

Adding LINQ-awareness to Scala.Net

© Miguel Garcia, STS, Hamburg University of Technology
<http://www.sts.tu-harburg.de/people/mi.garcia>

March 24, 2010

Contents

1	The LINQ story in .NET	2
2	Design and interoperability	3
2.1	LINQ and Entity Framework	3
2.2	Full semantics fidelity	3
2.3	Reified LINQ (Expression Trees)	4
3	Implementation notes	5
3.1	LINQ in Mono C#	5
3.2	Extending Scala syntax to account for LINQ	6
3.3	Generating statements to build expression trees	7
3.4	LINQ Expression Trees vs. Code Quotations	8

Abstract

Unlike embedded DSLs, LINQ makes available suitable syntax for object-oriented query, to simplify database access as well as in-memory queries. This write-up shows that adding LINQ support to Scala.Net on par with that of C# is not only doable but arguably beneficial.

Making Scala.Net LINQ-aware requires (a) modifying the parser so as to perform a syntax-driven desugaring into Standard Query Operators (SQO); and (b) adding a transformation to reify SQO expressions for data sources whose type indicates support for this capability (typically, to realize DBMS query shipping). Besides know-how on Scala parse and abstract syntax trees, a fairly detailed understanding of the semantics of LINQ in C# is required to extend the compiler as proposed here. For example, item (b) above involves following the same scheme as the C# compiler to reify anonymous functions (*delegates* in C# speak).

Once the proposed extension has been implemented, making use of it requires no additional expertise beyond familiarity with the same .NET tools for LINQ in C# (e.g., the *Entity Framework*).

1 The LINQ story in .NET

An exposition about the advantages of dedicated query syntax (as opposed to *embedded DSLs*) can be found in the paper *Comprehensive Comprehensions* [3] by Peyton Jones and Wadler. One particular realization of these ideas is LINQ. The C# 3.0 spec¹ states in §7.15:

Query expressions provide a language integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL and XQuery ...

*The C# language does not specify the execution semantics of query expressions. Rather, query expressions are translated into invocations of methods that adhere to the query expression pattern (§7.15.3). Specifically, query expressions are translated into invocations of methods named *Where*, *Select*, *SelectMany*, *Join*, *GroupJoin*, *OrderBy*, *OrderByDescending*, *ThenBy*, *ThenByDescending*, *GroupBy*, and *Cast*. These methods are expected to have particular signatures and result types, as described in §7.15.3. These methods can be instance methods of the object being queried or extension methods that are external to the object, and they implement the actual execution of the query.*

The translation from query expressions to method invocations is a syntactic mapping that occurs before any type binding or overload resolution has been performed. The translation is guaranteed to be syntactically correct, but it is not guaranteed to produce semantically correct C# code. Following translation of query expressions, the resulting method invocations are processed as regular method invocations, and this may in turn uncover errors, for example if the methods do not exist, if arguments have wrong types, or if the methods are generic and type inference fails.

Embedded LINQ queries are desugared into invocations on one of two interfaces defined by the .NET framework:

1. invocations on `IEnumerable`, or
2. invocations to build an `Expression` instance (Figure 2), i.e., invocations on `IQueryable`.

There are similarities with the desugaring of Scala for-comprehensions². The desugaring for case (1) above is discussed in [2] with a prototype, `LINQ4Java`. More detailed accounts are: (a) the project report by Kaichuan Wen [4] and (b) <http://blogs.msdn.com/wesdyer/archive/tags/Queries/default.aspx>

The desugaring for case (2) results in a reified *expression tree* (Sec. 2.3). For example³, the *expression compiler* of the C# compiler transforms this query:

¹<http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx>

²See comment for `TreeBuilder.makeFor()` at <http://lampsvn.epfl.ch/trac/scala/browser/scala/trunk/src/compiler/scala/tools/nsc/ast/parser/TreeBuilder.scala>

