re-linq
A General Purpose LINQ Foundation

Fabian Schmied, rubicon informationstechnologie gmbh, 2009-09-22

Introduction

With .NET 3.5, Microsoft introduced a new technology called *Language Integrated Query,* short *LINQ*. It is both a feature of the .NET framework and of the major .NET programming languages C# and Visual Basic .NET. It allows programmers to write expressions in a language-integrated query expression syntax, which get translated into an abstract syntax tree (AST) representation available for analysis at runtime. A provider model allows for external components (called *LINQ Providers*) to get involved into that analysis process

The .NET framework version 3.5 contains a simple object/relational (O/R) mapper called *LINQ to SQL* (implemented in the namespace *System.Data.Linq*), which acts as a LINQ provider. It includes the facilities of transforming the AST representations generated for user-defined queries into Microsoft SQL Server-specific queries. LINQ to SQL has a very tight coupling between the O/R mapper and the AST transformation, with all possible extension points being internal to the .NET framework, so although it is a very complete LINQ provider, it cannot be easily reused by third parties.

In addition to LINQ to SQL, Microsoft also offers LINQ providers for the ADO.NET Entity Framework *(LINQ to Entities)*, for in-memory XML trees *(LINQ to XML)*, and for in-memory collections *(LINQ to Objects)*.

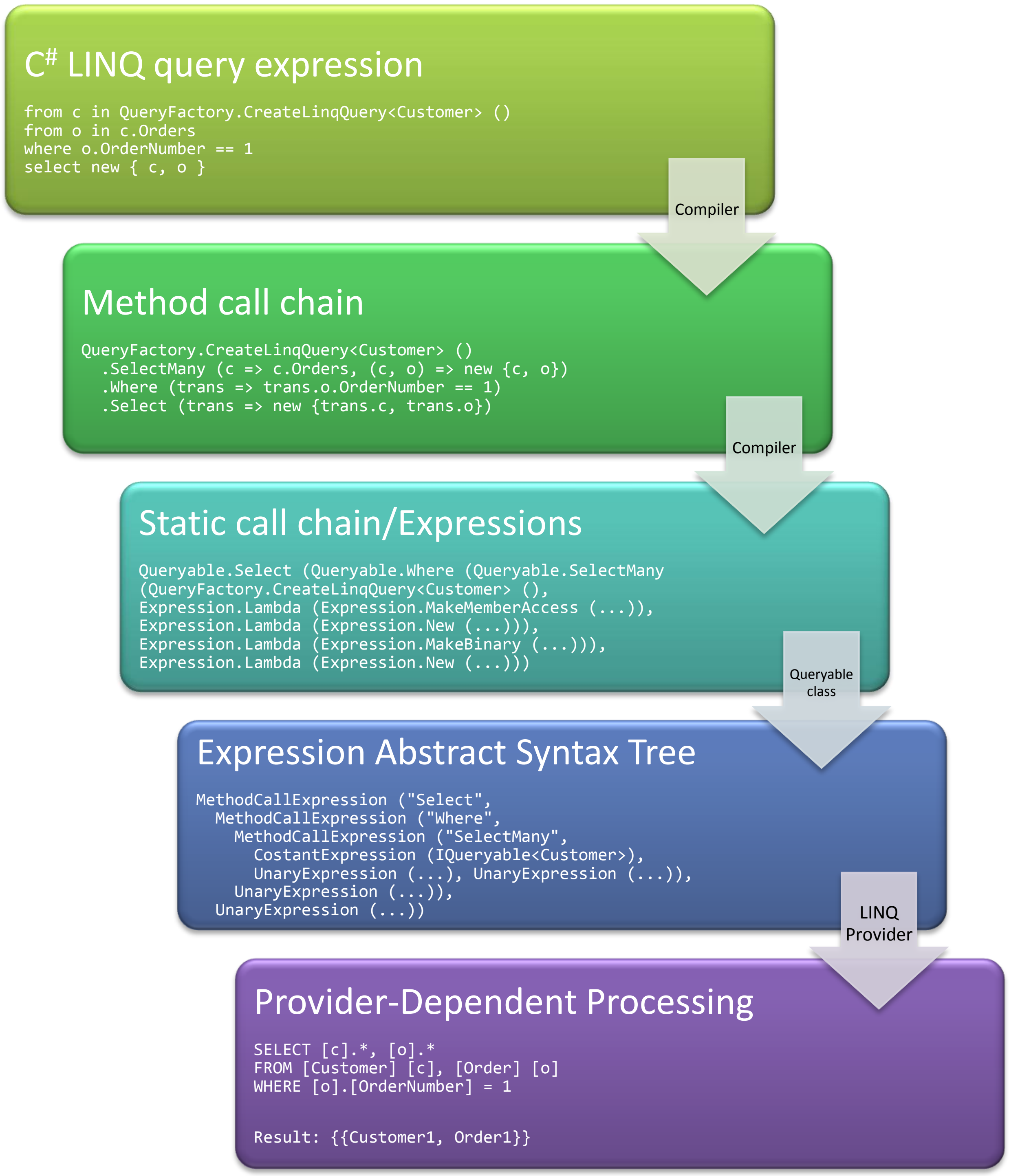
Apart from these, there is need for additional LINQ providers:

- to enable the language-integrated query features for other O/R mappers, query languages, and (non-relational) data sources (such as Lucene, LDAP, …);
- to enable mixed queries (such as a query to be executed partially by Microsoft SQL Server and partially by Lucene, or partially by SQL Server and partially in-memory);
- or even to enable query generation for relational databases other than Microsoft SQL Server.

While the LINQ provider model is powerful enough to make all of this possible, creating a new LINQ provider is not a simple task. To understand the difficulties, one first needs to comprehend how exactly a LINQ query is processed by a LINQ provider.

Query Execution with LINQ Providers

When a C# programmer writes a LINQ query expression, there is a lot of work going on before the query can be executed, as illustrated in Figure 1. First, the compiler will transform the language-integrated query expression into a chain of method calls. For example, a "select" clause will (usually, but not always) be translated into a call to the *Queryable.Select* extension method, a "where" clause will be translated into a call to *Queryable.Where*, an additional "from" clause will become a call to *Queryable.SelectMany*, and so on. Note that the compiler needs to insert parameters ("trans", so-called *transparent identifiers*); these identify objects flowing from one method call to the next.


C# LINQ query expression
Compiler
Method call chain
Compiler
Static call chain/Expressions
Queryable class
Expression Abstract Syntax Tree
LINQ Provider
Provider-Dependent Processing

```
from c in QueryFactory.CreateLinqQuery<Customer> ()
from o in c.Orders
where o.OrderNumber == 1
select new { c, o }
```

```
QueryFactory.CreateLinqQuery<Customer> ()
  .SelectMany (c => c.Orders, (c, o) => new {c, o})
  .Where (trans => trans.o.OrderNumber == 1)
  .Select (trans => new {trans.c, trans.o})
```

```
Queryable.Select (Queryable.Where (Queryable.SelectMany
(QueryFactory.CreateLinqQuery<Customer> (),
Expression.Lambda (Expression.MakeMemberAccess (...)),
Expression.Lambda (Expression.New (...))),
Expression.Lambda (Expression.MakeBinary (...))),
Expression.Lambda (Expression.New (...)))
```

```
MethodCallExpression ("Select",
  MethodCallExpression ("Where",
    MethodCallExpression ("SelectMany",
      CostantExpression (IQueryable<Customer>),
      UnaryExpression (...), UnaryExpression (...)),
    UnaryExpression (...)),
  UnaryExpression (...))
```

```
SELECT [c].*, [o].*
FROM [Customer] [c], [Order] [o]
WHERE [o].[OrderNumber] = 1

Result: {{Customer1, Order1}}
```

Figure 1. Steps in processing a LINQ query expression

In Figure 1, that call chain is written using extension methods for clarity, but the actual calls are made on static methods defined by the *Queryable* class. For an extension method call chain of *"a.SelectMany (b, c).Where (d).Select (e)"*, the static call chain actually executed is *"Queryable.Select (Queryable.Where (Queryable.SelectMany (a, b, c), d), e)"*. Also, these methods have a specific feature: their arguments defining *select projections*, *where conditions*, and similar parts of the query are not directly lambda expressions. Instead, these methods take instances of *Expression<Func<T>>,* an abstract representation of lambda expressions. The compiler recognizes this and uses the *Expression.Lambda* factory method to construct partial syntax trees made of *Expression<Func<T>>* instances equivalent to the lambda expressions passed to the methods.

