XLinq: XML Programming Refactored (The Return Of The Monoids)

Erik **Meijer**

Brian **Beckman**

Abstract

New programming languages such as XQuery [XQuery], XJ [XJ], and Xtatic [Xtatic] are changing the landscape of XML programming by providing deep language support for querying and constructing XML. By carefully factoring the fundamental principles underlying these languages, we can both reapply XML query technologies to other data models, and simplify construction and manipulation of XML values in languages that do not natively support XML.

Most query languages, including XQuery and SQL, can be factored into (1) general operations on collections, and (2) domain-specific operations on the elements of these collections. Examples of general operations on collections, are mapping functions over collections (projection), filtering (removing) elements from collections, grouping (partitioning) collections, sorting collections, and aggregating (folding) other operations over collections to reduce them to values.

Examples of domain-specific operations on elements of XML collections include selecting down the structure axes (children, attributes, descendants, siblings, etc.), and constructing new elements and attributes and attaching them under existing XML nodes.

Database [Comprehensions] and language researchers for the likes of Haskell [Haskell], Scala [Scala], Python [Python], Comega [Comega], long ago discovered that generic operations on collections are all instances of the concept of *monoids* or *monads* [Monads]. Considered as such, they satisfy many coherent algebraic properties and allow elegant syntactic sugar in the form of query comprehensions. For example, FLOWR expressions in XQuery are a form of query comprehension [ICDT 2001] and are not in any way specialized to operating over collections of XML nodes. On the contrary, FLOWR expressions would be equally useful both in ordinary programming for complex queries over collections of objects and in relational database systems for queries over tables of rows.

We propose in this paper that general-purpose languages should extend their query capabilities based on the theory and established practice of monads, allowing programmers to query any form of data, using a broad notion of collection. This proposal is in opposition to the common practice of inserting specialized sub-languages and APIs for querying XML one way, databases another way, and objects a third way. The LINQ project at Microsoft leads by example, providing basic monadic operations on collections of arbitrary origin, on top of which C# and Visual Basic programming languages have implemented generic query comprehension syntaxes on top of LINQ.

With respect to domain-specific XML operations, we notice that current APIs -such as the DOM- are very imperative, highly complex, and irregular when compared to (1) the regularity of the XPath axis view and (2) expression-oriented element and attribute construction as offered by XML-centric languages. The current DOM is also rather heavyweight. We propose an alternative, lightweight, rational, and simple API, called XLinq, designed specifically to mesh with the general query infrastructure of LINQ. XLinq is expression-oriented rather than imperatively statement-oriented like the

DOM, and supports node-centric rather than document-centric creation of XML, allowing the structure of code to mirror the structure of data.

Table of Contents

1. Introduction

XML has become so mainstream that many people are proposing either to create special purpose XML processing languages such as XSLT, XQuery, XDuce, etc., or to add deep support for XML in traditional programming languages leading to XJ, Comega, and so on. Without special language support, programmers need to fall back on the existing standard programming model and API, the DOM, which is rather complex. Look at the following scenario involving a simple purchase order

```
    <purchaseOrder orderDate="1999-10-20">
      <shipTo country="US">
        <name>Alice Smith</name>
        <street>123 Maple Street</street>
        <city>Mill Valley</city>
        <state>CA</state>
        <zip>90952</zip>
      </shipTo>
      <billTo country="US">
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <city>Old Town</city>
        <state>PA</state>
        <zip>95819</zip>
      </billTo>
      <items>
        <item partNum="872-AA">
        <productName>Lawnmower</productName>
        <quantity>1</quantity>
        <price>148.95</price>
       </item>
       <item partNum="926-AA">
         <productName>Baby Monitor</productName>
         <quantity>1</quantity>
         <price>39.98</price>
         <shipDate>1999-05-21</shipDate>
       </item>
     </items>
   </purchaseOrder>
```

Now, consider the following little plan for a program to manipulate such a bit of XML:

- First, we want to create an XML document for our simple purchase order. The construction should be compositional since, in a more real-world version of this example, a single purchase order will be created from subparts extracted from databases, address books, or other XML documents. If our construction is not compositional, we will be forced to separate the code that creates the subparts from the code that inserts them into the purchase order, broadening opportunities for incompatibility and bugs.
- Next, we want to print the raw XML document to the console, as a primitive form of debugging. Something as trivial as printing an XML document should be trivial indeed, should it not?
- Once we are certain we have created the right XML structure, the next step is to perform a simple calculation: to aggregate the total value of the order by summing the quantity times the price for each item.

- Finally, we want to add a new element `<total>` to the purchase order to contain the total value of the order as we computed in the previous step.

Let us proceed using the DOM to realize our little plan. First, we find that since the DOM is document centric, we must create a new document `Dim PO As New XmlDocument` as a root for all other nodes we will create. Then we find that creating and inserting child nodes requires two imperative steps. This is unfortunate for readability, debuggability, and maintainability, because now there is no helpful relationship between the structure of the code we write and the structure of the XML we need to create. Furthermore, we fill up our code with temporary variables, one for each node, because it might need to contain sub-nodes in code to-be-written or in a future version of the program. All the variables exist at one level in our code, even though the nodes they represent are at multiple sub-levels of the XML.