³<http://bartdesmet.net/blogs/bart/archive/2008/08/15/the-most-funny-interface-of-the-year-iqueryable-lt-t-gt.aspx>

```
var products = new Table<Product>(); // Table<T> implements IQueryable<T>
var res = from p in products where p.Price > 100 select p.Name;
```

into the following statements that build an expression tree:

```
IQueryable<Product> products = new Table<Product>();

ParameterExpression p = Expression.Parameter(typeof(Product), p);
LambdaExpression <>__predicate = Expression.Lambda(Expression.Greater(Expression.Property(p, Price), Expression.Constant(100)), p);
LambdaExpression <>__projection = Expression.Lambda(Expression.Property(p, Name), p);

var res = products.Where(<>__predicate).Select(<>__projection);
```

We can already see how a LINQ query ends up as a chain of query operator method calls that are all implemented as extension methods on `IQueryable<T>` and take in their “function parameter” as an expression tree, so that interpretation and translation can be deferred till runtime. Indeed, an expression tree representing `p => p.Price > 100` can easily be translated into a `WHERE` clause in SQL or whatever equivalent in another DSQL.

2 Design and interoperability

2.1 LINQ and Entity Framework

“Playing nice” with the *Entity Framework* (the Object/Relational Mapping approach favored by the data access product group at Microsoft) is an important goal for Scala.Net, because interoperability with *EF* amounts to interoperability with other LINQ-aware ORM solutions in .NET (they follow the same division-of-labor principles as *EF*). Given that most of the EF power comes from language-independent tooling and library-provided code, “playing nice” means “being able to reify LINQ expressions” (Sec. 2.3). Some resources on Entity Framework follow:

- <http://blogs.msdn.com/adonet/>
- http://en.wikipedia.org/wiki/ADO.NET_Entity_Framework
- What’s new with Entity Framework in Visual Studio 2010 RC⁴
- A simpler approach to data access is “LINQ to SQL”. As with EF, no special support is required beyond reifying LINQ expressions.

2.2 Full semantics fidelity

C# is for all practical purposes the reference implementation of LINQ, and we plan to follow it. However it’s not possible in general to expect that LINQ queries can be cut&pasted seamlessly between C# and Scala.Net programs: a subexpression may use language-specific constructs.

For example, C#’s nullable types cut on the boilerplate required for trashing data in and out of SQL IS NULL columns. Scala’s `Option[T]` is more general,

⁴<http://blogs.msdn.com/adonet/archive/2010/02/12/what-s-new-with-entity-framework-in-visual-studio-2010-rc.aspx>

because only a value type may be the *underlying type* for a C# nullable type. That's not a problem for SQL: columns aren't typed with reference types. Besides, nullables can't be nested. Quoting from §4.1.10 of the C# 3.0 spec:

A nullable type can represent all values of its underlying type plus an additional null value. A nullable type is written T?, where T is the underlying type. This syntax is shorthand for System.Nullable<T>, and the two forms can be used interchangeably. ... Implicit conversions are available from the null literal to T? (§6.1.5) and from T to T? (§6.1.4).

Other than that, full semantics fidelity is achieved as a result of desugaring, because the “LINQ contract” estipulates which methods (*Standard Query Operators*) are to be called on `IEnumerable` (resp. `IQueryable`) irrespective of the language in which the query is embedded (besides Scala.Net and C#, the other common case is VB.NET).

The syntax of LINQ hasn't changed since C# 3.0, but the upcoming C# 4.0 will typecheck some LINQ queries previously deemed non-valid. Both the C# 3.0 spec and the delta for C# 4.0 can be downloaded from links listed at <http://antlrsharp.codeplex.com/>. In effect, it is now possible in 4.0 to specify site-def variance for type parameters (a subset of what Scala can). Quoting from the delta document:

In .NET 4.0 the IEnumerable<T> and IEnumerator<T> interfaces will be declared in the following way:

```
public interface IEnumerable<out T> : IEnumerable ...
```

The out in these declarations signifies that T can only occur in output position in the interface, the compiler will complain otherwise.

Other issues to watch out for:

- the desugaring of `from x in e let y = f ...` involves *transparent variables* and *anonymous types*, e.g. `new { x, y = f }`. Transparent variables has to do with scoping [2] but anonymous types are more tricky to integrate cleanly in Scala's type system (which can be done, of course). The issue to watch out for is structural vs. nominal subtyping.
- a thread discussing extension methods à la C# vs. Scala implicits⁵
- the dual of `IEnumerable<T>` is `IObservable<T>`, as discussed in the series <http://lambda-the-ultimate.org/node/3822> (but this is off-topic)

2.3 Reified LINQ (Expression Trees)

Why should the compiler bother to convert LINQ syntax into statemens that build an Expression Tree? Because it's the key to interact with ADO.NET, Entity Framework, and other LINQ *query providers*. A query provider is essentially a plumbing component (e.g., a DBMS driver, the local proxy to a web service, etc.)

