F# 3.0 Query, Data & Service Programming

Contents

[1 Introduction 3](#_Toc300669430)

[1.1 Motivation & Scenarios 3](#_Toc300669431)

[1.2 Philosophy and Desired User Experience 4](#_Toc300669432)

[1.3 Problems F# 2.0 Query Support 4](#_Toc300669433)

[2 Overview of the F# 3.0 Design for Data/Services 4](#_Toc300669434)

[2.1 Queries - Syntax Examples 5](#_Toc300669435)

[2.2 Standard Query Operators 8](#_Toc300669436)

[2.3 Nullable 9](#_Toc300669437)

[2.4 Terminology: query v. seq v. combinators 11](#_Toc300669438)

[3 The query { … } and QueryBuilder types 13](#_Toc300669439)

[3.1 The query API 13](#_Toc300669440)

[3.2 How query { … } executes 13](#_Toc300669441)

[3.3 The QueryBuilder type 13](#_Toc300669442)

[3.4 From F# Queries to LINQ Operator Calls 14](#_Toc300669443)

[3.5 The Tuple/Record <-> Mutable Tuple translation 14](#_Toc300669444)

[4 Standard Data/Service/Resource Type Providers 15](#_Toc300669445)

[4.1 SqlDataConnection<…> 15](#_Toc300669446)

[4.2 DbmlFile<…> 19](#_Toc300669447)

[4.3 SqlEntityConnection<…> 20](#_Toc300669448)

[4.4 EdmxFile<…> 24](#_Toc300669449)

[4.5 ODataService<…> 26](#_Toc300669450)

[4.6 WsdlService<…> 29](#_Toc300669451)

[4.7 ResxFile<…> 32](#_Toc300669452)

[5 Language Support for Queries 32](#_Toc300669453)

[5.1 Lexing 32](#_Toc300669454)

[5.2 Parsing 32](#_Toc300669455)

[5.3 Language Support For Calling LINQ Methods 33](#_Toc300669456)

[5.4 Computation Expression Custom Operations 33](#_Toc300669457)

[5.5 Attributes for Specifying Computation Expression Custom Operations 35](#_Toc300669458)

[5.6 Example: Specifying Your Own Query Builder 36](#_Toc300669459)

[6 Leaf Expression Translation to LINQ 37](#_Toc300669460)

[6.1 The Translation API 37](#_Toc300669461)

[6.2 The Translation 37](#_Toc300669462)

[6.3 Helper functions that indicate specific translations when used in input quotations 39](#_Toc300669463)

[6.4 Helper function for nested quotation literals 40](#_Toc300669464)

[7 Performance for Queries 40](#_Toc300669465)

[7.1 Client-side CPU costs of executing queries 40](#_Toc300669466)

[7.2 Size of FSharp.Core 41](#_Toc300669467)

[8 Syntax Highlighting, IntelliSense, Diagnostics and Debugging for Queries 41](#_Toc300669468)

[8.1 Syntax Highlighting 41](#_Toc300669469)

[8.2 IntelliSense 41](#_Toc300669470)

[8.3 QuickInfo 41](#_Toc300669471)

[8.4 Diagnostics 42](#_Toc300669472)

[8.5 Debugging 42](#_Toc300669473)

[9 Additions to Standard F# Project and Item Templates 42](#_Toc300669474)

[9.1 RESX Item Template 43](#_Toc300669475)

[9.2 DBML Item Template 43](#_Toc300669476)

[9.3 EDMX Item Template 43](#_Toc300669477)

[9.4 F# Data Consumption Project (SQL, Entity Framework or ADO.NET) 43](#_Toc300669478)

[9.5 F# Service Consumption Project (OData or WSDL) 43](#_Toc300669479)

[10 Out of Band Work 43](#_Toc300669480)

[10.1 Query Keyword Highlighting 43](#_Toc300669481)

[10.2 F# Power Pack gets EasyChart library 43](#_Toc300669482)

[10.3 Ship Additional Providers in F# Power Pack 44](#_Toc300669483)

[10.4 Other Major Uses of LINQ – DryadLINQ 45](#_Toc300669484)

[11 Risks 46](#_Toc300669485)

[12 Resolved Issues 46](#_Toc300669486)

[12.2 No Nullable arithmetic operators 47](#_Toc300669487)

[12.3 No Nullable default conversions 47](#_Toc300669488)

[12.4 No Nullable user-defined conversions 47](#_Toc300669489)

[12.5 No syntax for array/list 47](#_Toc300669490)

[12.6 No Seq operators 47](#_Toc300669491)

[12.7 No Concat operator 47](#_Toc300669492)

[12.8 No query operators for array/list 47](#_Toc300669493)

[12.9 No Direct Emit to LINQ Expression Trees 47](#_Toc300669494)

[12.10 No Direct Translation to LINQ Query Pattern 47](#_Toc300669495)

[13 Appendix 48](#_Toc300669496)

[13.1 F# 2.0 LINQ support 48](#_Toc300669497)

[13.2 Problems F# 2.0 Query Support 48](#_Toc300669498)

[13.3 Additional LINQ implementations to investigate 50](#_Toc300669499)

[13.4 Splicing Into query { … } 51](#_Toc300669500)

[13.5 Added Random Notes and Jottings 51](#_Toc300669501)

# Introduction

## Motivation & Scenarios

F# 3.0 will include Type Providers, bringing many external queryable data sources to F# developers, including

* Online schematized data stores such as Freebase
* SQL databases, LinqToSQL
* SQL databases, LinqToEntites
* Web-based sources of data such as OData and Windows Azure DataMarket feeds.
* Sources of streaming, event-based data such as StreamInsight

The fundamental scenarios involve writing queries over these data sources where the query is shipped to the source of the data and executed there.