We *could* restructure our code with lexical blocks to reflect the hierarchy in the XML, but such would result in puffy, pedantic, unidiomatic code, difficult to modify and to explain to colleagues; would not reduce the cognitive burden on us to invent new variable-naming schemes for things that already have names; and would interfere with the imperative style, which forces spaghetti-like *define-and-refer* patterns via the required two-step *create-then-append* programming pattern. To wit, here is our DOM code:

```
Dim PO As New XmlDocument

Dim purchaseOrder As XmlElement = _
       PO.CreateElement("purchaseOrder")
PO.AppendChild(purchaseOrder)

Dim orderDate XmlAttribute = PO.CreateAttribute("orderDate")
orderDate.Value = "1999-10-20"
purchaseOrder.Attributes.Append(orderDate)

Dim shipTo As XmlElement = PO.CreateElement("shipTo")
purchaseOrder.AppendChild(shipTo)

Dim country As XmlAttribute = PO.CreateAttribute("country")
country.Value = "US"
shipTo.Attributes.Append(country)

...
```

Since it is far from obvious that the above fragment actually produces the XML we need, we should like to verify so by printing the XML on the screen. The obvious first attempt is to simply call `Console.WriteLine(PO.ToString())`. To our disappointment, the `ToString()` method on the type of `PO` namey `XmlDocument` has not been overwritten, and the only thing we see on the screen is `System.Xml.XmlDocument`, which we already knew was the case. The next attempt is to use `Console.WriteLine(PO.OuterXml)`, but yet again we are disappointed, this time we do see some XML, but is not formatted so we cannot verify if it actually matches our intended structure. To just print a nicely formatted version of the document, we first have to learn about a whole generic IO system with leaf types `StringWriter` and `XmlWriter`, and write the following six lines of gratuitous code:

```
Dim SW As New StringWriter()
Dim XW As New XmlTextWriter(SW)
XW.Formatting = Formatting.Indented
PO.WriteTo(XW)
XW.Close()
Console.WriteLine(SW.ToString())
```

By now, we have hardly achieved anything, but the accidental complexity is already so high that we are about to give up on using XML altogether, cursing the hype that XML makes dealing with data simple, which no one who has actually written DOM code could claim.

But, giving up on XML would mean losing our job, so we press on. Let's compute the total value of a purchase order. Conceptually, this is simple: we just loop through the items, keeping a running total of the quantity multiplied by the price of each item:

```
Function ComputeTotal(XmlDocument P) As Double
Total = 0.0
For Each Item As XmlElement In PO.GetElementsByTagName("item")
  Dim Price As Double = _
    XmlConvert.ToDouble(Item("price").FirstChild.Value)
  Dim Quantity As Integer = _
    XmlConvert.ToInt32(Item("quantity").FirstChild.Value)
  Total += Quantity * Price
Next
End Function
```

However, also here the accidental complexity of the DOM API is apparent, just to extract the quantity of an item we have to write a complex expression like `XmlConvert.ToInt32(Item("quantity").FirstChild.Value)`. The call to `FirstChild.Value` is needed because the `quantity` element has a text node as its content.

The access methods for children in the DOM is rather non-uniform, with many special cases. For example to access a particular child *attribute*, you use the `GetAttribute` or `GetAttributeNode` methods, but to access a particular child *element* you use the `Item` default property (indexer). There are special properties `FirstChild` and `LastChild` to access certain child elements, but no such helpers for accessing attributes. These are not just different APIs, but different *kinds* of API, reflecting gratuitously different programming patterns for structures that have more similarities than differences: they are each simply a kind of enumerable substructure of an element.

Anyway, given all the above, adding a new node to the document is now a relatively simple task

```
Dim total As XmlElement = PO.CreateElement("total")
total.AppendChild(PO.CreateTextNode(ComputeTotal(PO).ToString()))
PO.AppendChild(total)
```

In the remainder of this paper we will look at a new XML API, designed in the context of the LINQ project [LINQ] at Microsoft. The LINQ Project is a set of standard libraries in the .NET Framework for query, update, and transformation operations applicable to any collection of local or remote objects. LINQ also provides two specialized APIs: DLinq [DLinq] for accessing SQL relational databases, and XLinq [XLinq] a lightweight, rational, and simple API for manipulating XML. Where the DOM API reflect the state of the art of nearly a decade ago, and is aimed at programmers who are deeply familiar with XML. XLinq reflects recent advances in programming technology and is aimed at all XML developers from experts to novice. In addition, C# and Visual Basic have new query syntax, extending LINQ's SQL-like and XQuery-like capabilities to in-memory data. This new syntax is compiled into the underlying LINQ query operators.

