield return elements[i];
  }
}
```

This class shows how a method that returns an implementation of IEnumerable<T> can be used for enumerating all the elements in an array. The body of this method doesn't return an explicit implementation of the interface and instead uses yield return, which is compiled into an interface implementation by the C# compiler. Understanding what the code does is easy (it just iterates over all the elements and returns every single element from the array using yield return), but understanding how the code is used to implement the IEnumerable<T> interface would require another article. The important thing to understand is what happens when you call the method implemented using yield return as in the following example:

```
var en = myList.GetEnumerable();
```

When you perform this call, the object generated by the compiler is created, but not a single line of the code that you wrote is executed! The code starts executing after we call the MoveNext method. It steps to the first occurrence of yield return and then "leaves" the method and the control is transferred back to the caller of MoveNext. Of course it is not possible to step outside from an executing method and then "jump" back (keeping the original state of variables in the method), so the compiler has to translate the code in the method to a class, lift the variables to convert them to class members and generate code that implements a transition between each use of yield return. The C# compiler translates the code of the method to a state machine where every use of yield return represents a single state.

Infinite Sequences and LINQ Query Operators

The most important thing that I wanted to point out in the previous section is that the entire body of the method is not executed unless the caller repeatedly calls MoveNext until it reaches the end and the method returns false. This means that if we write code that would execute forever, and include a call to yield return inside the code, it will not cause an infinite loop. Let's look at the simplest possible example:

```
IEnumerable<int> GetNumbers()
{
  int i = 0;
  while(true) yield return i++;
}

// Create an infinite sequence of integers
var nums = GetNumbers();
```

The GetNumbers method in the previous code returns an implementation of IEnumerable<int> that can be used for looping over all integers starting from 0. Since the int type has a limited range the value will eventually overflow and start counting up again from Int32.MinValue.

In the example shown above we call the method and create a variable called nums. This call simply returns a number when we call MoveNext—in and of itself it doesn't cause any infinite loops. However, you still have to be careful when working with this code. If you passed it as an argument to foreach it would cause an infinite loop because foreach would continue calling MoveNext until it returned false, which will never happen in the case of the sequence generated by the GetNumbers method!

Now, let's look at what will happen when we use LINQ query operators with the infinite sequence that we just declared. In the following example we use the Where extension method to filter only even numbers and the Select extension method to calculate the square of the filtered (even) numbers:

```
var sq = GetNumbers().Where(n => n % 2 == 0).Select(n => n * n);
```

We can use C# 3.0 query comprehension syntax to write semantically equivalent code:

```
var sq = from n in GetNumbers() where n % 2 == 0 select n * n;
```

Will this cause an infinite loop? No, because we're still working with lazy sequences represented using IEnumerable<int>. The call to the Where query operator returns a new sequence with a MoveNext method that, when called, iterates over the underlying sequence until it finds an even number and then stops. Similarly, a single step in the sequence returned by the Select operator just takes one element and calculates its square. The only remaining problem is that we have to be careful when printing the sequence, because we can't simply use foreach. Luckily, LINQ provides a Take query operator, which returns a sequence that will take the first *X* elements and then stop, so we can write the following code:

```
foreach(int n in sq.Take(10))
  Console.WriteLine(n);
```

Clearly, Where and Select are not all query operators that you can use with infinite sequences, but you have to be careful: for example, Aggregate, Reverse or any join operators need to iterate over all the elements, which isn't possible when working with infinite sequences. Finally, you can implement your own iterators to perform operations that are not covered by standard LINQ query operators—for more information, you can look at the C# Developer Center article Custom Iterators from Bill Wagner.

Generating Infinite Sequences

So far we generated sequences explicitly using yield return. Sometimes it may be useful to write a reusable function that hides the call to yield return and provides a higher level of abstraction. For example, many sequences can be described in an inductive way, by giving the first element and a function that takes a value as an argument and calculates the next value in a sequence. This function can be implemented using the C# 3.0 Func<A0,T> delegate type:

```
IEnumerable<T> Generate<T>(T initial, Func<T, T?> next)
    where T : struct {
  T? val = initial;
  while (val.HasValue) {
    yield return val.Value;
    val = next(val.Value);
  }
}
```

The Generate function is generic, so it can generate sequences of any value type, so we can use C# nullable types. The first argument of the function is the initial value of type T. The second argument is a function that accepts a value of T as an argument and returns a T?, which means that it can return null or a valid value. The null return value is used to denote the end of a sequence and as you can see in the body, the condition in the while loop tests if the returned nullable type contains a value. This means that the Generate function can be used for generating both infinite and finite sequences.

The following code shows a few sequences that can be written in a very compact way using the Generate method:

```
// All positive integers: 1, 2, 3, 4, ...
var nums = Generate(0, n => n + 1);

// Powers of two: 1, 2, 4, 8, 16, 32, ...
var pow2 = Generate(1, n =>; n * 2);

// Finite sequence of powers of two that fit into an 'int' type
// When we get to a number that would overflow, we return 'null'
var intpow2 = Generate(1, n => {
    if (n > Int32.MaxValue / 2)
      return (int?)null;
    else
      return (int?)(n * 2);
  });
```

Example: Mandelbrot Set Using Infinite Sequences

In the final example we will use infinite sequence to draw the famous Mandelbrot set. When drawing the image shown on the screenshot above, we will need to calculate a color for every pixel separately. This operation can be implemented using sequences. This technique makes the code easy to read, because the implementation doesn't hide how the Mandelbrot set is defined mathematically. We'll get to the calculation of the color soon, but first, we need to build a palette with colors that you can see on the screenshot. To do this we first define two more utility functions for working with IEnumerable<T>:

```
IEnumerable<int> Range(int from, int to) {
  for (int i = from; i <= to; i++) yield return i;
}
IEnumerable<T> Concat<T>(params IEnumerable<T>[] args) {
  foreach (var en in args)
    foreach (var el in en) yield return el;
}
```

The method called Range simply returns all integers in a specified range. The second function is used for a concatenation of sequences. It takes any number of sequences and iterates step by step over all the given sequences. Now let's look how the palette is generated:

```
Color[] Palette {

  get {
    return Seq.Concat(
        Seq.Range(0, 127).Select(n =>
          Color.FromArgb(0, n * 2, 0)),
        Seq.Range(0, 127).Select(n =>
          Color.FromArgb(0, 255 - n * 2, n)) ).ToArray();
  }
}
```

As you saw on the screenshot, the palette starts with the color black, then blends to blue and finally changes to a green color. This means that we can generate it by concatenating two transitions (from black to blue and from blue to green) and this is exactly what the previous code does. It generates two ranges of integers and uses these numbers to generate colors using the Select query operator. The two ranges are then concatenated and the result is converted to an array and returned.

Now we can get back to the Mandelbrot set. As already mentioned, we will first write code that generates (in theory) infinite sequence of numbers and then use it to choose the color. The calculation of the sequence takes two arguments that specify the *x* and *y* coordinates of the point that we're processing. The sequence is calculated inductively, which means that we start with some initial value and then use the current value to calculate a new one. We already discussed this kind of sequence and showed how to use the Generate method to implement it easily.

The value in our example is represented using the PointF type. We use a zero point as the initial value. In every step we will first test if x^2 + y^2 is larger or equal to two. If yes, we have reached the end of the sequence (because the value is out of the specified range), otherwise we have to calculate the new value (using an equation for multiplication and addition of complex numbers), because we still can't tell if the value will eventually escape the range or not.