The C#, VB and .NET standard way of querying both in-memory and external data sources is the LINQ query pattern. There are two kinds of LINQ providers:

* **IQueryable** providers. This is used by LinqToEntities, LinqtoSql, OData and others.
* **LINQ pattern** providers. This is used by StreamInsight and the other more adhoc uses of LINQ listed at the end of this document.

Practically speaking, LINQ also includes:

* A complete, self-contained mindset for querying and transforming data. The C# 2.0 LINQ user steps in and out of LINQ syntax, using LINQ to do “data programming in the small”.
* A huge reservoir of “standard queries” and examples, e.g. a typical user action is to search for “LINQ query sum” and copy a C# example.
* Many examples of practically working with databases, e.g. w.r.t. transactions.

## Philosophy and Desired User Experience

* F# 3.0 **supports LINQ queries** (”[[1]](#footnote-1)).
* With F# 3.0, **SQL[[2]](#footnote-2) and OData data is easy to access and query**, including real-world cases.
* F# 3.0 data/service/query programming is **code-focused**. You don’t leave your script.
* Writing queries with F# 3.0 is simple, intuitive and easy.
* With F# 3.0, you can access data and services **without seeing any generated code**.
* With F# 3.0, you can write your own type provider and implement IQueryable to give query access to a new kind of data or service.

## Problems F# 2.0 Query Support

The F# 2.0 PowerPack includes reasonable support for IQueryable LINQ queries. The support is vaguely palatable, but has many problems. See the appendix.

# Overview of the F# 3.0 Design for Data/Services

The design elements for F# 3.0 design for data and services are:

* **Language additions:**
  + The Type Providers feature
  + Computation expression custom operations
* **Compiler additions:**
  + query { … } gets compiled as state machines when primary input is not IQueryable
* **FSharp.Core additions:**
  + A query { … } computation builder for IEnumerable and IQueryable queries
    - This includes a set of custom query operators like minBy, maxBy, groupBy etc.
    - This includes matching extensions to seq { … } computation builder and Seq.\*
  + A small, simple set of operators & rules associated with Nullable
* **A new DLL, FSharp.Data.TypeProviders.dll:**
  + A set of standard type providers:
    - ODataService<…>
    - SqlDataConnection<…>
    - DbmlFile<…>
    - EdmxFile<…>
    - ResxFile<…>
    - WsdlService<…>.
* **Tooling:**
  + Intellisense
  + Debugging
  + Syntax highlighting for query operators

## Queries - Syntax Examples

Here are examples of what queries look like:

query { for c in db.Customers do

yield c.ContactName }

“select” can also be used for “yield”:

query { for c in db.Customers do

select c.ContactName }

“Where”, “Select”

query { for c in db.Customers do

for e in db.Employees do

where c.ContactName = e.LastName

select c.ContactName }

“Sum by”

query { for c in db.Customers do

sumBy c.Name.Length }

“Min by”

query { for c in db.Customers do

minBy c.Name.Length }

“Max by”

query { for c in db.Customers do

maxBy c.Name.Length }

“Average by”

query { for c in db.Customers do

averageBy c.Name.Length }

“Order by”

query { for p in products do

for e in employees do

sortBy (p.Name + "," + e.Name)

yield p.Salary }

“Order by, Then by”

query { for student in students do

sortBy student.LastName

thenBy student.FirstName

select student.Age }

“Order by descending”

query { for student in students do

sortByDescending student.LastName

thenByDescending student.FirstName

select student }

“Group by” (no into)

let productsGroupedByName =

query { for p in db.Products do

groupBy p.ProductName }

“Group by” (into)

let productsGroupedByName =

query { for p in db.Products do

groupBy p.ProductName into g

select (g.Key, g) }

“Group selected values by” (into)

let productsGroupedByName =

query { for p in db.Products do

groupValBy p.ProductName p.ProductCategory; into g

select (g.Key, g) }

“Inner Join” (no into)

let joinCustomersAndEmployeesByName =

query { for c in db.Customers do

join (for e in db.Employees -> c.Country = e.Country)

select (c.ContactName, e.LastName) }

“Inner Join” (nullable key on right)

let innerJoinQuery =

query { for c in db.Categories do

join (for p in db.Products -> c.CategoryID =? p.CategoryID)

select (p.ProductName, c.CategoryName) }

“Inner Group Join”

let innerGroupJoinQuery =

query { for c in db.Customers do

groupJoin (for e in db.Employees -> c.Country = e.Country) into employeeGroup

select (c.ContactName, employeeGroup.Count) };