⁵<http://old.nabble.com/-scala--Class-overloading-vs-implicit-conversion-td17232911.html>

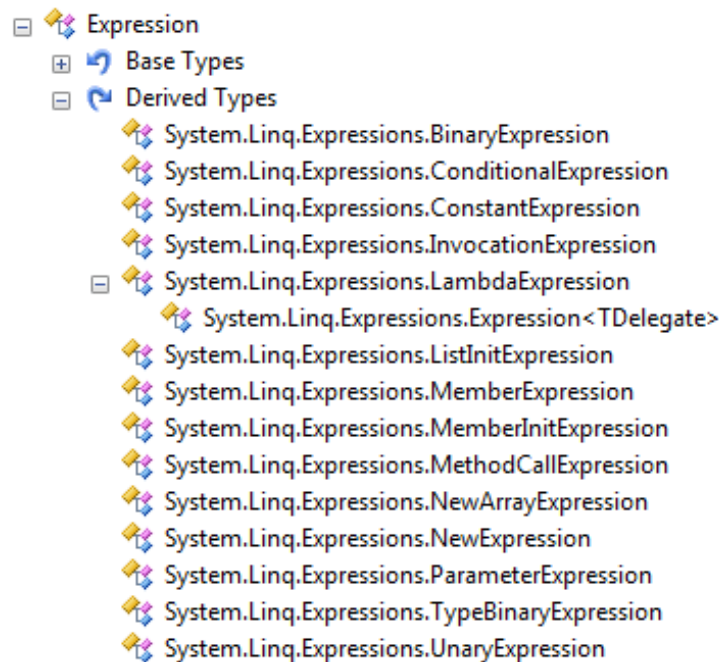


Figure 1: Instantiate these classes to have an Expression Tree

As a sidenote, the steps to write a query provider are discussed in <http://blogs.msdn.com/mattwar/pages/linq-links.aspx> Supporting LINQ in Scala.Net does not require writing any such provider. However, in case we want Scala collections to be queried using LINQ (say, by code written in another .NET language) then we have to play the provider side of the LINQ contract. This can be as direct as a wrapper or as sophisticated as an optimizing mediator [1].

Figure 2 shows an expression tree at debug time, whose nodes are instances of the AST classes (provided by the .NET framework) shown in Figure 1.

3 Implementation notes

3.1 LINQ in Mono C#

In his blog, Miguel de Icaza comments⁶ that Marek Safar completed (as of 2008) full support for LINQ in the Mono C# compiler (`gmcs`), including Expression Trees⁷ Quoting from that blog entry:

In his announcement he used Luke Hoban's brutal ray tracer-in-one-LINQ statement program. This was a hard test case for our C# compiler to pass, but we are finally there. The above now compiles and runs as fast as it does on .NET. ... Jeremie then modified the above program to use the parallel extensions to LINQ. He replaced `Enumerable.Range` with `ParallelEnumerable.Range` and `foreach` with