2. XLinq Design Principles

As we noted above, the query component of most query languages, including XQuery and SQL, can be factored into (1) general operations on collections, and (2) domain specific operations on the elements of these collections.

Examples of general operations, useful on any collection, are *mapping* functions over collections, *filtering* elements from collections, *grouping* collections into partitions, *sorting* collections, and *reducing* collections to values by aggregating functions over them.

All these examples have in common that they are parameterized by caller-supplied *functions*. The mapping operation takes a caller-supplied function that transforms elements; the filtering operation takes a predicate function that tests elements for exclusion; the grouping operation takes a function of elements that assigns them to their partitions; the sorting operation takes a function of element pairs that defines a total order on them; and the reducing operation takes a function of elements and values that transforms and accumulates.

They are *generic* in the sense that they work on any collection, regardless of source or element type, whether from a relational database, an XML stream, or objects in memory. In fact, we can generalize even further, abstracting the notion of collection to that of monoids or monads [ICDT 2001].

It is easy to understand how ordinary collections like arrays, lists, etc. are examples of monoids, but the real power of this generalization comes from the fact that remotely sourced and diversely typed collections are also monads [HaskellDB]. This allows us to use one mechanism -query comprehensions translated to standard query operators- to manipulate all kinds of data.

The other side of the factoring equation concerns domain-specific operations on the elements of collections, like all monad libraries, such as parsers [Parsec]. Examples of domain-specific operations in the XML domain include (1) accessing XPath-style structure axes children, attributes, descendants, siblings, and (2) constructing and modifying XML elements. To leverage the new query syntax introduced in C# and Visual Basic, the XLinq API supports node-centric, rather than document-centric, XML creation through a composable, expression-oriented style rather than the imperative, sequential, statement-oriented style of the DOM.

2.1. Functional Construction

As illustrated in the example above, XML construction in the DOM is document-centric, that is, elements and attributes are always created in the context of a particular container document. It is also imperative, that is, XML fragments are created by a sequence of calls to factory methods and update statements to insert the newly constructed nodes into the tree. As with most imperative approaches, a proliferation of temporary variables is necessary to bridge from one statement to the next and to preserve the options to modify intermediate results

The imperative, sequential style of construction does not directly resemble the recursive, tree structure of the generated XML: there is a lack of visible relationship between the code and the data it produces. Even without this deficiency, however, imperative update statements as a programming style is a non-starter for Visual Basic and C# query comprehensions and LINQ query operators since they require XML to be created using expressions, hence using imperative updates is a non-starter.

The XLinq API instead provides several convenient direct constructors for creating elements, attributes and documents. To construct an element such as

```
<item partNum="926-AA">
  <productName>Baby Monitor</productName>
  <quantity>1</quantity>
  <price>39.98</price>
  <shipDate>1999-05-21</shipDate>
</item>
```

you just create an *expression* composed from nested `XElement` and `XAttribute` constructors:

```
New XElement("item", _
  New XAttribute("partNum", "926-AA"), _
  New XElement("productName", "Baby Monitor"), _
  New XElement("quantity", 1), _
  New XElement("price", 39.98), _
  New XElement("shipDate", "1999-05-21"))))
```

2.2. Context Free

The document-centric approach makes the DOM API non-compositional, since elements and attributes are not first-class, standalone values. For example, it is hard to write a function that returns an XML fragment since that fragment cannot exist by itself, even in a temporary variable, but must live inside a global container document. To insert the result of that function inside another document, we first must import it into the destination container using the `Function ImportNode(Node As XmlNode, Deep As Boolean) As XmlNode` method. Similarly, we cannot easily define a function that takes an `XmlNode` as an argument, since, again, to use that argument inside the function body, we must clone it into the ambient document via `ImportNode`.

The XLinq API makes working with XML elements and attributes much easier by constructing them independently of a container document. As a result, nodes are truly first-class values that can be freely passed to and returned from functions. The object model, of course, does provide the notion of a `XmlDocument` when needed, but does not require it in any ways, shape or form to construct individual elements or attributes.

Both elements and attributes may be parented. Already parented items are automatically cloned when passed as arguments to the constructor of another element or document.

```
REM BillTo is unparented
Dim BillTo = New XElement("billTo", ...)
REM BillTo becomes parented
Dim PO = New XElement("purchaseOrder", ... BillTo ...)
REM BillTo get's cloned
Dim AnotherPO = New XElement("purchaseOrder", ... BillTo ...)
```

2.3. Simplified treatment of names

XML namespaces and namespace prefixes are perceived by many as one of the most confusing aspects of XML. For example, the DOM has three overloads for the factory method that create elements:

```
Function CreateElement _
  (prefix As String, localName As String, namespaceURI As String) _
     As XmlElement
```

where according to the documentation, `prefix` is the prefix of the new element (if any, both the empty string and null indicate no prefix), `localName` is the local name of the new element, and `namespaceURI` is the namespace URI of the new element (if any, both the empty string and null indicate no namespace URI)

```
Function CreateElement _
```

```
    (qualifiedName As String, namespaceURI As String) _
       As XmlElement
```

and

```
Function CreateElement(name As String) As XmlElement
```

where according to the documentation is the `qualifiedName` contains a colon, then the part of the name preceding the colon will be used as the `prefix` and the part of the name after the colon will be used as the `localName`.

