Calling functions in LINQ queries

Tomas Petricek tomas@tomasp.net The LINQ project (see [1]) is an extension to .NET Framework and most common .NET languages (C# and VB.Net) that extends these languages with query operators and some other features that make it possible to integrate queries in the languages. This article requires some previous knowledge of LINQ and C# 3, so I recommend looking at the the specification at LINQ home page.

1 What is and what isn't possible

There is a limitation of what you can write in a DLINQ query. This is clear, because DLINQ queries are translated at runtime to T-SQL statements, so the possibilities of DLINQ queries are limited by the T-SQL language. Conversion to T-SQL supports many of standard language constructs, but it can hardly support calling of your methods for two reasons. First (more important) reason is that the method is compiled and it is not possible to get its expression tree (as long as you don't use utility like Reflector). The second reason is that T-SQL is very limited when compared with C# as I mentioned earlier, although saying that T-SQL is limited is not fair, because it has different purpose than languages like C# and it does it very well. Let's take a look at the following example. It is a simple DLINQ query against the Northwind database that calls function MyTest in the where clause:

```
// function used in filter
static bool MyFunc(Nwind.Product p)
{
   return p.ProductName.StartsWith("B");
}
// query that uses MyFunc
var q =
   from p in db.Products
   where MyPriceFunc(p.UnitPrice) > 30m
   select p
```

It compiles with no errors, but when you execute it DLINQ throws an exception saying: "Static method System.Boolean MyTest(LINQTest.Nwind.Product) has no supported translation to SQL." The exception is actually thrown when you try to fetch results from q (for example using the foreach statement), because DLINQ attempts to convert the expression trees to T-SQL only when the results are needed and the query must be executed. To fix the example you can simply copy the code that checks whether product name starts with "B" to the where clause of the query and it would work fine.

I'd say that checking whether query can be translated to T-SQL at runtime is slightly against the DLINQ objective to catch all errors at compile time, but this is not the aim of the article. Also the rules for writing correct queries are not difficult. You can use all operators and some basic methods (like String.StartsWith) for working with numbers and strings, but you can't call any other methods, particlularly methods that you wrote.

2 The problem

The problem I wanted to solve is that you can't call your methods from queries. If you have a more complex application with many similar queries it would be natural to put the common subqueries to some function that is called by other queries. For example you might want to write function that does some price calculations or function that selects information about one significant product in category. If you look at some questions in MSDN Forums you can see that I'm not the only one who is asking for this (see [2]). First I'll show a little example that demonstrates what you can do with the library I wrote to keep your interest and than I'll explain the solution details. The following query selects products from the Northwind database and performs calculation of price for every product. Because we want to select only products that cost more than 30, the price calculation is repeated twice in the query:

```
// Notice that code for calculating price is repeated
var q =
  from p in db.Products
  where (p.UnitPrice * 1.19m) > 30.0m
  select new
       {
            p.ProductName,
            OriginalPrice = p.UnitPrice,
            ShopPrice = (p.UnitPrice * 1.19m)
        };
```

Using the extensions that I'll describe later you can extract price calculation to lambda expression and use this expression in the query. Following code can be used for querying both database and in-memory objects, because it is possible to translate the lambda expression to T-SQL as well as execute it at runtime:

```
// Lambda expression that calculates the price
Expression<Func<Nwind.Product, decimal?>> calcPrice =
  (p) => p.UnitPrice * 1.19m;
// Query that selects products
var q =
  from p in db.Products.ToExpandable()
  where calcPrice.Invoke(p) > 30.0m
  select new
        {
        p.ProductName,
        OriginalPrice = p.UnitPrice,
        ShopPrice = calcPrice.Invoke(p)
    };
```

Declaration of lambda expression that will be used by the query is straightforward, because it isn't different than any other C# 3 lambda expressions. The query is more interesting because it uses two extension different methods. First method called **ToExpandable** creates a thin wrapper around the DLINQ **Table** object. Thanks to this wrapper you can use second method called **Invoke** that extends the **Expression** class to invoke the lambda expression while making it still possible to translate query to T-SQL. This works because when converting to the expression tree, the wrapper replaces all occurrences of **Invoke** method by expression trees of the invoked lambda expression and passes these expressions to DLINQ that is able to translate the expanded query to T-SQL.