“Inner Group Join with Aggregation”

let innerGroupJoinQueryWithAggregation =

query { for c in db.Categories do

groupJoin (for p in db.Products -> c.CategoryID =? p.CategoryID) into prodGroup

let groupMax = query { for p in prodGroup do maxByNullable p.UnitsOnOrder }

select (c.CategoryName, groupMax) }

“Left outer join”

let leftOuterJoinQuery =

query { for c in db.Categories do

leftOuterJoin[[3]](#footnote-3) (for p in db.Products -> c.CategoryID =? p.CategoryID) into prodGroup

for item in prodGroup do

select (c.CategoryName, (match item with null -> "" | \_ -> item.ProductName)) }

“Composite Key Query”

// The following example demonstrates how to use a composite key to join data from three tables:

let compositeKeyQuery =

query { for o in db.Orders do

for p in db.Products do

groupJoin (for d in db.OrderDetails ->

(o.OrderID, p.ProductID) = (d.OrderID, d.ProductID)) into details

for d in details do

select (o.OrderID, p.ProductID, d.UnitPrice) }

“Exists/Any”

query { for c in db.Customers do

exists (c.Address.Length > 10) }

“Forall/All”

query { for c in db.Customers do

all (c.Address.Length < 10) }

“Nested queries”

let queryWithNestedQueryInFinalSelect =

query { for c in db.Customers do

select (c.ContactName, query { for o in db.Orders do

where (o.CustomerID = c.CustomerID)

select o }) }

“Queries without select”

A “select” is not required. For example, this is a valid query, returning (Product,Employee) pairs:

let salaries =

query { for p in products do

for e in employees do

sortBy (p.Name + "," + e.Name) }

## Standard Query Operators

The following custom query operators are defined as part of the **Microsoft.FSharp.Linq.QueryBuilder** type.

where key

sortBy key

sortByDescending key

thenBy[[4]](#footnote-4) key

thenByDescending key

distinct

groupBy key

groupBy value key

zip e into var

join (for x in e -> k1 = k2)

groupJoin (for x in e -> k1 = k2) into group

leftOuterJoin (for x in e -> k1 = k2) into group

head[[5]](#footnote-5)

headOrDefault

last

lastOrDefault

single

singleOrDefault

nth n

all pred

contains key

exists pred

find pred

averageBy expr

maxBy[[6]](#footnote-6) key

minBy key

sumBy expr

skip n

skipWhile pred

take n

takeWhile pred

In all cases the semantics follows that of LINQ.

## Nullable

F# 3.0 adds minimal support for operations related to Nullable. This is

* A set of query operators for taking min/max/average/sum/orderings of nullable columns in data
* A set of relational operators for relating nullable columns in data

No new conversions are added, and no special nullable operations are supported for arithmetic, conversions or user-defined conversions.

### A note on selecting null rows

Consider the following query:

query

{ for house in db.DVDTable do

where (house.HouseNumber = null) }

where the intent is to select the null rows. This will not typecheck at the moment because in F# “null” is not a Nullable value.

In C#, this works both because there is a special typing rule for “null”, but also because “null” expressions are represented as ConstantExpression(null,typeof<Nullable<int>>) nodes. Equality comparisons against these are translated in a special way by Linq to SQL (and most likely other Linq providers too). For example, the C# equivalent of these give different results:

let v : Nullable<int> = null

query

{ for house in db.DVDTable do

where (house.HouseNumber = v) }

and

query

{ for house in db.DVDTable do

where (house.HouseNumber = null) }

### Query syntax for Nullable joins

The computation expression custom operation translation is extended to cope with the following:

join (for x in e -> k1 ?= k2)

join (for x in e -> k1 =? k2)

join (for x in e -> k1 ?=? k2)

groupJoin (for x in e -> k1 ?= k2) into group

groupJoin (for x in e -> k1 =? k2) into group

groupJoin (for x in e -> k1 ?=? k2) into group

leftOuterJoin (for x in e -> k1 ?= k2) into group

leftOuterJoin (for x in e -> k1 =? k2) into group

leftOuterJoin (for x in e -> k1 ?=? k2) into group

### Query operators for Nullable

The following custom query operations are added to FSharp.Core.dll:

averageByNullable expr

maxByNullable expr

minByNullable expr

sumByNullable expr

sortByNullable keyExpr

sortByNullableDescending keyExpr

thenByNullable keyExpr

thenByNullableDescending keyExpr

### Relational Operators for Nullable

These follow the truth tables in the C# specification.

