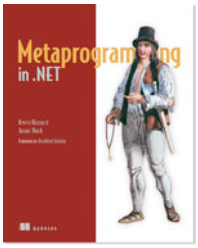

Using Expressions Effectively

By Kevin Hazzard and Jason Bock, author of *Metaprogramming in .NET*

Writing code in IL can easily lead to incorrect implementations and requires a mental model of code execution in .NET that's not as intuitive as the one a high-level language provides. Fortunately, there's another API in .NET that lets you create code without having to know anything about IL. This is the Expression API that exists within the LINQ world. In this article, based on chapter 6 of Metaprogramming in .NET*, the authors show you how to use expressions effectively.*

Now that you've seen how expressions are created, let's get into areas that will make your expression life easier with debugging, emitting, and mutating. Learning these techniques is important so that you know your expressions do what you think they should do. Let's begin by looking at how to add debug information to your expression.

Debugging expressions

Using the Expression API is a simpler, cleaner experience than trying to work with IL. That said, any time a developer writes a piece of code, something can go wrong. Fortunately, there are a couple of techniques you can use to debug your expressions. Let's start with the first one: visualizing your expression in the debugger.

Visualizing an expression in Visual Studio

Whenever you create an expression of any type, you can get a visualization of that node in Visual Studio when you run your code under the debugger. You move your mouse pointer over the variable in code, drill down to the Debug View option, and select Text Visualizer. Figure 1 shows what an expression looks like in this visualizer.

You may wonder why the language used in the visualizer doesn't use C# or VB. Expressions can support options that a language may not be able to express (like a fault block), so the designers of the visualizer came up with a different language to show the expression. It's not that hard to follow, and it's a quick and easy way to get a good idea for what your expression looks like at a particular point in its construction.

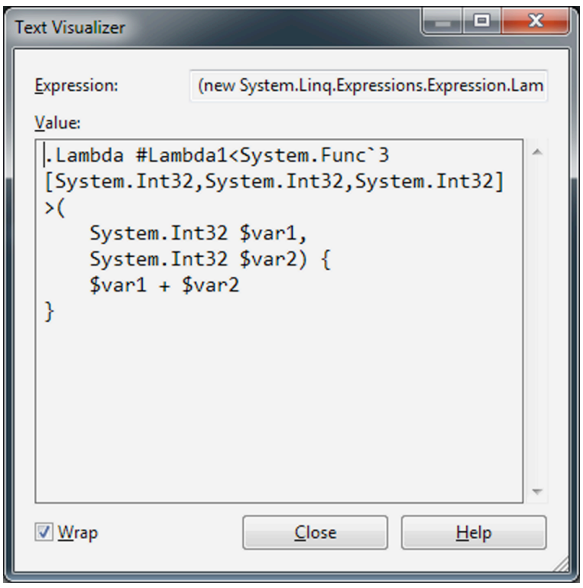

Figure 1 Visualizing an expression in Visual Studio. The language may not look like anything you've seen before, but the intent is clear.

NOTE For more information on the visualizer, see http://mng.bz/E6Q7.

In the next section, you'll see how you can step into an expression at runtime.

Using Reflection.Emit to debug expressions

Although a visual representation of an expression is a nice tool to have, sometimes you want to have the debugger dive right into the code. Unfortunately, an expression is a tree that represents your code structure. Or is it? When you compile your method, it's emitting IL for you. Surprisingly, there's a connection between expressions and Reflection.Emit that lets you create debug information for an expression. Let's see how you can get it to work.

Similar to the exception handler example, you're going to start with a simple "add two numbers together" code snippet. The first thing you need to do is create a bunch of dynamic members from the Reflection.Emit API. Don't worry, you won't need to write any IL for debugging purposes; these members act as a host to your expression, as you'll see in a moment:

```
var name = Guid.NewGuid().ToString("N");
var assembly = AppDomain.CurrentDomain.DefineDynamicAssembly(
  new AssemblyName(name), AssemblyBuilderAccess.Run); var module =
assembly.DefineDynamicModule(name, true);
var type = module.DefineType(
  Guid.NewGuid().ToString("N"), TypeAttributes.Public);
var methodName = Guid.NewGuid().ToString("N");
var method = type.DefineMethod(methodName,
  MethodAttributes.Public | MethodAttributes.Static,
  typeof(int), new Type[] { typeof(int), typeof(int) });
```

All this code is doing is getting an in-memory dynamic assembly set up along with dynamic type and method. As you can see, the names of these members don't matter in this case—what does matter is the debugging support. For that, you need a `DebugInfoGenerator`:

```
var generator = DebugInfoGenerator.CreatePdbGenerator();
var document = Expression.SymbolDocument("AddDebug.txt");
```

The generator created from `CreatePdbGenerator()` will be used when the expression is compiled. You also need to create a symbol document based on a text file. The AddDebug.txt file in this example is the expression

expressed in the language used in the expression visualizer demonstrated in the previous section. We won't show the code here, but you'll see what some of it looks like in a figure near the end of this section.

To create debug symbols for sections of the code, you need to wrap a specific node in the expression tree with a `DebugInfoExpression`:

```
var addDebugInfo = Expression.DebugInfo(document,
  6, 9, 6, 22);
var add = Expression.Add(x, y);
var addBlock = Expression.Block(addDebugInfo, add);
```

The section in the document that maps to the expression node being wrapped is defined in the `DebugInfo()` call. This `DebugInfoExpression` object is used in a `Block()` call to wrap the `BinaryExpression` that represents the addition functionality of this expression.