3 Implementation details

3.1 Expression expansion

As I mentioned, the crucial task for making the previous code work is replacing calls to Invoke (extension) method with the actual expression tree of the used lambda expression. This is done behind the scene by the wrapper created by ToExpandable in the previous code, but you can do it directly as you can see in the following example:

```
// Declare 'calc' that will be used by other lambda expressions
Expression<Func<int, int>> calc =
    i => i * 10;
// Expression that uses 'calc' (Invoke is an extension method)
Expression<Func<int, int>> test =
    i => calc.Invoke(i) + 2;
```

The first declaration in this example creates expression tree for lambda expression that multiplies its parameter by 10, later this expression can be used by other expressions (like test that calls calc and adds 2 to the result). To use expression you have to use Invoke extension method that is declared in the ExpressionExtensions class (EeekSoft.Expressions namespace). This method is very simple, because it just uses Compile method of the expression and executes the compiled delegate, but if you write calc.Compile().Invoke(i) directly to the expression it is partially evaluated while creating expression tree and it will not be possible to get expression tree of the used lambda expression.

The expansion can be done by the Expand extension method that is also declared in ExpressionExtensions. This is possible when you have variable of Expression or Expression<F> type, where F is one of the Func delegates. The following example demonstrates the expansion (some variables from previous example are used):

```
// Expand the expression using Expand extension method
Expression<Func<int, int>> expanded1 =
    test.Expand();
// You can use var because type is the same as the type of 'test'
var expanded2 = test.Expand();
// Output text representation of both expressions
Console.WriteLine("Original: {0}", test.ToString());
Console.WriteLine("Expanded: {0}", expanded2.ToString());
```

The output is following:

```
Original: i => Add(i => Multiply(i, 10).Invoke(i), 2)
Expanded: i => Add(Multiply(i, 10), 2)
```

You can see that using of Expand extension method is very simple. This example also showed that you can use var keyword and let the compiler infer the type of returned expression. This is possible because the type of returned expression is same as the type of variable, on which the extension method is invoked or in other words as the parameter passed to the method.

Now, let's examine the output printed in the previous example. The first line represents the original expression before calling the Expand method. You can see that expression calc is printed using syntax for lambda expressions as part of the test expression. The important point here is that we didn't lose the expression tree of this inner expression. The inner expression is followed by the call to Invoke method and it is part of the Add expression that represents addition in the test.

The second line represents expression tree that is created by the Expand method. You can see that inner lambda expression and call to the Invoke method was replaced by its expression tree, which is multiplication applied to the parameter and the number 10. This representation can be later converted to T-SQL, because it doesn't contain any calls to .NET methods that could not be translated. If you try to use non-expanded expression the conversion will fail on the call to the Invoke method.

Question that could came to your mind is why do you have to call expression that will later be replaced using the Invoke method instead of Compile().Invoke(..) code that does exactly the same thing. It is because when C# 3 builds the expression tree it executes the Compile method and the expression tree of the inner expression is be replaced by delegate, so it would not be possible to access the original expression tree. This is demonstrated by the following example:

```
// What happens when we use Compile() instead of Invoke()?
Expression<Func<int, int>> wrong =
    i => calc.Compile().Invoke(i) + 2;
Console.WriteLine(wrong.ToString());
```

The following output contains the delegate which is the result of **Compile** method (embedded using **value**) instead of inner expression tree that is needed for replacement:

```
i => Add(value(System.Query.Func`2
        [System.Int32,System.Int32]).Invoke(i), 2)
```

You may be also wondering whether you could write $(x \Rightarrow calc.Invoke(x) + 1).Expand()$ instead of using a variable for expression tree. If you try it, you'll get an error message saying that "Operator '.' cannot be applied to operand of type anonymous method." The problem here is that compiler needs some way to decide whether lambda expression should be returned as a delegate or as an expression tree. This depends on the type of variable to which lambda expression is assigned and in this code it is not clear what should the type of return value be. Anyway you can expand lambda expression if you call the Expand method as standard static method. In this case compiler knows what the expected type of parameter is and it can decide whether it should return delegate or expression tree:

```
// Use Expand as standard static method
Expression<Func<int, int>> ret =
  ExpressionExtensions.Expand<Func<int, int>>
    ( (int x) => calc.Invoke(x) + 2 );
// In this case type is specified so you can use var
var ret = ExpressionExtensions.Expand<Func<int, int>>
    ( x => calc.Invoke(x) + 2 );
```

This would be even more interesting if C# 3 were able to infer return type from the type of lambda expression, but this is not possible in the current version, so you have to specify type explicitly. In this case type arguments of method Expand are specified so both type of method parameter and its return value are known. Because the return type is known, it is also possible to use new var keyword for declaring variables with infered type.

3.2 DLINQ integration

The latest LINQ preview contains one very important feature that makes LINQ extensible and it allows anybody to write his own provider that takes expression tree representation of the LINQ query and executes it (for download visit the project homepage [1], for more info on the latest release see Matt Warren's weblog [3]). This extensibility is possible thanks to the IQueryable interface that contains aside from other methods, the CreateQuery method. This method is used by LINQ for building the expression trees that represent the query which is, in case of DLINQ translated to

T-SQL and executed on the database side.