In contradistinction to the DOM, the XLinq API does not expose prefixes, local names, and namespaces, as separate arguments or components of seemingly equal status. Instead, it presents the notion of `XName` that represents a fully expanded XML name of the form `"{namespace}localname"`, consisting of an XML namespace concatenated to a local name. Prefixes are just a serialization option when reading and writing XML documents from the outside.

```
REM define prefix as normal string
Dim a = "{http://ecommerce.org/schema}"
REM concatenate prefix and localname
Dim BillTo = New XElement(a & "billTo", ...)
```

It is possible to explicitly control prefix declarations in the constructed XML by creating an attribute on a node that uses the `http://www.w3.org/2000/xmlns/` namespace to declare a prefix. For example, to declare the prefix `a` for the namespace `http://ecommerce.org/schema`, we would use:

```
New XAttribute("{http://www.w3.org/2000/xmlns/}a", _
               "http://ecommerce.org/schema")
```

or to define `http://ecommerce.org/schema` as the default namespace we would use:

```
New XAttribute("{http://www.w3.org/2000/xmlns/}xmlns", _
               "http://ecommerce.org/schema")
```

As we will see below, XML literals in Visual Basic make construction of XML and dealing with namespaces even easier. In fact, one of the design goals is to allow cut and paste of existing XML into Visual Basic programs.

2.4. No Text Nodes

The XLinq object model is much simpler than the DOM, in particular Xlinq does not expose entity references (`Xm-lEntityReference`), text nodes (`XmlText`), significant whitespace nodes (`XmlSignificantWhiteSpace`), and whitespace nodes (`XmlWhiteSpace`), and document fragments (`XmlDocumentFragment`) as separate types in the object model.

To address the pervasive pattern of accessing leaf-node values of an element, XLing provides explicit conversions from `XElement` and `XAttribute` to several base types such as `String`, `Integer`, `Double`, etc. So to access the `price` content of an `<item>` element, we just write `CDbl(Item.Element("price"))` or `CType(Item.Element("price"), Double)`.

As we will see below, atomization and compiler inserted conversions in Visual Basic make converting (collections of) elements and attributes into their underlying values even easier, and instead of the above we can write `Item.price`.

2.5. Collection Friendly

All XLinq axis, such as `Function Elements(name As XName) As IEnumerable(Of XElement)`, come in two flavours, one that is defined on singletons and one that is lifted over collections of node by using the new *extension methods* feature. This enables very natural path-style expressions that uses method-chaining where each next axis just dots through the result of a previous axis.

For example to access all `partNum` attributes in an order we write `PO.Descendants("item").Attribute("partNum")`. Note that the first part `PO.Descendants("item")` of this expression returns a collection of type `IEnumerable(Of XElement)`. The compiler will automatically rewrite the expression to `Attribute("partNum", PO.Descendants("item"))` to use the extension method defined on collections of elements.

As we will see below, axes members in Visual Basic navigating XML even easier, and instead we can write `PO...item.@partNum`.

2.6. The Example Using XLinq

The power of functional construction and composition is patently obvious in the following fragment. Not only is the proliferation of temporary variables reduced, but the nesting structure of the desired XML is visually apparent in the code that builds it. We initialize only one temporary, the variable `BillTo` with the billing address of the order, and subsequently use that to construct the complete order and assign it to the `PO` variable:

```
Dim BillTo = New XElement("billTo", _
   New XAttribute("country", "US"), _
   New XElement("name", "Robert Smith"), _
   New XElement("street", "8 Oak Avenue"), _
   New XElement("city", "Old Town"), _
   New XElement("state", "PA"), _
   New XElement("zip", 95819))

Dim PO = New XElement("purchaseOrder", _
    New XAttribute("orderDate", "1999-10-20"), _
    New XElement("shipTo", New XAttribute("country", "US"), _
     New XElement("name", "Alice Smith"), _
     New XElement("street", "123 Maple Street"), _
     New XElement("city", "Mill Valley"), _
     New XElement("state", "CA"), _
     New XElement("zip", 90952)), _
     BillTo, _
   New XElement("items", _
     New XElement("item", _
 New XAttribute("partNum", "872-AA"), _
 New XElement("productName", "Lawnmower"), _
 New XElement("quantity", 1), _
 New XElement("price", 148.95)), _
     New XElement("item", _
 New XAttribute("partNum", "926-AA"), _
 New XElement("productName", "Baby Monitor"), _
 New XElement("quantity", 1), _
 New XElement("price", 39.98), _
 New XElement("shipDate", "1999-05-21"))))
```

A small, but extremely convenient detail is that the `XElement` class properly overwrites the `ToString` method. Recall how we had to learn about string writers and xml writers and details of opening and closing IO channels just to dump our XML DOM to the screen for verification, all because our first, educated guess, `ToString`, surprised us. With XLinq, though, what ought to be trivial actually is trivial: print out the document to the console just as follows `Console.WriteLine(PO)`.