namespace Microsoft.FSharp.Linq

/// A set of relational operators for working with nullable values

[<AutoOpen>]

module NullableOperators =

open System

/// The '>=' operator where a nullable value appears on the left

val ( ?>= ) : Nullable<'T> -> 'T -> bool when 'T : comparison

/// The '>' operator where a nullable value appears on the left

val ( ?> ) : Nullable<'T> -> 'T -> bool when 'T : comparison

/// The '<=' operator where a nullable value appears on the left

val ( ?<= ) : Nullable<'T> -> 'T -> bool when 'T : comparison

/// The '<' operator where a nullable value appears on the left

val ( ?< ) : Nullable<'T> -> 'T -> bool when 'T : comparison

/// The '=' operator where a nullable value appears on the left

val ( ?= ) : Nullable<'T> -> 'T -> bool when 'T : equality

/// The '<>' operator where a nullable value appears on the left

val ( ?<> ) : Nullable<'T> -> 'T -> bool when 'T : equality

/// The '>=' operator where a nullable value appears on the right

val ( >=? ) : 'T -> Nullable<'T> -> bool when 'T : comparison

/// The '>' operator where a nullable value appears on the right

val ( >? ) : 'T -> Nullable<'T> -> bool when 'T : comparison

/// The '<=' operator where a nullable value appears on the right

val ( <=? ) : 'T -> Nullable<'T> -> bool when 'T : comparison

/// The '<' operator where a nullable value appears on the right

val ( <? ) : 'T -> Nullable<'T> -> bool when 'T : comparison

/// The '=' operator where a nullable value appears on the right

val ( =? ) : 'T -> Nullable<'T> -> bool when 'T : equality

/// The '<>' operator where a nullable value appears on the right

val ( <>? ) : 'T -> Nullable<'T> -> bool when 'T : equality

/// The '>=' operator where a nullable value appears on both left and right sides

val ( ?>=? ) : Nullable<'T> -> Nullable<'T> -> bool when 'T : comparison

/// The '>' operator where a nullable value appears on both left and right sides

val ( ?>? ) : Nullable<'T> -> Nullable<'T> -> bool when 'T : comparison

/// The '<=' operator where a nullable value appears on both left and right sides

val ( ?<=? ) : Nullable<'T> -> Nullable<'T> -> bool when 'T : comparison

/// The '<' operator where a nullable value appears on both left and right sides

val ( ?<? ) : Nullable<'T> -> Nullable<'T> -> bool when 'T : comparison

/// The '=' operator where a nullable value appears on both left and right sides

val ( ?=? ) : Nullable<'T> -> Nullable<'T> -> bool when 'T : equality

/// The '<>' operator where a nullable value appears on both left and right sides

val ( ?<>? ) : Nullable<'T> -> Nullable<'T> -> bool when 'T : equality

Note that some query providers (LINQ-to-SQL) also tolerate the use of nullableValue.HasValue and nullableValue.GetValueOrDefault() in expression fragments. However most do not.

## Terminology: query v. seq v. combinators

It is very important to ensure consistency and fluency between these:

* seq { … } for in-memory queries and more general state machine iterators
* query { … } for in-memory-or-out-of-memory queries.
  + These are a more restricted than seq { … } (no try/finally, while, try/with etc.)
  + These are also more general than seq (you can use sortBy, groupBy, join, averageBy etc.)
  + These can fail at runtime if the IQueryable provider doesn’t support the query operators used.
* Seq.\* combinatory pipelining, often used in prototyping F# query code.

The following table indicates the operations that are supported by each query syntax/combinatory-set

* The elements highlighted in green are new to F# 3.0.
* The elements highlighted in red are not yet implemented in the QUERIES prototype.
* (?) indicates some design issues arise

|  |  |  |
| --- | --- | --- |
| query { … } | Seq.\* combinators | seq { … } |
| for |  | for |
| where if-then | where (also for List, Array, Event, Observable)  filter | if-then |
| yield[[7]](#footnote-7)  select | map | Yield |
| yield! | collect | yield! |
| sortBy  thenBy | sortBy |  |
| sortByDescending  thenByDescending |  |  |
| distinct head  last  exactlyOne nth | distinct  head  last  exactlyOne  nth |  |
| skip[[8]](#footnote-8) skipWhile take takeWhile | skip  skipWhile  take  takeWhile |  |
| groupBy[[9]](#footnote-9) | groupBy |  |
| groupValBy |  |  |
| zip | zip |  |
| join groupJoin leftOuterJoin |  |  |
| averageBy  sumBy | averageBy  sumBy |  |
| maxBy minBy | maxBy, minBy |  |
| all, exists, contains, find | forall, exists, contains, find |  |
|  | tryFind, tryPick, etc. |  |
| headOrDefault[[10]](#footnote-10) lastOrDefault singleOrDefault |  |  |
| maxByNullable[[11]](#footnote-11)  minByNullable  sumByNullable sortByNullable  sortByNullableDescending thenByNullable thenByNullableDescending |  |  |
|  |  | while, try/finally, try/with, match, if-then-else |
| let v = e |  | let v = e |
|  |  | let f x = e |
|  |  | let rec f x = e |
| Combinator compilation for non-IQueryable inputs  LINQ compiled for IQueryable inputs |  | State machine compiled |

# The query { … } and QueryBuilder types

## The query API

val query : QueryBuilder

## How query { … } executes

The extensions to F# computation expressions are primarily to enable the definition of a single computation builder query.

The query computation builder executes in one of two ways

* If the primary input is not IQueryable, then it is compiled to combinators.
* If the primary input is IQueryable, then a quotation is taken of the body of the query (see the Quote member in the “custom computation expression operators” language feature below) and the query.Run function is called. This executes the query by translating it to LINQ calls and expression trees.