When the methods of the *Queryable* class are executed at program run-time, they do not immediately execute the query; instead, they use the expression syntax trees passed to them to construct an abstract syntax tree for the full query. In Figure 1, this can be seen to be a nested series of *MethodCallExpressions.* The *LambdaExpression* nodes from the previous step are wrapped into *UnaryExpressions* objects.

When the query result is enumerated, the abstract syntax tree for the full query is processed by the LINQ provider, which analyzes it, usually creates an internal model of the query, executes it, and returns the query results.

This last step is the difficult part in writing a custom LINQ provider; it will be discussed in the next section.

The Difficulty in Writing LINQ Providers

The main problem in writing LINQ providers is to analyze and understand the structure of the AST generated by the compiler for a given LINQ query.

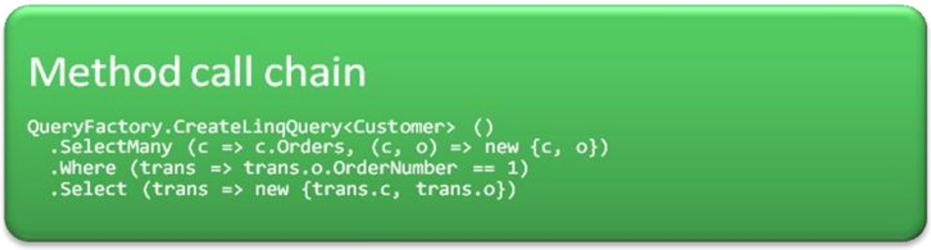
For example, when a programmer writes *"from c in QueryFactory.CreateLinqQuery<Customer> ()"*, as in Figure 1, a SQL-generating LINQ provider would have to generate SQL similar to "FROM [CustomerTable] [c]". For *"from o in c.Orders"*, it would generate a SQL *JOIN*, and for *"where o.OrderNumber == 1"*, it would generate a SQL *WHERE* clause. To perform these generations, the provider first needs to understand that the first *ConstantExpression* in the call chain corresponds to the main *from* clause, a call to *SelectMany* corresponds to an additional *from* clause, a call to *Where* corresponds to a *where* clause, and a call to *Select* corresponds to the *select* part of the query statement. We call this the *structural parsing* part of understanding a LINQ abstract syntax tree. The difficulties in this step lie in the facts that:

- Calls to the same methods can mean different things; for example, *SelectMany* can stem from an additional *from* clause or also a sub-query, *Select* can stem from a *select* clause or also a *let* clause;
- expression trees get very complex very quickly, and parsers need to prepare for the fact that *any* query method can follow any other query method, allowing for an exorbitant number of combinations that the parser needs to recognize and handle; and
- expression trees can contain calls to any user-defined methods, so parsers need to be prepared for the possibility of encountering methods they do not recognize.

After the structure has been determined, the provider needs to understand the lambda expressions (in their *Expression<Func<T>>* form) representing the conditions of *where* clauses, the projections of *select* clauses, and so on. We call this the *detail parsing* part of understanding a LINQ AST.

This is difficult because those lambda expressions can reference data coming from clauses prior to the current clause in the call chain. Such data is passed to the lambda expressions via parameters (often *transparent identifiers,* as explained in the previous section).

For example, in Figure 1, the *SelectMany* method call combines the data coming from the main *from* clause in the query with the data being added by the additional *from* clause. The method call chain illustrating this is reprinted in Figure 2.



Method call chain

```
QueryFactory.CreateLinqQuery<Customer> ()
  .SelectMany (c => c.Orders, (c, o) => new {c, o})
  .Where (trans => trans.o.OrderNumber == 1)
  .Select (trans => new {trans.c, trans.o})
```

Figure 2. The method call chain corresponding to the LINQ query expression

This combination of data is done by a lambda expression that returns an instance of an anonymous type, storing in it both the *Customer c* it gets from the main *from* clause and the *Order o* added by the additional *from* clause. That instance then goes to the *Where* method call as a transparent identifier. The *Where* method's condition lambda expression accesses the *Order* by accessing the transparent identifier's *o* member. Note that the C# compiler preserves the naming of the objects involved, i.e. both the member of the transparent identifier and the lambda parameter of the *SelectMany* clause are called *o* because the variable was originally called *o* in the query expression. In general, however, it is not guaranteed that parameters and transparent identifier members have corresponding names, so a provider cannot rely on that.