```
private static IEnumerable<PointF> GenerateMandel(float x, float y)
{
  var init = new PointF(0.0f, 0.0f);
  return Generate(init, (t) => {
      float x2 = t.X * t.X, y2 = t.Y * t.Y;
      if (Math.Sqrt(x2 + y2) >= 2)
        return null;
      else
        return new PointF(x2 - y2 + t.X, 2 * t.X * t.Y + t.Y);
    });
}
```

As mentioned earlier, we used the Generate method to build the sequence. You can see that using anonymous functions from C# 3.0 makes this code really compact—because the computation of the next value is a bit longer we use anonymous functions with a statement block as a body—this means that the code is wrapped in curly braces and uses a return statement.

Now we know how to calculate a sequence for every point and the only remaining problem is how to draw the bitmap using the sequence. We want to determine if the sequence will at some time in the future leave the specified range. If this happens then we know how many iterations it took and we can choose a color according to the iteration; however, when the sequence still stays in the specified range we don't know if it will ever leave it or not, so we could iterate forever! To avoid this we just stop calculating after 100 iterations and choose the lightest (green) color.

To count the number of iterations we would normally use the Count query operator, but we also need to limit the number of iterations to some maximum number. To do this we can write the following helper method that uses TakeWhile to read values from the sequence until the counter reaches zero and then uses Count to get the number of elements in the sequence (also note that we don't need the argument of the anonymous function, so we just use underscore as a variable name):

```
int CountMax<T>(this IEnumerable<T> en, int max) {
  return en.TakeWhile(_ => max-- > 0).Count();
}
```

Using this CountMax method we can now implement the code that draws the Mandelbrot set. We first generate the palette and initialize variables that determine which part of the set is drawn. After that we also use a custom method called InitBitmap (described below) that generates a bitmap of specified size by calling a given function to get a color for every single pixel. In the anonymous function that returns the color we first calculate a point that we want to analyze and then call GenerateMandel to generate a sequence for this specific point. Then we use CountMax to calculate how much iterations did the sequence perform and finally we get a color for the performed number of iterations.

```
var colors = Palette;

// Initialize a location and a zoom of the set
float dx = 0.005f, dy = 0.005f;
float xmin = -2.0f, ymin = -1.3f;

// Generate bitmap using given function..
var mandel = InitBitmap(this.Width, this.Height, (x, y) => {
  float xs = xmin + x * dx, ys = ymin + y * dy;

  // Calculate number of iterations
  int count = GenerateMandel(xs, ys).CountMax(100);

  // Get a color for the calculated # of iterations
  int val = (int)((count / 100.0) * 255.0);
  return colors[val];
 });
```

As already mentioned, the code uses InitBitmap method, which takes the function to calculate the color for a pixel as the third argument. We use C# 3.0 anonymous functions to implement this computation. In the body we first pre-compute the point in the Mandelbrot set coordinates. Using this point we generate the sequence of iterations and count the number (with 100 as a maximum). Finally, we just choose a color from the palette and return it as a result of the anonymous function.

The InitBitmap method is quite simple—it just iterates over all the pixels in the bitmap and calls the given function to calculate the color. (Indeed, it is more efficient to use the C# unsafe code to do this—this implementation can be found in the code that is attached for download.)

```
Bitmap InitBitmap(int width, int height, Func<int, int, Color> f) {
 Bitmap b = new Bitmap(width, height);
 for (int x = 0; x < this.Width; x++)
  for (int y = 0; y < this.Height; y++)
   b.SetPixel(x, y, f(x,y));
 return b;
}
```

Conclusion

In this article we looked at one not widely know aspect of the IEnumerable<T> type, which is the fact that it can be used to represent infinite sequences of values. We also demonstrated that it is possible to use some of the LINQ query operators including the query comprehension syntax for working with these infinite data structures. I also demonstrated that it is possible to use C# 3.0 anonymous functions when manipulating or generating these types of sequences. Finally, we looked at one more complicated example that used the described technique and several new C# 3.0 language features to build an easier to understand implementation of a program for drawing the famous Mandelbrot set in a way that is closer to the original mathematical definition of the problem.

Download the souce here.