⁶<http://tirania.org/blog/archive/2008/Jul-26-1.html>

⁷<http://msdn.microsoft.com/en-us/library/bb397951.aspx>

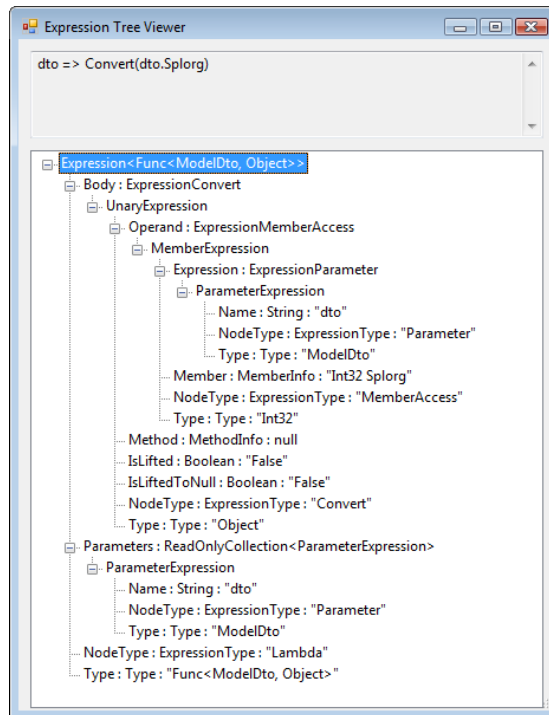


Figure 2: Debug time, `dto => Convert(dto.Splorg)`

the parallel ForAll method to take advantage of his library. You can watch the above ray tracer with and without LINQ on his screencasts.

TODO: phase where the *expression compiler* gets activated in `gmcs` (the C# compiler of Mono)

TODO Compiler tests for LINQ features can be found in the Mono `mcs` tree under `/mcs/tests/` The relevant groups of tests are `gtest-linq*` and `gtest-etree*`

3.2 Extending Scala syntax to account for LINQ

The extension to the parser has to cater for the *contextual reserved keyword from* introduced by LINQ (as discussed in §7.15.1, *Ambiguities in query expressions* of the C# 3.0 spec)

Related resources:

- The write-up *Untangling Scala ASTs*⁸ contains in §1.2 a very brief discussion about the parser.
- A term project report⁹, *Refactoring for Scala*, by Mirko Stocker describes the shapes of ASTs after parsing, naming, and typing.

⁸<http://www.sts.tu-harburg.de/people/mi.garcia/ScalaCompilerCorner/UntanglingScalaASTs1ofN.pdf>

⁹<http://scala.ifs.hsr.ch/doc/scalarefactoring-term.pdf>

- “*The parser does too much*”. There’s a tension between early desugaring and IDE-based refactoring: refactoring always takes place on the surface syntax, whose connection to ASTs may be lost during desugaring. This tension already showed up with for-comprehensions as discussed in <http://www.scala-lang.org/node/3458>. It would be great if the desugaring $\text{LINQ} \rightarrow \text{SQO}$ anticipated this scenario.
- A kind of parser generators that accomodate extensible grammars are PEG (Parsing Expression Grammars), for example *Rats!*¹⁰, used at a time to parse Scala in the NetBeans plugin¹¹

3.3 Generating statements to build expression trees

I tried googling for “Lang.NET LINQ”, “integrating LINQ in your own language”, and so on, to no avail. There have been to my knowledge no presentations at the Lang.NET symposia on this.

The IL emitted to build expression trees can be explored at runtime¹², or disassembled, as shown in Listing 1 for

```
Expression<Func<int, int>> twiceE = x => x * 2;
```

That blog entry also explains that the additional AST node types introduced in `System.Linq.Expressions v4.0` do not affect the IL to emit.

As expected, .NET Reflector makes for a more concise formulation.

```
private static void Main(string[] args)
{
    ParameterExpression expression2;
    Expression<Func<int, int>> expression =
        Expression.Lambda<Func<int, int>>(
            Expression.Multiply( expression2 = Expression.Parameter(typeof(int), "x"),
                                Expression.Constant(2, typeof(int))
                            ),
            new ParameterExpression[] { expression2 }
        );
}
```

The example above does not capture any variable, unlike the following, reproduced from <http://blogs.msdn.com/mattwar/archive/2007/08/01/linq-building-an-iqueryable-pro.aspx>

```
string city = "London";
var query = db.Customers.Where(c => c.City == city);
```

The blog entry goes on to say, “The C# compiler has made a class to hold local variables that are being referenced in the lambda expression.” The disassembler is your friend! (Figure 3). As a last example, compare with the IL generated for:

```
int howMany = 2;
Expression<Func<int, int>> twiceE = x => x * howMany;
```

¹⁰<http://cs.nyu.edu/rgrimm/xtc/rats.html>

¹¹<http://wiki.netbeans.org/ScalaImpl>

¹²<http://community.bartdesmet.net/blogs/bart/archive/2009/08/10/expression-trees-take-two-introducing-system-linq-expressions-v4-0.aspx>