I created a simple wrapper around DLINQ implementation that expands all expressions while building the query. Of course the translation to T-SQL is done by the wrapped DLINQ object. The wrapper is the ExpandableWrapper class which implements the IQueryable interface. Actually, the ExpandableWrapper is wrapper around any class that implements the IQueryable interface so you should be able to use it with any other providers that allow you to execute LINQ queries. Some of these providers were mentioned in Channel9 video [4], so in the future you should be able to use this extension with entities (in future version of ADO.NET) as well as any third party OR mapper that supports LINQ.

Lets look at the example similar to the one I presented at the beginning:

```
var q =
  from p in db.Products.ToExpandable()
  where calcPrice.Invoke(p) > 30.0m
   select p;
```

In this query, the ToExpandable method is used for creating the wrapper around DLINQ object that represents database table. When LINQ builds the expression tree that represents the query, it uses CreateQuery method of this wrapper instead of underlying DLINQ Table object. This is the trick that makes the expansion work, because the wrapper simply calls Expand method described in the previous section and than calls the CreateQuery of underlying DLINQ object. The returned value is wrapped using new instance of the ExpandableWrapper class, because when building queries that contain both where and select clauses, LINQ calls CreateQuery method of the returned object again.

This is the implementation of CreateQuery method:

```
// Creates query from expanded expression using underlying object
// (_item is underlying IQueryable passed in the constructor)
public IQueryable<S> CreateQuery<S>(Expression expression)
{
    return new ExpandableWrapper<S>
        (_item.CreateQuery<S>(expression.Expand()));
}
```

The method that creates first ExpandableWrapper is already mentioned method ToExpandable:

```
// Returns wrapper that automatically expands expressions
public static IQueryable<T> ToExpandable<T>(this IQueryable<T> q)
{
    return new ExpandableWrapper<T>(q);
}
```

4 Beta version difficulties

There are several difficulties that appeared during implementation of this project, some of them will be probably fixed before final version of the LINQ project will be released, some of them are caused by design.

First problem I'd like to mention is regarding limited scope of anonymous types. As you know, anonymous types can be used only locally inside body of one method. This prevents developers from overusing anonymous types in situations where anonymous types are not needed and would lead to unreadable code. On the other side if you want to use expression that will be later used in the **select** clause and that will

be defined globally, for example as public property of some class, you have to explicitly define its return type instead of using anonymous type as you do in all other DLINQ queries.

The second problem comes out when you want to declare expression returning anonymous type only locally inside method body. You can't use var, because you have to declare variable as either Expression or Func, so that compiler knows whether you want delegate or expression tree, but you can't declare it, because you need the name of returned anonymous type. I think that solution for this problem would be allowing something like this (though I'm not sure whether this is an acceptable solution):

```
Expression<Func<Product,_>> selector =
    p => new { p.ProductName, p.UnitPrice };
```

This can be partially solved by using method that returns **Expression**, because method type parameters can be inferred, so you it wouldn't be needed to specify return type explicitly. The problem is that in current preview type inference doesn't use information from lambda expression that you pass as parameter to method so this can't be done. The following code can't be currently compiled, but lets see whether this will be fixed in the future releases:

```
// Using this method you could declare Expression
// variables that return anonymous type
public Expression<Func<T, A0>> CreateExpression<T, A0>
 (Expression<Func<T, A0>> expr) {
    return expr;
}
// Expression with anonymous return type! (unfortunatelly current
// preview isn't able to infer type from the lambda expression)
var ret = CreateExpr( (Product p) =>
    new { p.ProductName, Price = p.UnitPrice} );
```

The minor problem is that it is currently difficult to modify expression trees. There is a class called ExpressionVisitor in the System.Query.dll, but it is currently internal. This class makes modification of expression trees quite simple, so hopefully it will be public in the future LINQ releases. For now, I used Reflector to extract this class, because I didn't want to write the same class myself and I see no reasons why the class shouldn't be public in the future.

References and downloads

Download the article in PDF (81kB) http://tomasp.net/articles/linq-expand/linq-expand.pdf Download code samples (35kB) http://tomasp.net/articles/linq-expand/demos.zip

[1] The LINQ Project - Microsoft.com http://msdn.microsoft.com/data/ref/linq[2] Query re-use and refactoring - some tips? - MSDN Forums

http://forums.microsoft.com/MSDN/ShowPost.aspx?PostID=428914

[3] Oops, we did it again - The Wayward WebLog http://blogs.msdn.com/mattwar/archive/2006/05/10/594966.aspx

[4] Chatting about LINQ and ADO.NET Entities http://channel9.msdn.com/showpost.aspx?postid=202138