Once you're done defining your expression, save the expression's implementation into the dynamic assembly:

```
var lambda = Expression.Lambda(addBlock, x, y);
lambda.CompileToMethod(method, generator);

var bakedType = type.CreateType();
return (int)bakedType.GetMethod(methodName)
  .Invoke(null, new object[] { a, b });
```

In this case, you use `CompileToMethod()` to specify which method you're implementing in the dynamic assembly along with the debug information related to this method. At this point, when you step into the `Invoke()` call in a debugger, you'll get into the file specified in the `SymbolDocumentInfo` object. Figure 2 shows what this looks like when you step into the "expression" language copied to the text file.

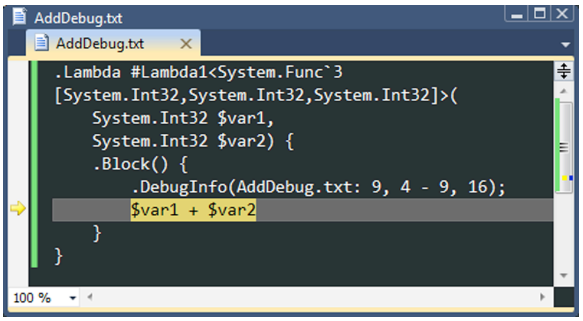

Figure 2 Debugging an expression. Even though the language may look a little odd, it's clear that you've stopped at the point where addition occurs in the method.

> **CAVEAT** Using expressions to implement your methods in Reflection.Emit is a nice alternative to IL. But there's one major limitation to this approach: you can only use expressions to define static methods. You can't use an expression for an instance method. For more information on why this is the case, see http://mng.bz/U254.

You now know how you can debug your expression.

Let's move on to another topic: immutability. It will definitely affect your design, and that's what the next section is all about.

Mutating expression trees

We'll now look at expressions, immutability, and how you can create new expressions based on existing ones via the `ExpressionVisitor` class. Let's start by looking at the reasoning behind having immutable trees.

Immutability of expression trees

To frame the conversation on immutable expressions, let's look at an expression that adds two integers together:

```
Expression<Func<int, int, int>> add = (x, y) => x + y;
```

Let's say someone wanted to change that expression to make it subtract the two arguments rather than add them. Because `Expression<T>` is a reference type, the expression you were referencing will now perform a

subtraction, not an addition. That's not at all what you want! Immutable data structures are easier to reason about because you know that once the structure is created, it won't change. This also has benefits from a concurrency perspective because these structures are automatically thread-safe.

> NOTE For more information on the benefits (and some shortcomings) of programming using immutable values see http://en.wikipedia.org/wiki/ Immutability and http://mng.bz/AId8.

Being able to use a structure as the basis for future changes isn't a bad thing. In the next section, you'll see how you can create a new expression based on an existing one.

Creating variations of expressions

You now know that expressions are immutable. Let's see how to create a new expression from the contents of an existing expression.

The key class you need is `ExpressionVisitor`. As the name implies, this class is based on the visitor pattern, which is designed to allow you to traverse complex object structures in a simplistic way by "visiting" specific methods that you care about.

> NOTE For more on the visitor pattern, see http://en.wikipedia.org/wiki/Visitor_pattern.

You feed a subclass of `ExpressionVisitor` the expression that's your baseline and then you overwrite the `VisitXYZ()` methods you're interested in to create a new expression. Let's create a custom visitor that will change an add operation to a subtract operation. The following code listing demonstrates what it takes to write such a visitor.

Listing 1 Creating an expression visitor

```
internal sealed class AddToSubtractExpressionVisitor
  : ExpressionVisitor
{
  internal Expression Change(Expression expression)
  {
    return this.Visit(expression);
  }

  protected override Expression VisitBinary(BinaryExpression node)
  {
    return node.NodeType == ExpressionType.Add ?
      Expression.Subtract(
        this.Visit(node.Left), this.Visit(node.Right)) :
      node;
  }
}
```

As you can see, it doesn't take that much code to change an expression. In this case, you override `VisitBinary()` because you're trying to find the mathematical expression node `Add`. If you find one, then you create a new node that subtracts the child nodes. Note that you need to keep visiting the `Left` and `Right` nodes from the given node because they may also contain addition operations. For example, if you didn't do that, an expression like this

```
(x, y) => ((((32 * x) / 4) + y) + (x + 4))
```

would turn into this:

```
(x, y) => ((((32 * x) / 4) + y) (x + 4))
```

That's not what you want, because there are still two addition operations in the expression. Visiting the child nodes gives you the right result:

```
(x, y) => ((((32 * x) / 4) y) (x 4))
```

> **NOTE** Before .NET 4.0, there wasn't a way in the .NET Framework to visit an expression. The `ExpressionVisitor` class existed in `System.Linq.Expressions`, but it was marked as internal, so you couldn't use it. There are a couple of ways to support this technique in .NET 3.5—see http://mng.bz/09bP and http://mng.bz/a3N3 for details on these approaches.

You now know how to debug and change expressions in .NET.

Here are some other Manning titles you might be interested in:

Windows 8 Apps with HTML5 and JavaScript
Dan Shultz, Joel Cochran, and James Bender

Fast ASP.NET Websites
Dean Alan Hume

Windows Store App Development: C# and XAML
Pete Brown

Last updated: 8/2/13