There Is No Impedance Mismatch (Language Integrated Query In Visual Basic 9)

Erik Meijer
Microsoft SQL Server
emeijer@microsoft.com

Abstract

Language Integrated Query (LINQ) is a framework that is rooted in the theoretical ideas of monads and monad comprehensions to allow seamless querying over objects, XML, and relational data. Instead of blindly gazing at the perceived impedance mismatch between the *structure* of these various data models, LINQ leverages the commonalities between the *operations* on these data models to achieve deep semantic integration.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Contructs and Features – *frameworks.* Language Classifications – *Applicative languages, Object-oriented languages.* H.2.3 [**Database Management**]: Languages – *Database (persistent) languages, Query languages.*

General Terms Algorithms, Design, Languages, Theory.

1. Introduction

Today's three most prevalent data models each come tightly coupled with a query language. The relational model has sets of tuples and SELECT-FROM-WHERE queries. The XML model has ordered sets of nodes and FROM-LET-WHERE-ORDERBY-RETURN queries. The OO model has a wide variety of collections and an even larger assortment of looping and conditional control structures.

What is common in each of these models is the notion of "collection" of items that can be filtered, transformed, and flattened. In addition, each data model provides domain-specific operations (non-proper morphisms). The collection type, together with the (second-order) transformation, forms a *monad* or *Standard Query Operators* in LINQ-speak.

The real payoff from the monadic approach comes from defining syntactic sugar over the underlying monadic operations via query comprehensions. This is very similar to the usual translation of SQL into relational algebra. However, instead of dealing with just sets of tuples, query comprehensions work over any data model that supports the monad pattern.

2. LINQ to Objects

Perhaps the simplest example monads in the context of an OO language such as Visual Basic, C# or Java are co-inductive (lazy) collections of the interface `IEnumerable(Of T)`. This interface defines a single method `GetEnumerator()` that returns an iterator over the collection, which is exposed via the generic interface `IEnumerator(Of T)` that contains the two members `MoveNext() As Boolean` and `Current As T`. The consumer of the collection repeatedly calls `MoveNext` and `Current` to advance from one item in the collection to the next until eventually `MoveNext` returns `False`.

The following query selects the names and phone numbers of all customers older than `42` given an enumerable source collection of customers.

```
Dim Customers
  As IEnumerable(Of Customer) = …

Dim Q As IEnumerable(Of
  { Name As String, Phone As Integer}) =
    From C In Customers
    Where C.Age > 42
    Select C.Name, C.Phone
```

The query gets translated into the following calls to the static methods defined in the `System.Query.Sequence` class that implement the standard query operators on `IEnumerable(Of T)`:

```
Select
 ( Where
     ( Customers
     , Function(C) C.Age > 42
     )
 , Function(C)
     New With { C.Name, C.Phone }
 )
```

The `Where` operator removes all items from the source collection that do not satisfy the given predicate

```
Shared Function Where(Of T)
   ( Src As IEnumerable(Of T)
   , Pred As Func(Of T, Boolean)
   )
As IEnumerable(Of T)
```

The `Select` operator applies a function to each item in the source collection. Similarly, the `SelectMany` operator applies a function that returns a collection and flattens the results into a single target collection.

Functional programmers will recognize `Where` as the standard filter function, `Select` as map, and `SelectMany` as monadic bind. The actual implementation in Visual Basic, Java or C# is standard.

OOPSLA'06, October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

```
Shared Function Select(Of T,S)
  ( Src As IEnumerable(Of T)
  , F As Func(Of T, S)
  )
As IEnumerable(Of S)

Shared Function SelectMany(Of T,S)
  ( Src As IEnumerable(Of T)
  , F As Func(Of T, IEnumerable(Of S))
  )
As IEnumerable(Of S)
```

3. LINQ To SQL

In our example query, the lambda expression `Function(C) C.Age > 42` is implicitly converted to a delegate (the intrinsic representation of higher-order functions in .NET) of type `Func(Of Customer, Boolean)`.

An alternative implementation of the standard query operators, based on the `IQueryable(Of T)` collection type, passes an *intensional* representation of the predicate (and selector functions) as an expression tree to the `Where`, `Select` and `SelectMany` methods:

```
Shared Function Where(Of T)
   ( Src As IQueryable(Of T)
   , Pred As Expression(Of
      Func(Of T, Boolean))
   )
As IQueryable(Of T)
```

The implementation of `Where` can use the expression tree to build up concrete representation of the complete query. Using syntax for quasi-quoting with ⌜ ⌝ for quote and ⌞ ⌟ for unquote, the following example implementation of the `Where` operator reconstructs an explicit representation of itself:

```
Shared Function Where(Of T)
   ( Src As IQueryable(Of T)
   , Pred As Expression(Of
      Func(Of T, Boolean))
   )
As IQueryable(Of T)
   Return ⌜Where(⌞Src⌟, ⌞Pred⌟)⌝
End Function
```

The conversion from `IQueryable` to `IEnumerable` can now look at the complete query in order to produce the requested result. The *LINQ to SQL* and *LINQ to EDM* (ADO.Net vNext) implementations of `IQueryable` implements this by compiling the query expression to SQL and executing the resulting program on a SQL database back-end via standard ADO.Net connection.

Like most O/R mapping infrastructures, the *LINQ to SQL* implementation also maintains a context for mapping relational tuples to objects, keeping track of changes, and interacting with the database transaction manager.

4. LINQ To XML

To produce query results we must be able to create complex values using expressions instead of via a sequence of imperative commands.

To allow for this, both Visual Basic 9 and C# 3.0 introduced the notion of object initializers, or object literals. Constructing complex object instances using expressions should be familiar to anyone that has used functional languages with algebraic data types such as Haskell or SML or JavaScript with object literals.

For instance, in order to return a business-card object from our example query, we can use the following object intializer expression:

```
Dim Q =
  From C In Customers
  Where C.Age > 42
  Return New BusinessCard With {
           .Person = With { C.Name },
           .Contact = With {
                C.Phone, C.Email }
         }
```

Note that this query uses the `Return` operator since it creates a collection of `BusinessCard` values as opposed to the `Select` operator that returns a collection of tuples.

Another situation where we need to return complex expressions is when we are generating XML from a query. Unfortunately, the W3C XML DOM does not support expression-based construction and hence it is not a natural fit for LINQ. The *LINQ to XML* framework is a new API for querying and manipulating XML designed specifically to complement LINQ.

Visual Basic 9 adds further syntactic sugar on top of LINQ to XML for constructing XML literals, and for accessing XML axis members.

Inside XML literals, the brackets `<%= %>` unquote embedded Visual Basic expressions in any position where the underlying API allows us to pass a computed value:

```
Dim Cards =
  From C In Customers
  Where C.Age > 42
  Return <Business-Card>
           <Person>
             <Name>
               <%= C.Name %>
             </Name>
           </Person>
           <Contact
              Email=<%= C.Email %>>
              <Phone Work="True">
                <%= C.Phone %>
              </Phone>
           </Contact>
         </Business-Card>
```

XML axis members provide XPath-like accessors over XML documents. The *child axis* `Cards.<Person>` selects all the direct child elements named `"Person"` from `Cards`. The *descendant axis* `Cards...<Phone>` selects all descendant elements named `"Phone"` from the `Cards` document. Finally the *attribute axis* `Cards...<Phone>.@Work` selects all `"Work"` attributes from all `"Phone"` elements in the `Cards` document.

Acknowledgements

We would like to thank the Visual Basic, C#, LINQ to SQL, LINQ to XML, and the Tesla I and II teams for their support and inspiration