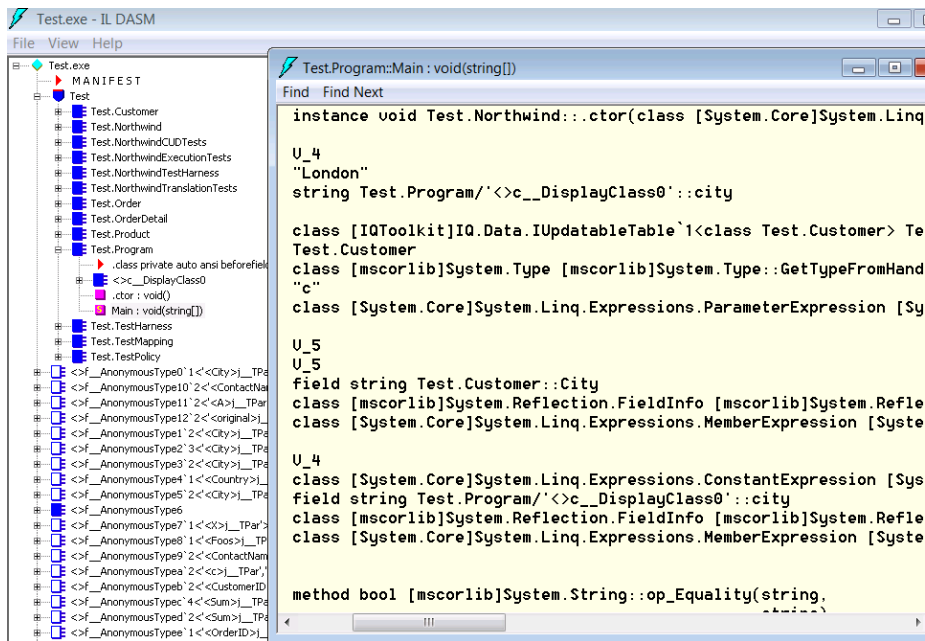


Figure 3: `db.Customers.Where(c => c.City == city)`

Interestingly, when .NET Reflector is asked to decompile into v2.0 of C#, the following is obtained for the "London" example (i.e., no dedicated class in sight to hold the captured variable).

```
string city = "London";
IQueryable<Customer> queryable = d
    b.Customers.Where<Customer>(
        Expression.Lambda<Func<Customer, bool>>(
            Expression.Equal(
                Expression.Field(expression = Expression.Parameter(typeof(Customer), "c"), fieldof(Customer.City)),
                Expression.Constant(city), false, (MethodInfo) methodof(string.op_Equality)),
            new ParameterExpression[] { expression }
        ));
```

TODO I guess this needs more exploration :-)

- *TreeDSL* implements the Builder Pattern to simplify piecing together Scala ASTs, as needed when reifying SQO expressions. *TreeDSL* is easy enough to use, and is covered in at least two write-ups at *The Scala Compiler Corner*: the one on the Cleanup phase and *How the Scala-to-Java translator copes with Pattern Matching*.

3.4 LINQ Expression Trees vs. Code Quotations

I'm including some references to quotations not because they are needed to realize Expression Trees à la LINQ (they are not) but for completeness, and because someone might ask what connection there is (none).

A few discussions on code quotations follow (if you know more detailed re-

sources, please let me know). There's also a thread on the future of `scala.reflect.Code`¹³.

- <http://old.nabble.com/-scala--LINQ-for-scala-td25203779.html>

```
scala> import scala.reflect._
import scala.reflect._

scala> val param = LocalValue(NoSymbol, "x", NamedType("scala.Int"))
param: scala.reflect.LocalValue = LocalValue(NoSymbol,x,NamedType(scala.Int))

scala> val tree = Function(List(param), Apply(Select(Ident(param), LocalValue(param, "*", NoType)), List(
tree: scala.reflect.Function = Function(List(LocalValue(NoSymbol,x,NamedType(scala.Int))),Apply(Select(Ident
```

- <http://harrah.github.com/browse/samples/compiler/scala/tools/nsc/transform/LiftCode.scala.html>

```
package scala.tools.nsc
package transform
/** Translate expressions of the form reflect.Code.lift(exp)
 * to the lifted "reflect trees" representation of exp.
 *
 * @author Gilles Dubochet
 * @version 1.0
 */
abstract class LiftCode extends Transform with Reifiers {
```

- “off-topic” but anyway I’ll mention this to avoid confusion. In v4, expression trees have been extended to be able to represent general imperative code, as summarized in <http://community.bartdesmet.net/blogs/bart/archive/2009/08/10/expression-trees-take-two-introducing-system-linq-expressions.aspx>. There’s also a 100-page description: <http://dlr.codeplex.com/Project/Download/FileDownload.aspx?DownloadId=85661>.