## The QueryBuilder type

The QueryBuilder type is the implementation of the query { … } computation builder.

type QueryBuilder =

member For : source:seq<'T> \* body:('T -> seq<'U>) -> seq<'U>

member Zero : unit -> seq<'T>

member YieldFrom : 'T -> seq<'T>

member YieldFrom : 'T -> seq<'T>

member Quote : Quotations.Expr<'T> -> Quotations.Expr<'T>

member Run : Quotations.Expr<'T> -> 'T

## From F# Queries to LINQ Operator Calls

Translating from F# quotations to LINQ operator calls is a non-trivial operation and has the following specification:

1. Replaces uses of builder operators and custom operations with calls to their LINQ equivalents
2. Replaces uses of tuples and records by mutable tuples
3. The translation may fail if this is not a valid IEnumerable or IQueryable query
4. We have to write a full and proper spec of the exact semantics of things like “averageBy” and “maxBy” inside query { … }, e.g.
   1. the semantics of averageBy is given by LINQ’s Enumerable.Average(…) if the primary input is an IEnumerable,
   2. the semantics of averageBy is given by LINQ’s Queryable.Average if the primary input is IQueryable.
   3. Note that in corner cases (e.g. NaN, or equality/comparison for other operators), both of these may be different to Seq.averageBy which uses F# comparison.
5. Translation is precisely specified and tested w.r.t. common Linq query providers. The sprit is: *simple* F# expressions should translate to *simple* Linq expressions, interpretable by common Linq QueryProviders; the shape of resulting Linq expression must follow that of original F# expression.

This largely aligns with C# language spec “7.16.2 Query Expression Translation”

## The Tuple/Record <-> Mutable Tuple translation

It is highly desirable for F# to support LINQ queries that include the construction of (immutable) tuple and (immutable or partially-immutable[[12]](#footnote-12)) record values as part of query result sets, and as intermediate results in query logic. Indeed, the desugaring of custom operators in the query { … } syntax implicitly introduces these values when custom computation expression operators are used.

However, key implementations of IQueryable such as LINQ-to-SQL, LINQ-to-Entities and OData all have problems with constructing immutable values as part of query logic. For all of these, the natural representation of an intermediate value is the compiled form of a C# anonymous type, i.e. a mutable record-like class.

For this reason, the implementation of QueryBuilder performs a translation of (immutable) tuple and (immutable or partially-immutable) record types into mutable tuples, and reverses this translation for outputs.

This is implemented by a flow-directed rewrite of the query:

* Any occurrences of productions **Patterns.NewTuple** within “yield”, “select”, “groupValBy”, “join”, “groupJoin” or “leftOuterJoin” are replaced by instances of a corresponding **Linq.Expressions.Expression.MemberInit** on the corresponding MutableTuple type.
* This replacement if flowed through the structure of the query by making corresponding type substitutions, and replacing accesses to tuple fields by accesses to mutable tuple fields.
* A reverse-translation is performed to the output of the query, that convert outputs from the query back into tuples. This is applied to the inner core of the query that produces values, ignoring aggregation operations such as SumBy and AverageBy. If the output of the core of the query includes a sequence of IGrouping produced by a GroupBy or GroupValBy operator then the contents of each group are also transformed.

A corresponding translation is performed for **Patterns.NewRecord** are replaced by instances of a corresponding **Linq.Expressions.Expression.MemberInit** on the corresponding MutableTuple type.

* Any occurrences of quotations matching **Patterns.PropertyGet** or **Patterns.TupleGet** are replaced by instances of a corresponding **Linq.Expressions.Expression.PropertyGet** on the corresponding MutableTuple type.

The tuple type elimination takes into account tuple types of size > 7.

The translation does not take into account hand-coded immutable F# class, struct or interface types.

# Standard Data/Service/Resource Type Providers

## SqlDataConnection<…>

This provider embeds a set of generated types to access a database via the given connection string using LINQ-to-SQL. For example:

[<Generate>]

type NorthwndSchema = SqlDataConnection< "…" >

let db = new NorthwndSchema.GetDataContext()

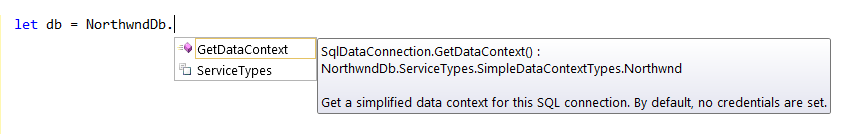
db.DataContext.Log <- System.Console.Out

let customers =

query { for c in db.Customers do

yield c }

The top-level intellisense list is shown below:



### Static Parameters

The set of static parameters is shown below, and mostly corresponds to a subset of those accepted by **sqlmetal.exe**.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Type | Description | Default |
| ConnectionString | string | The connection string for the database connection. | Exactly one of ConnectionString and ConnectionStringName is required. |
| ConnectionStringName | string | The app.config configuration key setting for the connection string for the database connection. At design-time the app.config is read from the project/script directory, at runtime the app.config is read using System.Configuration.ConfigurationManager, and if that fails then the read the fully qualified path of the design-time config file[[13]](#footnote-13) | Exactly one of ConnectionString and ConnectionStringName is required. |
| ConnectionStringConfigFileName | string | The name of the file to use instead of app.config or web.config | Can only be set of ConnectionStringName is also set. |
| LocalSchemaFile | string | Local file in which to store the latest .dbml for the database schema | empty |
| ForceUpdate | bool | Require that the direct connection to the database be available at design/compile-time and the local schema file is refreshed | true |
| User | string | Login User ID | Use Windows Integrated Authentication |
| Password | string | Login User Password | Use Windows Integrated Authentication |
| Views | Boolean | Extract database views | True |
| Functions | Boolean | Extract database functions | True |
| StoredProcedures | Boolean | Extract database stored procedures | True |
| Timeout | Integer | Timeout value to use when SqlMetal accesses the database | 0 |
| Pluralize | Boolean | Automatically pluralize or singularize class and member names using English language rules | False |
| DataContext | String | The name of data context class | derived from database name |
| EntityBaseTypeName | String | Base class of entity classes in the types | entities have no base class |
| Serializable | Boolean | Generate (uni-directional) serialzable classes | false |

### The Generated Type Space

For the declaration

[<Generate>]

type MyDb = SqlDataConnection<*parameters*>

We assume that ***DataContextTypeName*** is the single type generated by **sqlmetal.exe** which has base type **System.Data.Linq.DataContext.** This will be the value of the staticparameter **DataContext** if it is given, else the name chosen by **sqlmetal.exe** based on the input parameters, e.g. **Northwnd**.

The complete set of generated types is as follows:

**type** MyDb

**Description**: The overall container type

**method** GetDataContext()

**Description**: A method returning a simplified view of the data context. The method creates and returns

**new** MyDb.ServiceTypes.SimpleDataContextTypes.*DataContextTypeName* (*connectionString*)

**Return type**: MyDb.ServiceTypes.SimpleDataContextTypes.*DataContextTypeName*

**method** GetDataContext(*connectionString*: string)

**Description**: A method returning a simplified view of the data context using the given connection string dynamically. The method creates and returns

**new** MyDb.ServiceTypes.SimpleDataContextTypes.*DataContextTypeName*(*connectionString*)

**Return type**: MyDb.ServiceTypes.SimpleDataContextTypes.*DataContextTypeName*

**type** MyDb.ServiceTypes

**Description**: Contains the embedded full types and simplified types for the service.

**type** MyDb.ServiceTypes.\*

**Description**: The embedded types generated by sqlmetal.exe, relocated during static linking. Contains the full set of types generated by **sqlmetal.exe** for the database mapping. One of these will be named ***DataContextTypeName***. and will have base type **System.Data.Linq.DataContext**.

**type** MyDb.ServiceTypes.SimpleDataContextTypes

**Description**: Contains the generated, relocated types representing the simplified view of the context types.

**type** MyDb.ServiceTypes.SimpleDataContextTypes.*DataContextTypeName*

**Description**: Represents the simplified view of the data context returned by GetDataContext(). This wraps a single instance of MyDb.ServiceTypes.*DataContextTypeName*.

**method** \*

Contains one method for each method on the full context type returning **System.Data.Linq.ISingleResult<\_>**.

**property** \*

Contains one property for each property on the full context type returning **System.Data.Linq.Table<\_>**.

**property** Connection : System.Data.Common.DbConnection

**Description**: Gets the ADO.NET connection used by the data context.

**property** DataContext

**Description**: Gets the full data context, of type **System.Data.Linq.DataContext**

### Background process execution

This provider runs **sqlmetal.exe**. The binary is located as a .NET Framework 4.0 SDK tool in the directory

HKLM\SOFTWARE\Microsoft\Microsoft SDKs\Windows\v7.0A\WinSDK-NetFx40Tools\InstallationFolder

This provider also runs **csc.exe**. The binary is located in:

System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory()

### Debugging uses of this provider

It is not expected that any particular issues will be detected in the step-over, step-through and breakpoint debugging experience for this provider. The internally generated C# stub code will not be available for use during debugging and should never be seen by the use.

### Diagnostics for this provider

Good diagnostics must be given on the following conditions:

* Invalid parameters, e.g. invalid type name to DataContext
* Invalid connection string
* Database is not reachable
* Lack of permissions to access database
* Incorrect file name or extension for LocalSchemaFile

### Caching and Liveness for this provider

See static parameters **LocalSchemaFile** and **ForceUpdate.** When **ForceUpdate** is false, the provider reacts to changes in the **LocalSchemaFile**.

This provider does not react to schema changes on the database nor poll for changes on the database.

## DbmlFile<…>

This provider embeds a set of generated types to access a database corresponding to the given DBML description using LINQ-to-SQL. The embedded types are the same as those generated by **sqlmetal.exe**.