The XLinq axes, in combination with the fact that the Visual Basic compiler is happy to inserts downcasts on behalf of the programmer, makes the computation of the `ComputeTotal` function a breeze. To access the descendants of a node just use the `Descendants` accessor, to access the first child element of a node, we use the `Element` accessor:

```
Function ComputeTotal(ByVal PO As XElement) As Double
  For Each Item As XElement In PO.Descendants("item")
    Dim Price As Double = Item.Element("price")
    Dim Quantity As Integer = Item.Element("quantity")
    Total += Quantity * Price
  Next
End Function
```

This is much shorter and more uniformly constructed than the prior version. All the axes are accessed the same way, rather than with one way (indexers) for some and another way (`IEnumerable`) for others.

Finally, adding a new child element that contains the computed total is also simple, since XLinq supports imperative update operations as well as functional construction. In this case we attach the total price as an attribute to the added node:

```
PO.Add(New XElement("Total", New XAttribute("Price", Total(PO))))
```

3. XLinq In More Depth

Now that we have seen a first comparison of the traditional DOM and the new XLinq API, let's look at it in some more detail.

3.1. The Object Model

The XLinq object model contains a handful of types. The abstract class `XNode` (written as `MustInherit` in Visual Basic) is the base for element nodes, and provides `Parent` and methods such as `AddBeforeThis`, `AddAfterThis` and `Remove` for updates in the imperative style. For IO, it provides methods for reading `ReadFrom` and writing `WriteTo`.

The ordinary cases, `XElement` and `XAttribute`, inherit from `XContainer`, immediately below, but the more rare cases `XProcessingInstruction`, `XDeclaration`, `XComment`, and `XDocumentType` all inherit from `XNode` and represent their obvious XML counterparts.

```
MustInherit Class XNode

Class XProcessingInstruction: Inherits XNode
Class XDeclaration: Inherits XNode
Class XComment: Inherits XNode
Class XDocumentType: Inherits XNode
```

The abstract class `XContainer` is the base for element nodes that have children. It adds axis methods such as `Content`, `Descendants`, and `Element` and `Elements`, as well as imperative update methods such as `Add`, `AddFirst`, `RemoveContent`, and `ReplaceContent`.

The `XElement` class represents proper XML elements and adds further axes such as `Ancestors`, `SelfAndAncestors`, `SelfAndDescendants` and `Attributes`, and explicit conversions for accessing the content of element nodes.

The `XDocument` class represents complete XML documents, but, because of the hegemony of DOM, we must emphasize again that `XElements` are created and manipulated independently of `XDocuments`.

```
MustInherit Class XContainer: Inherits XNode
Class XElement: Inherits XContainer
Class XDocument: Inherits XContainer
```

The `XCharacterNode` base class for `CData` and for the internal class `XText`. The `CData` class represent CData sections.

```
MustInherit Class XCharacterNode: Inherits XNode
Class XCData: Inherits XCharacterNode
```

The `XAttribute` class represents attributes and is stand-alone; it does not derive from `XNode`. It provides the `Parent` axis, as well as imperative updates and explicit conversions to get the value of attributes.

The `XName` class represents fully expanded XML names and provides accessors such as `ExpandedName`, `LocalName`, `NamespaceName`, conversions to and from strings, and the factory method `Get`. Internally, `XName` maintains a generational name table that allows for efficient comparison of names. The name table can be reset using the `ClearNameCache` method.

4. Monads

As mentioned earlier, LINQ leverages the mathematical foundation of monads to separate the general purpose query framework from the various domain specific operations, such as in this case XML axes. Monads are a recognized formalism for generalized mapping, filtering, partitioning, sorting, and folding over collections of arbitrary source and type.

The `System.Query` namespace contains the standard monadic query operators defined on both in-memory collections `IEnumerable(Of T)` and on remote collections `Query(Of T)`. In monad speak, the former correspond to the standard list monad, while the latter corresponds to the query monad as found in HaskellDB [HaskellDB]. Both Visual Basic and C# define their own language specific query comprehension syntax on top of these base operators.

4.1. Standard Query Operators

The CLR type-system is not expressive enough to define the notion of a monad directly in the type-system as is possible in Haskell. Instead we define what is called a standard query *pattern*. In many cases the operations are defined as *extension methods* which are specially decorated static methods that can be accessed using standard instance functions invocation syntax.