Parsing the *where* conditions, *select* projections, and similar lambda expressions contained in the query's AST is therefore hard because in order to determine what objects are actually used in such an expression, the provider needs to follow the lambda parameters, including transparent identifiers, through the method call chain, untangling the member access expressions, seeking to build up a "path" to the original query sources. Only with this path it can decide how to execute a query corresponding to the expressions written by the user.

All in all, writing a custom LINQ provider with a non-trivial feature set is a non-trivial task. Citing Frans Bouma, author of LLBLGen Pro (an O/R mapper, which – according to its author – has one of the very few *full* LINQ Providers):

Writing a Linq provider is a lot of work which requires a lot of code. If you're dealing with a Linq provider which is just, say, 32KB in size, you can be sure it will not support the majority of situations you will run into. However, the O/R mapper developer likely simply said 'We have Linq support', and it's even likely the provider can handle the more basic examples of a single entity type fetch with a Where, an Order By or even a Group By. But in real life, once you as a developer have tasted the joy of writing compact, powerful queries using Linq, you will write queries with much more complexity than these Linq 101 examples. Will the Linq 'provider' you chose be able to handle these as well? In other words: is it a full Linq provider or, as some would say, a 'toy' ?

(http://weblogs.asp.net/fbouma/archive/2008/06/17/linq-to-llblgen-pro-feature-highlights-part-1.aspx)

Calling for a General Purpose LINQ Foundation

Most existing LINQ providers follow the (bad) example of *LINQ to SQL* in that they:

- Are bound to a very specific O/R mapper, rendering re-use with other mappers or for non-database scenarios impossible; and/or
- are bound to a specific query system such as SQL Server, Oracle, or others, rendering reuse with other systems (which needn't even be relational database systems) impossible.

This leads to the fact that the complex logic of parsing and understanding the LINQ ASTs is *duplicated* in all of these providers, often only implemented in a half-hearted way (e.g. supporting only the most basic query structures, not allowing usage of transparent identifiers, or relying on naming conventions employed by the C# compilers).

Wouldn't it be much more intelligent, then, to implement that parsing logic *only once* in a generic way, and have all other LINQ providers reuse it?

re-linq - A General Purpose LINQ Foundation

The idea of *re-linq* is to implement the difficult parts of parsing and understanding the abstract syntax tree generated by LINQ query expressions once and to be reused for all kinds of purposes. Its goal is to provide a semantically rich, interlinked model of the LINQ query, so that other, specific LINQ providers can take that model to build and execute their query. Figure 3 illustrates that schema.

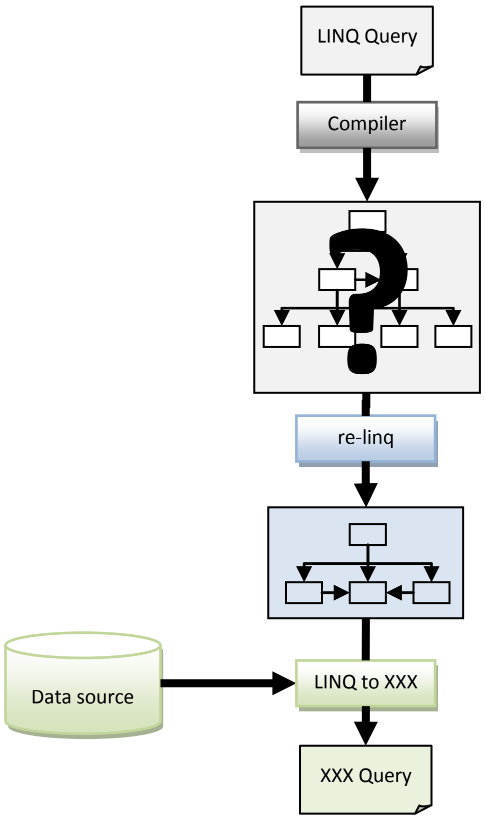


Figure 3. A LINQ Query Being Transformed with re-linq

What re-linq Is... and Is Not

It is important to note that while re-linq takes over much of the complexity a LINQ provider needs to deal with, it is neither a full LINQ provider nor an O/R mapper. A full LINQ provider needs to actually execute the query written by the user and return a result set of objects; an O/R mapper needs to define the mappings between entity classes in the application and tables in the database.

re-linq implements neither query execution nor O/R mapping because there is no need to do so. There are plenty of O/R mappers in the .NET ecosystem, and they usually come with their own querying mechanisms. re-linq does not want to compete with any of them, and it also does not want to be constrained to O/R mapping (or even to relational databases).