[<Generate>] type NorthwndSchema = DbmlFile< "northwnd.dbml" >

let db = new NorthwndSchema.NORTHWND(“…connection string…”, Log=System.Console.Out)

let customers =

query { for c in db.Customers do

yield c }

A connection string must be given when instantiating the provider, unless one is already present in the DBML file.[[14]](#footnote-14)

### Static Parameters

The set of static parameters is shown below, and mostly corresponds to a subset of those accepted by **sqlmetal.exe**.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Type | Description | Default |
| File | String | The name of the DBML file for the database mapping. |  |
| DataContext | String | The name of data context class | derived from database name |
| EntityBaseTypeName | String | Base class of entity classes in the types | entities have no base class |
| Serializable | Boolean | Generate (uni-directional) serialzable classes | false |

### Background process execution

This provider runs **sqlmetal.exe**. The binary is located as a .NET Framework 4.0 SDK tool in the directory

HKLM\SOFTWARE\Microsoft\Microsoft SDKs\Windows\v7.0A\WinSDK-NetFx40Tools\InstallationFolder

This provider also runs **csc.exe**. The binary is located in:

System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory()

### Diagnostics for this provider

Good diagnostics must be given on the following conditions:

* Invalid parameters, e.g. invalid type name to DataContext
* Bad file name

### Debugging uses of this provider

It is not expected that any particular issues will be detected in the step-over, step-through and breakpoint debugging experience for this provider. The internally generated C# stub code will not be available for use during debugging and should never be seen by the use.

The SQL queries generated can be sent to an output stream by specifying the DataContext.Log property of the simplified data context object.

### Caching and Liveness for this provider

This provider reacts to changes in the saved file on disk and raises the provider Invalidate event when these changes occur.

## SqlEntityConnection<…>

This provider embeds a set of generated types to access a database via the given connection string using LINQ-to-Entities. For example:

[<Generate>] type NorthwndSchema = SqlDataConnection< "…”" >

let db = new NorthwndSchema.NORTHWND(Log=System.Console.Out)

let customers =

query { for c in db.Customers do

yield c }

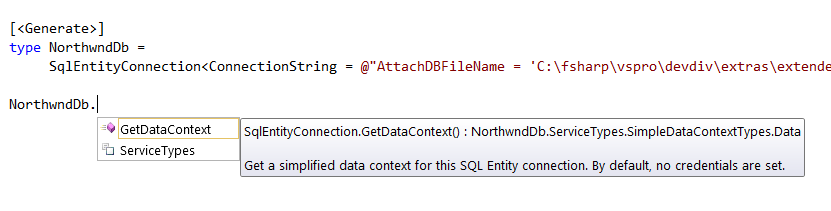
### Static Parameters

The set of static parameters is shown below, and mostly corresponds to a subset of those accepted by **edmgen.exe**.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Type | Description | Default |
| ConnectionString | string | The connection string for the database connection. | Exactly one of ConnectionString and ConnectionStringName is required. |
| ConnectionStringName | string | The app.config configuration key setting for the connection string for the database connection. At design-time the app.config is read from the project/script directory, at runtime the app.config is read using System.Configuration.ConfigurationManager, and if that fails then the read the fully qualified path of the design-time config file[[15]](#footnote-15) | Exactly one of ConnectionString and ConnectionStringName is required. |
| ConnectionStringConfigFileName | string | The name of the file to use instead of app.config or web.config | Can only be set of ConnectionStringName is also set. |
| LocalSchemaFile | string | Local file in which to store the latest .dbml for the database schema | empty |
| ForceUpdate | bool | Require that the direct connection to the database be available at design/compile-time and the local schema file is refreshed | true |
| Provider | string | The name of the ADO.NET data provider to  be used for ssdl generation | System.Data.SqlClient |
| EntityContainer | string | The name to use for the EntityContainer in the conceptual model | “EntityContainer” |
| Pluralize | Boolean | Automatically pluralize or singularize class and member names using English language rules | false |
| SuppressForeignKeyProperties | Boolean | Exclude foreign key properties in entity  type definitions. | false |

### The Generated Type Space

For example:



For the declaration

[<Generate>]

type MyDb = SqlEntityConnection<*parameters*>

We assume that ***DataContextTypeName*** is the single type generated by **edmgen.exe** which has base type **System.Data.Objects.ObjectContext.** This will be the value of the staticparameter **DataContext** if it is given, else the name chosen by **edmgen.exe** based on the input parameters, e.g. **Northwnd**.

The types are as follows:

**type** MyDb

**Description**: The overall container type

**method** GetDataContext()

**Description**: A method returning a simplified view of the data context, instantiating an instance of MyDb.ServiceTypes.*DataContextTypeName* with Entity Framework connection string

metadata=res://\*/*entityNamespaceName*.csdl|res://\*/*entityNamespaceName*.ssdl|res://\*/*entityNamespaceName*.msl;provider=*provider*;provider connection string="*connectionString*"

Here the connection string is referring to resources and *entityNamespaceName* is SqlEntityConnection*NNN*.\*, see “Details of statically linked code” below.

**Return type**: MyDb.ServiceTypes.SimpleDataContextTypes.*DataContextTypeName*

**method** GetDataContext(*providerConnectionString*: string)

**Description**: A method returning a simplified view of the data context with the same Entity Framework connection string as above, except the given provider connection string replace the parameter *connectionString.*

**Return type**: MyDb.ServiceTypes.SimpleDataContextTypes.*DataContextTypeName*

**type** MyDb.ServiceTypes

**Description**: The embedded types generated by edmgen.exe. Contains the full set of types generated by edmgen.exe for the database mapping.