The two main query operators captured by the standard query operators pattern are the `Select` and `SelectMany` *functors*. Both are defined as extension methods. The `Select` function applies a function from `S` to `T` represented by a delegate of type `Func(Of T, S)` to each element in the source collection of type `IEnumerable(Of T)` to create a collection of type `IEnumerable(Of S)`. Functional programmers will recognize this as the standard `map` function:

```
 <Extension>
 Shared Function Select(Of T, S) _
    (source As IEnumerable(Of T), selector As Func(Of T, S)) _
      As IEnumerable(Of S)
```

The version of `Select` on remote collections takes as its second argument an *expression* representing a function from `T` to `S`, i.e. this version of `Select` takes a remote collection, and an *intensional* representation of a function as it's arguments:

```
<Extension>
Shared Function [Select](Of T, S) _
  (source As Query(Of T), _
     selector As Expression(Of Func(Of T, S))) _
As Query(Of S)
```

Notice the `Expression(Of Func(Of T, S))` term as the type of the selector. It implies a delegate-to-expression-tree translator built into the implementation of `Query`.

The `SelectMany` function creates a new target collection of type `IEnumerable(Of S)` from a *collection generating* function from `T` to `IEnumerable(Of S)` by applying this function to each elements in the source collection. Again, there is a second overload of this function that takes a remote collection and an expression tree as its arguments. Functional programmers will recognize this as the standard `concatMap` function, or indeed as monadic bind:

```
<Extension>
  Shared Function SelectMany(Of T, S) _
   (source As IEnumerable(Of T), selector As Func(Of T, S)) _
     As IEnumerable(Of S)

<Extension>
  Shared Function SelectMany(Of T, S) _
   (source As Query(Of T), _
selector As Expression(Of Func(Of T, IEnumerable(Of S)))) _
  As Query(Of S)
```

Other functions in the core sequence operators pattern are filters `Shared Function Where(Of T)(source As IEnumerable(Of T), predicate As Func(Of T, Boolean)) As IEnumerable(Of T)` on in-memory collections and `Shared Function Where(Of T)(source As Query(Of T), predicte As Expression(Of Func(Of T, Boolean))) As Query(Of T)` or remote collections, grouping `Shared Function GroupBy(Of T, K)(source As IEnumerable(Of T), keySelector As Func(Of T, K)) As IEnumerable(Of Grouping(Of K, T))`, and sorting.

Based on this set of base operators, programming languages can define syntactic sugar in the form of query comprehensions. The main advantage of using query comprehensions as opposed to using the underlying operators is that query comprehensions eliminate the use of delegates and make binding much easier to use.

4.2. Query Comprehensions in C# and Visual Basic

The C# syntax [C# 3.0] for query comprehensions is similar to XQuery FLWR expressions, and is of the form

```
from b in BN.Descendants("book"), a in AZ.Descendants("book")
where b.Attribute("title") == a.Attribute("title")
orderby a.Element("price") descending
select new XElement("book", a.Element("price"), b.Attribute("title"))
```

The Visual Basic [VB9] syntax for query comprehensions is similar to the SQL or OQL select statement, and is of the form

```
Select New XElement("book", a.Element("price"), b.Attribute("title")) _
From b In BN.Descendants("book"), a In AZ.Descendants("book") _
Where b.Attribute("title") == a.Attribute("title") _
OrderBy a.Element("price") Desc
```

The translation of both queries into the underlying base operators essentially looks something like the following expression where we use `Function(...,X,..)E` as the syntax for anonymous delegates:

```
BN.Descendants("book").SelectMany( _
  Function(b) AZ.Descendants("book").Select(Function(a) New{ a, b }).
    Where(Function(It) It.b.Attribute("title") == It.a.Attribute("title")).

Orderby(Function(It)It.a.Element("Price")).
  Select(Function(It) New XElement("book", It.a.Element("price"), _
           It.b.Attribute("title"))
```

5. XLinq In Visual Basic

With respect to XML support, Visual Basic goes one step further and adds *XML literals* and *axis members* to the language.

5.1. XML Literals

With XML literals, we can directly embed XML notation inside Visual Basic. Inside XML literals we can leave *holes* for attributes, attribute names, or attribute values, for element names by using `(expression)`, or for child elements using the ASP.Net style syntax `<%= expression %>`, or `<% statement %>` for blocks. The following is our original example, showing now, near maximal fidelity between the XML data and the code that produces it:

```
Dim BillTo = <billTo country="US">
   <name>Robert Smith</name>
   <street>8 Oak Avenue</street>
   <city>Old Town</city>
   <state>PA</state>
   <zip>95819</zip>
       </billTo>

Dim PO = <purchaseOrder orderDate=(System.DateTime.Today)>
    <shipTo country="US">
      <name>Alice Smith</name>
      <street>123 Maple Street</street>
      <city>Mill Valley</city>
      <state>CA</state>
      <zip>90952</zip>
    </shipTo>
    <%= BillTo %>
    <items>
    <%=
      Select <item partNum=(O.PartID)>
         <productName>
     <%= O.Product %>
         </productName>
         <quantity>
     <%= O.Quantity %>
         </quantity>
         <price>
     <%= O.Price %>
    </price>
        </item>
      From O In Orders
      Where O.Name = "Robert Smith"
    %>
    </items>
        </purchaseOrder>
```

The Visual Basic compiler takes XML literals and translates them into constructor calls of to the underlying XLinq API. As a result, XML produced by Visual Basic can be freely passed to any other component that accepts XLinq values, and similarly, Visual Basic code can accept XLinq XML produced by external components.