Instead, re-linq's goal is to solely act *as a base* for other LINQ providers, taking away the pain of analyzing the abstract syntax tree produced by LINQ. It provides a well-defined interface towards arbitrary O/R mappers and other data sources, and well-defined points to hook into for the actual query generation and execution.

Multi-Stage Translation of LINQ Queries

re-linq's transformation mechanism works in multiple stages:

First, the *structural parsing* stage analyzes the LINQ AST, identifying *select, from, where, order by, group by,* as well as custom clauses and subqueries. This stage constructs a *Query Model*, which holds instances of *SelectClause, MainFromClause, AdditionalFromClause, LetClause, OrderByClause, GroupByClause,* and other custom *Clause* objects. Figure 4 illustrates how the LINQ query from Figure 1 can be represented by such a *QueryModel*.

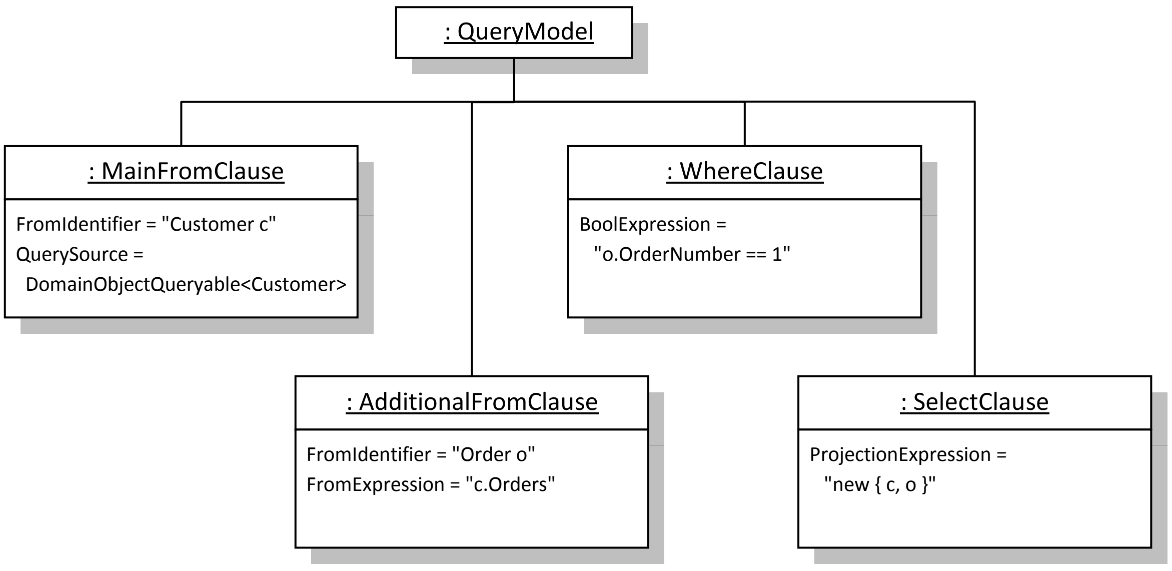
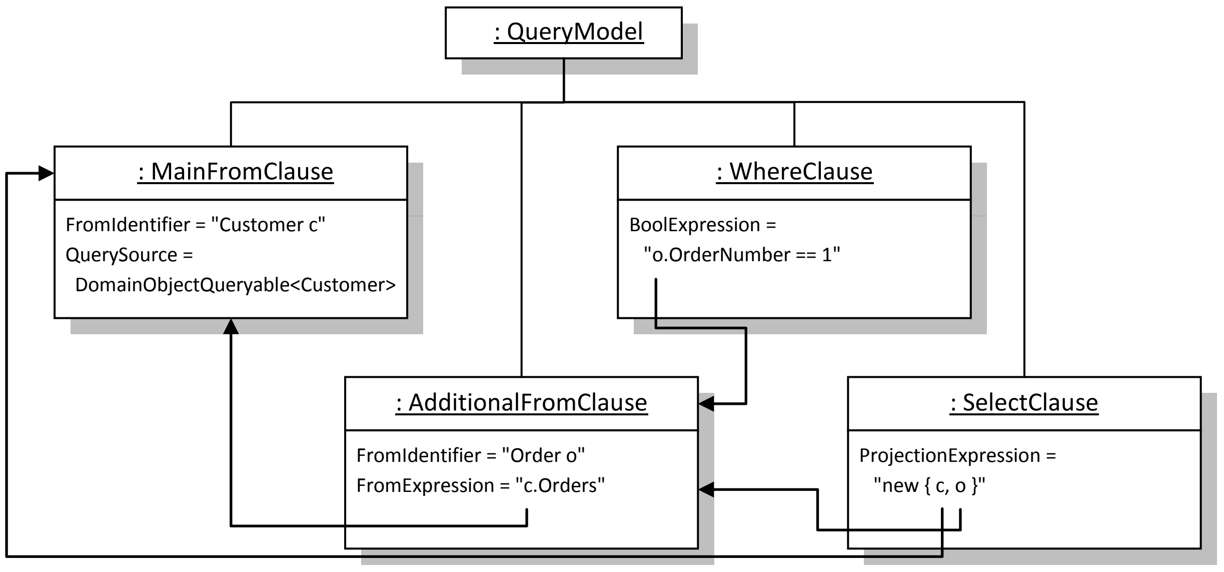


