Type Inference

By **Bill Wagner**

May 2012

Introduction

This column is about one line of code that doesn't compile:

```
var lambda = x => x.M();
```

That line generates CS0815: "Cannot assign lambda expression to implicitly typed local variable."

In this column, I'll discuss why the C# language doesn't allow this, and explain the rules that govern type inference for implicitly typed local variables and lambda expressions.

What doesn't work

Think about what the compiler must do logically, and it's reasonable why assigning a lambda expression to an implicitly typed local variable isn't allowed. Implicitly typed local variables (those declared using the var keyword) infer their type from the initialization expression. Anonymous functions (which may be written as lambda expressions or anonymous method expressions) do not have a type, but are converted to compatible delegate or expression tree types when assigned. That means the above line of C# code tells the compiler to infer the type of the left-hand-side expression from the type of the right-hand-side expression, and infer the right-hand-side expression from the type of the left-hand-side expression. The C# compiler is good at understanding the intent of code, but that kind of circular logic requires human thinking.

I'll elaborate on that short answer so that you have a greater understanding of the features involved.

Type Inference Rules for var

Most developers are familiar with implicitly typed local variables, so I'll discuss it very briefly. Var simply means "this variable's type is the static type of the right hand side of the initialization expression." There is one important concept in that description whose importance developers sometimes miss. Implicitly typed local variables are based on the static type of the initializer, not the runtime type of the initializer. Consider this small code fragment:

```
public class B { }
public class D : B { }
static B FactoryForD()
{
   return new D();
}

var v = FactoryForD();
```

The static type of ‘v’ is B, not D. The compiler uses the static return type of FactoryForD() and that type is substituted for ‘var’ in the declaration.

Section 8.5.1 of the C# Language spec describes the restrictions on variable declarations that can be implicitly typed. The one concerning lambda expressions is that the initializer expression must have a compile time type. As a simple example, you cannot declare an implicitly typed variable using the null keyword (e.g. var x = null;) because null does not have a type (S 2.4.4.6). You can use var with the default expression, because that does have a type (e.g. var x = default(string);).

The restrictions on using null provide a simpler way of seeing the reason why lambda expressions cannot be used for the initializer. Furthermore, the fix is the same: you must coerce the initializer to have a known static type. The default expression is strongly typed, so does convey type information that the compiler can use when determining the type of the variable being declared. Lambda expressions require more changes than the simple change from null to default(type).

Anonymous Functions and Inference

Lambda expressions are one syntactic form for anonymous functions. Lambda expressions are generally a more concise format for anonymous method expressions, which remain for compatibility. The language rules for anonymous functions satisfy two very important goals. First, anonymous functions can be used for any compatible delegate, or expression tree type. Second, the supported syntax should be as lightweight as possible.

The second goal manifests itself in the concise syntax for lambda expressions. Developers use lambda expressions so often because the syntax is concise and easy to read. The addition of lambda expressions was motivated in part to provide that more concise syntax, which is especially useful for LINQ queries. You can define equivalent anonymous functions using either syntax:

```
Func<int, int> lambda = x => x + 1;
Func<int, int> anonMethod = delegate(int x) { return x + 1; };
```

However, there is a subtle difference between the two expressions on the right hand side of those declarations. Notice that I mentioned earlier that anonymous methods can be converted to either delegates (as shown above) or expressions. A lambda expression can be converted to an Expression:

```
Expression<Func<int, int>> lambda = x => x + 1;
```

However, the anonymous method expression cannot be converted to an expression tree:

```
// does not compile:
Expression<Func<int, int>> anonMethod = delegate(int x) { return x + 1; };
```

That highlights one difference between anonymous method expressions and lambda expressions. You can run into some rather surprising behavior because of these rules. Consider this code:

```
public class B { }
public class D : B { }


public static void M(Func<B> f) { }
public static void M(Expression<Func<D>> f) { }


M(() => new D());
M(() => new B());
M(delegate { return new D(); });
M(() => { return new D(); });
```