References

- [1] Nicolas Bruno and Pablo Castro. Towards declarative queries on adaptive data structures. In *Intl. Conf. on Data Engineering*, pages 1249–1258, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
- [2] Miguel Garcia. Compiler plugins can handle nested languages: AST-level expansion of LINQ queries for Java. In Moira C. Norris and Michael Grossniklaus, editors, *Proc. of ICODDB 2009*, pages 41–58, July 2009. <http://www.sts.tu-harburg.de/people/mi.garcia/pubs/2009/icoodb/compplugin.pdf>.
- [3] Simon Peyton Jones and Philip Wadler. Comprehensive Comprehensions. In *Proc. of the ACM SIGPLAN Workshop Haskell '07*, pages 61–72, New York, NY, USA, 2007. ACM Press. <http://research.microsoft.com/~simonpj/papers/list-comp/list-comp.pdf>.
- [4] Kaichuan Wen. Translation of Java-embedded database queries, with a prototype implementation for LINQ, 2009. <http://www.wen-k.com/files/JavaEmbeddedLinq.PDF>.

¹³<http://old.nabble.com/Future-of-scala.reflect.Code-td27152268.html>

Listing 1: IL emitted by the CSharp compiler for Expression<Func<int, int>> twiceE = x => x * 2;

```

.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // Code size 68 (0x44)
    .maxstack 4
    .locals init ([0] class [System.Core]System.Linq.Expressions.Expression'1<class [mscorlib]System.Func'2<int32,int32>> twiceE,
    [1] class [System.Core]System.Linq.Expressions.ParameterExpression CS$0$0000,
    [2] class [System.Core]System.Linq.Expressions.ParameterExpression[] CS$0$0001)
    IL_0000: nop
    IL_0001: ldtoken [mscorlib]System.Int32
    IL_0006: call class [mscorlib]System.Type [mscorlib]System.Type::GetTypeFromHandle(valuetype [mscorlib]System.RuntimeTypeHandle)
    IL_000b: ldstr "x"
    IL_0010: call class [System.Core]System.Linq.Expressions.ParameterExpression [System.Core]System.Linq.Expressions.Expression::Parameter(class [mscorlib]System.Type,
    string)
    IL_0015: stloc.1
    IL_0016: ldloc.1
    IL_0017: ldc.i4.2
    IL_0018: box [mscorlib]System.Int32
    IL_001d: ldtoken [mscorlib]System.Int32
    IL_0022: call class [mscorlib]System.Type [mscorlib]System.Type::GetTypeFromHandle(valuetype [mscorlib]System.RuntimeTypeHandle)
    IL_0027: call class [System.Core]System.Linq.Expressions.ConstantExpression [System.Core]System.Linq.Expressions.Expression::Constant(object,
    class [mscorlib]System.Type)
    IL_002c: call class [System.Core]System.Linq.Expressions.BinaryExpression [System.Core]System.Linq.Expressions.Expression::Multiply(class [System.Core]System.Linq.Expressions.Expression,
    class [System.Core]System.Linq.Expressions.Expression)
    IL_0031: ldc.i4.1
    IL_0032: newarr [System.Core]System.Linq.Expressions.ParameterExpression
    IL_0037: stloc.2
    IL_0038: ldloc.2
    IL_0039: ldc.i4.0
    IL_003a: ldloc.1
    IL_003b: stelem.ref
    IL_003c: ldloc.2
    IL_003d: call class [System.Core]System.Linq.Expressions.Expression'1<!10> [System.Core]System.Linq.Expressions.Expression::Lambda<class [mscorlib]System.Func'2<int32,int32>>(class [System
    class [System.Core]System.Linq
    IL_0042: stloc.0
    IL_0043: ret
} // end of method Program::Main

```