Figure 4. The QueryModel for the query from Error! Reference source not found.

Next, the *detail parsing* stage analyzes the expressions used by the different clauses, for instance, the projection expression from a *select* clause, or the condition expression from a *where* clause. It builds a *Data Model* for each of these expressions, linking the values being used with their originating clauses, resolving property paths (thus removing transparent identifiers), and partially evaluating those expressions that do not involve external data. For example, the *o.OrderNumber* expression from the *where* clause from Figure 1 (*"where o.OrderNumber == 1"*) would be represented by a link to the from clause defining *o* (*"from o in c.OrderItems"*). This interlinking is illustrated in Figure 5.



: QueryModel
: MainFromClause
FromIdentifier = "Customer c"
QuerySource = DomainObjectQueryable<Customer>
: WhereClause
BoolExpression = "o.OrderNumber == 1"
: AdditionalFromClause
FromIdentifier = "Order o"
FromExpression = "c.Orders"
: SelectClause
ProjectionExpression = "new { c, o }"

Figure 5. The QueryModel with interlinked clauses

The *Query Model* and *Data Model* are then passed to a *query executor*, which must be supplied by the specific LINQ provider. The query executor will begin the *custom query generation* stage, where it generates a custom query for its data source and executes it, returning the query results to re-linq, which in turn hands them to the user.

Due to the simple nature of the *Query Model* and the high interlinking provided by the *Data Model*, translation of the clauses and expressions into custom queries is usually very straight-forward. re-linq implements the visitor pattern for both the *Query Model* and the *Data Model* in order to allow for double-dispatch processing of the respective query elements.

The current re-linq codebase contains an exemplary implementation of custom query generation for Microsoft SQL Server queries (with other SQL dialects to be supplied on demand) that can be reused by specific LINQ providers based on re-linq. If this exemplary SQL generation engine is to be reused, it needs to interface with an underlying O/R mapper in order to map the entity types to tables, the fields to columns, and the property paths to joins. Using this with a custom O/R mapper is achieved via a simple implementation of the *IDatabaseInfo* interface shown in Figure 6.

```
public interface IDatabaseInfo
{
  string GetTableName (FromClauseBase fromClause);
  string GetRelatedTableName (MemberInfo relationMember);
  string GetColumnName (MemberInfo member);
  Tuple<string, string> GetJoinColumnNames (MemberInfo relationMember);
  object ProcessWhereParameter (object parameter);
  MemberInfo GetPrimaryKeyMember (Type entityType);
  bool IsTableType (Type type);
}
```

Figure 6. The *IDatabaseInfo* interface

All in all, the process of generating and executing custom queries based on re-linq is much easier than working directly on the LINQ AST, as re-linq can completely supply the first two stages (structural parsing and detail parsing), and assist in the third stage (custom query generation). For LINQ providers generating SQL, adoption of re-linq only requires to supply an implementation of *IDatabaseInfo* in order for re-linq to be able to interact with the O/R mapper.

Current Status and Outlook

The re-linq idea is not only a concept, it is already implemented as part of the re-motion project (https://svn.re-motion.org/svn/Remotion/trunk/Remotion/Data/Linq/). It is a project of currently 4.000 lines of code (plus unit tests), which has been developed completely in a test-driven way. It is ready for use right now, as we have demonstrated by implementing the LINQ provider of re-motion's own O/R mapper (re-store) on top of re-linq.