We also allow XML fragments enclosed between empty angle brackets such as `<><city>123 Maple Street</city><state>CA</stat></>`, and complete XML documents that start with the XML declaration like `<?xml version="1.0" encoding="UTF-8"?>`.

One thing that Visual Basic's XML literals make particularly easy is handling of namespaces. We support normal namespace declarations, default namespace declarations, and no namespace declarations, as well as qualified names for elements and attributes. The compiler generates the correct XLinq calls to ensure that prefixes are preserved when the XML is serialized.

```
Dim BillTo = <a:billTo
  xmlns:a="http://ecommerce.org/schema" country="US">
 <a:name>Robert Smith</a:name>
 <a:street>8 Oak Avenue</a:street>
 <a:city>Old Town</a:city>
 <a:state>PA</a:state>
 <a:zip>95819</a:zip>
      </a:billTo>
```

Finally, notice that Basic line-continuation characters, consisting of white space and underscores, is neither required nor permitted inside XML literals.

5.2. Axis members

Whereas XML literals make constructing XML easy in Visual Basic, the concept of *axis members* makes accessing XML easy. The essence of the idea is to delay the binding of identifiers to actual XML attributes and elements until run time. When the compiler cannot find a binding for a variable, it emits code to call a helper function at run time. This tactic will be familiar to many under the rubric "late binding", and, indeed, it is a form of ordinary Visual Basic late binding. But it has the advantage that the names of element tags and attributes can be used directly in Visual Basic code without quoting. As such, it relieves the programmer of the significant cognitive burden of switching between object space and XML-data space. The programmer can treat the spaces the same: as hierarchies accessed through "dot".

For example, we can use the *child axis* member `BillTo.street` to get all `street` child elements from the `BillTo` XML element, or we can use the *attribute axis* member `BillTo.@country` to get the `country` attribute of the `BillTo` element. The third axis we support directly is the *descendants axis*, written literally as three dots in the source code, `PO...item` to get all `item` children of the `PO` document, no matter how deeply in the hierarchy they occur.

Axis member access works on both singletons as well as on collections, since in general there can zero, one, or several elements that are returned, or the accessor is applied to a collection to start with. This makes axis members compositional in the sense that we can chain dotting through the results of previous axis as in `PO...item.@partNum` to obtain the collection of all `partNum` attributes in the `PO` document.

Both the child and descendant axes return collections of elements. Using the extension indexer on `IEnumerable(Of T)` we can select any desired element of the resulting sequence, for example we can get the second item in the order by writing `PO...item(1)`. Note that in most programming languages indexing starts at 0, not at 1 as in XQuery.

We also allow atomizing conversions from `IEnumerable(Of XElement)` and `IEnumerable(Of XAttribute)` by lifting the respective conversions on `XElement` and `XAttribute` already supplied by XLinq.

5.3. The Example Revisited

We have already seen how we can create our input document using XML literals with 100% fidelity to the final XML. The combination of XML literals, axes members, the XLinq API and query comprehensions allow us to write an extremely concise version of computing the total value of the order and adding it to the orginal document. We consider this single line of code the symbolic victory of our approach:

```
Dim POWithTotal = PO.Add( _
    <Total Price=(Select Sum(quantity*price) From PO...item)/>)
```

6. Conclusions

By switching from a document-centric model to a node-centric model, and by aggressively demanding functional composability as a design principle instead of an imperative, step-wise style, we achieve great economy of expression in code that constructs XML. Not only do we reduce the proliferation of temporary variables to hold results from one step to the next, but the code that creates an XML tree visually resembles the tree structure of the XML tree itself. The visual resemblance reaches near maximal fidelity through XML literals, retaining the flexibility to substitute expression values from the ambient code through syntactic holes in the XML. The approach here will be familiar to some readers under the names of `quasiquote` and `unquote`.

By noticing that XML programs can be factored into generic operations and domain-specific operations; and by noticing that the generic query operations are virtually the same for other sources and kinds of data, like relational [DataModel] and objects in-memory and that they follow a well understood *monadic* discipline, we are able to provide a single API that compositionally performs mapping, filtering, grouping, sorting, and aggregating values in the same way for all kinds of data. This is LINQ and XLinq.

Finally, with axis members and late binding in Visual Basic, we are able to achieve full compositionality over retrieval of content and attributes from XML into the ambient program.