The first call to M resolves to second overload of M, the version that takes an Expression<Func<D>>. That's because the lambda expression is a better match for the formal parameter (an expression that returns a D). The second call to M resolves to the first overload of M. That's because it takes an expression that returns a B. The third call to M resolves to the first overload of M. That's because the 'delegate' keyword declares that the anonymous method expression represents a delegate, and not an expression tree. The fourth call is where there is an issue: It generates a compile error. That's because the lambda expression resolves to the second overload. However, because the lambda expression has statement for a body (instead of an expression, as in the first line), the conversion fails at compile time. The better overload, based on resolution rules, is invalid at compile time. You can avoid this issue by preferring expression bodies for lambda expressions when possible instead of statement bodies. They are more versatile.

That little dive into the subtleties of lambda expression conversion illustrates the first goal, that lambda expressions can be used for any applicable delegate or expression. This goal is represented by the language rule that states anonymous functions do not have a type but can be implicitly converted to a compatible delegate type or expression tree type.

As you saw above, the lambda expression "x => x + 1" is convertible to Func<int, int> or Expression<Func<int, int>>. What may not be obvious is that those are two of a large number of possible delegate or expression types. Consider this declaration:

```
public delegate int IntFunc(int a);
```

That expression "x => x + 1" could be converted to an IntFunc, or an Expression<IntFunc>. In practice, that expression could be converted to any delegate type that took an int parameter and returned an int. Furthermore, the language rules do not permit conversions between different delegate types. (It's not explicitly called out that conversions between delegate types are illegal. However, the absence of any

rule that there are conversions between delegate types means they are illegal.) That means an IntFunc cannot be assigned to a variable of type Func<int, int>, and vice versa.

Of course, we've only scratched the surface of possible delegate and expression types that "x => x + 1" could be converted into. It could also be converted to any delegate that took any numeric type and returned that numeric type (short, byte, long, float, decimal, double). But, that's not all. The expression could be converted to any delegate whose parameter has a valid operator + . For example:

```
public class Counter
{
   private int counter = 0;

   public static Counter operator +(Counter l, int r)
   {
      var rVal = new Counter();
      rVal.counter = l.counter + r;
      return rVal;
   }
}
```

The expression "x => x + 1" is now convertible to Func<Counter, Counter>, and Expression<Func<Counter, Counter>>.

Now you see why the compiler made the clear rule that you cannot assign a lambda expression to an implicitly typed local variable. You may think it's obvious what you mean, and what type the variable should take. The language rules were written with other goals in mind, namely that lambda expressions can be converted to the largest possible set of compatible delegate or expression types. In order to have greater flexibility using anonymous functions, you cannot assign any anonymous function to an implicitly typed local variable.

This article describes the rules for the C# language. Other languages have other goals, and therefore have other rules. Visual Basic allows this construct:

```
Dim lambda = Function(x) x + 1
```

The variable lambda is a compiler generated delegate type that takes an int and returns an int. This is keeping with VB's goals to optimize programmer productivity even when that means less control over exactly how an expression may be resolved. Furthermore, VB does allow implicit conversions between different anonymous functions that have the same parameters and return types. The following VB.NET code is legal, whereas it is not in C#.

```
Sub Main()
   Dim lamdba = Function(x) x + 1
   Test(lambda)
End Sub
```

```
Sub Test(ByVal f As Func(Of Int16, Int16))
End Sub
```

F# is somewhere between C# and VB.NET with its rules. F# allows more implicit conversions and requires fewer explicit type declarations than C#. However, its rules are a little tighter than VB.NET.

Different languages, even general purpose programming languages, start with different goals. A well designed language will keep those goals in mind as it defines the language rules. C#'s rules for evaluating anonymous functions are designed around programmer productivity, while still providing control over how your code translates into executable instructions. The team chose maximum flexibility in how an anonymous function can translate into a delegate or an expression tree. The C# language brings potential ambiguities to your attention. The VB.NET language chose to make the best guess at your intent, getting out of your way, but possibly missing some ambiguity.