The repeated theme is compositionality. By making all our facilities fully compositional, we are able finally to achieve what we believe to be near maximal notational economy for programs that create, read, and write XML

Acknowledgements

Many thanks to the members of the XLinq design team Don Box, Arpan Desai, Anders Hejlsberg, Asad Jawahar, Andrew Kimball, Dave Remy, Michael Rys, Erik Saltwell, and Ion Vasilian, and the Tesla and Visual Basic design teams Avner Aharoni, David Schach, Peter Drayton, Ralf Lammel, Amanda Silver, and Paul Vick for all their hard work to make XLinq and the Visual Basic 9 language a reality.

Bibliography

[ComprehendingMonads] *Comprehending Monads*. Mathematical Structures in Computer Science, Special issue of selected papers from 6'th Conference on Lisp and Functional Programming, 2:461-493, 1992. Available at http://homepages.inf.ed.ac.uk/wadler/papers/monads/monads.ps.

[Comprehensions] *Monad Comprehensions: A Versatile Representation for Queries*. Torsten Grust. In Gray, P.M.D., Kerschberg, L., King, P.J.H., and Poulovassilis, A., editors, The Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data, pages 288--311, Springer Verlag, September 2003. Available at http://www.inf.uni-konstanz.de/dbis/publications/download/monad-comprehensions.pdf.

[ICDT 2001] *A Semistructured Monad for Semistructured Data*. Mary Fernandez, Jerome Simeon, Philip Wadler. Proceedings ICDT 2001. Available at http://homepages.inf.ed.ac.uk/wadler/papers/xalgebra-icdt/xalgebra-icdt.pdf.

[HaskellDB] *Domain Specific Embedded Compilers*. Daan Leijen and Erik Meijer. 2nd USENIX Conference on Domain-Specific Languages (DSL), Austin, USA, October 1999. Available at http://www.cs.uu.nl/people/daan/papers/dsec.ps.

[Parsec] *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Daan Leijen and Erik Meijer. Technical Report UU-CS-2001-35, Departement of Computer Science, Universiteit Utrecht, 2001. Available at http://www.cs.uu.nl/~daan/pubs.html#parsec.

[XQuery] *XQuery 1.0: An XML Query Language*. Available at http://www.w3.org/TR/xquery/.

[XJ] *XML Enhancements for Java*. Available at http://www.alphaworks.ibm.com/tech/xj.

[Xtatic] *The Xtatic Project: Native XML processing for C#*. Available at http://www.cis.upenn.edu/~bcpierce/xtatic/.

[Haskell] *Haskell 98 Language and Libraries The Revised Report*. Available at http://www.haskell.org/onlinereport/.

[Scala] *The Scala Programming Language*. Available at http://scala.epfl.ch/.

[Python] *Python Reference Manual*. Available at http://docs.python.org/ref/ref.html.

[Comega] *The essence of data access in Comega*. Available at http://research.microsoft.com/Comega/.

[LINQ] *LINQ Project*. Available at http://msdn.microsoft.com/netframework/future/linq/.

[DLinq] *DLINQ Overview*. Available at http://download.microsoft.com/download/c/f/b/cfbbc093-f3b3-4fdb-a170-604db2e29e99/DLinq%20Overview.doc.

[XLinq] *XLINQ Overview*. Available at http://msdn.microsoft.com/VBasic/Future/XLinq%20Overview.doc.

[C# 3.0] *C# 3.0 Language Specification*. Available at http://download.microsoft.com/download/9/5/0/9503e33e-fde6-4aed-b5d0-ffe749822f1b/csharp%203.0%20specification.doc.

[VB9] *Overview of Visual Basic 9.0* . Available at http://msdn.microsoft.com/vbasic/future/default.aspx?pull=/library/en-us/dnvs05/html/vb9overview.asp.

[W3C DOM] *Document Object Model (DOM) Level 3 Core Specification*. Available at http://www.w3.org/TR/DOM-Level-3-Core/.

[DataModel] *XQuery 1.0 and XPath 2.0 Data Model*. Available at http://www.w3.org/TR/xpath-datamodel/.

[W3C DOM] *Document Object Model (DOM) Level 3 Core Specification*. Available at http://www.w3.org/TR/DOM-Level-3-Core/.

[DataModel] *A Formal Data Model and Algebra for XML*Ashok Malhotra. Available at http://www-db.stanford.edu/db-seminar/Archive/FallY99/.

Biography

Erik **Meijer**
Architect
Microsoft Corporation [http://www.microsoft.com]
Redmond
Washington
United States of America
http://www.research.microsoft.com/~emeijer/ [http://www.research.microsoft.com/~emeijer]

Erik Meijer is an architect in the WebData XML group at Microsoft where he works with the C# and Visual Basic teams on language and type-systems for data integration in programming languages. Prior to joining Microsoft he was an associate professor at Utrecht University and adjunct professor at the Oregon Graduate Institute. Erik is one of the designers of the Mondrian scripting language, standard functional programming language Haskell98, and Comega.

Brian **Beckman**
Architect
Microsoft Corporation [http://www.microsoft.com]
Redmond
Washington
United States of America

Brian is an architect in SQL Server at Microsoft where amongst many other things, he works with the WinFS team on the design of the data models and type systems.