**type** MyDb.ServiceTypes.SimpleDataContextTypes

**Description**: Contains the simplified view of the context types

**type** MyDb.ServiceTypes.SimpleDataContextTypes.*DataContextTypeName*

**Description**: Represents the simplified view of the data context returned by GetDataContext(). This wraps a single instance of MyDb.ServiceTypes.*DataContextTypeName*

**property** \*

**Description**: Contains one property for each property on the full context type which returns a System.Data.Objects.ObjectSet <\_>.

**property** Connection

**Description**: Gets the ADO.NET connection used by the data context

**property** DataContext

**Description**: Gets the full data context, of type System.Data.Linq.DataContext

### Details of statically linked code

For compiled code, a use of a SqlEntityConnection generation declaration results in the addition of statically linked code into the generated .NET assembly.[[16]](#footnote-16)

This declaration is made up of the merge of the types of these two assemblies:

* The assembly generated by edmgen.exe containing (without relocation):
  + **types:**

**type** SqlEntityConnection*NNN*.\*

Here *NNN* increments for each SqlEntityConnection declaration encountered textually as the assembly is compiled. These type names are implicitly “taken” by any assembly containing this statically linked code, and defining them in F# code will result in a duplicate-type-definition error.

* + **resources:**

**resource** SqlEntityConnection*NNN*.csdl

**resource** SqlEntityConnection*NNN*.ssdl

**resource** SqlEntityConnection*NNN*.mdl

Three statically linked .NET managed resources for the MDL, SSDL and CSDL resources that make up the contents of the EDMX file. These resources are used by EF at runtime.

These resources can be seen in the “MANIFEST” section of the ILDASM of a generated binary as, for example:

.mresource public SqlEntityConnection1.csdl

{

// Offset: 0x00000000 Length: 0x00008385

}

These resources are referenced by the EF connection string used to create the simplified data context vies.

These resource names are implicitly “taken” by any assembly containing this statically linked code.

* An additional assembly generated by the provider containing the types:
  + **types:**

**type** MyDb.ServiceTypes.SimleDataContextTypes.\*

**type** MyDb

### Background process execution

This provider runs **edmgen.exe**. The binary is located as a .NET Framework 4.0 SDK tool in the directory

HKLM\SOFTWARE\Microsoft\Microsoft SDKs\Windows\v7.0A\WinSDK-NetFx40Tools\InstallationFolder

This provider also runs **csc.exe**. The binary is located in:

System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory()

### Debugging uses of this provider

It is not expected that any particular issues will be detected in the step-over, step-through and breakpoint debugging experience for this provider. The internally generated C# stub code will not be available for use during debugging and should never be seen by the use.

### Diagnostics for this provider

Good diagnostics must be given on the following conditions:

* Invalid parameters, e.g. invalid type name to DataContext
* Invalid connection string
* Database is not reachable
* Lack of permissions to access database
* Incorrect file name or extension for LocalSchemaFile

### Caching and Liveness for this provider

See **LocalSchemaFile** and **ForceUpdate**.

When **ForceUpdate** is false, the provider reacts to changes in the **LocalSchemaFile**.

This provider does not react to schema changes on the database.

## EdmxFile<…>

This provider embeds a set of generated types to access a database corresponding to the given EDMX description (which includes the Conceptual, Storage and Mapping models for Entity Framework 4.0). The types are generated from the Conceptual model (CSDL) portion of the EDMX file. The embedded types are the same as those generated by **edmgen.exe** or the corresponding Visual Studio actions in C#/VB.

[<Generate>] type NorthwndSchema = EdmxFile< "northwnd.edmx" >

let db = new NorthwndSchema.NORTHWNDEntities(connString)

let customers =

query { for c in db.Customers do

select c }

As a side effect, the CSDL (Conceptual), SSDL (Storage) and MSL (Mapping) portions of the EDMX file are embedded as resources into the generated assembly, using names filebase.csdl, filebase.ssdl and filebase.msl.

EDMX is strongly associated with Visual Studio designer support for EDMX models. See below, “project and file templates”.

Note the Entity Framework uses its own notion of connection strings, which contain an embedded “provider connection string” to access the underlying database. For example:

let connString =

"metadata=res://\*/northwnd.csdl|res://\*/northwnd.ssdl|res://\*/northwnd.msl;" +

"provider=System.Data.SqlClient;" +

"provider connection string=\"Data Source=.\SQLEXPRESS;" +

"AttachDbFilename=" + dbPath + ";" +

"Integrated Security=True;" +

"Connect Timeout=30;" +

"User Instance=True;" +

"MultipleActiveResultSets=True\""

### Static Parameters

The set of static parameters is shown below.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Type | Description | Default |
| File | string | The EDMX file containing the conceptual, storage and mapping schema descriptions |  |

### Background process execution

This provider runs **edmgen.exe**. The binary is located as a .NET Framework 4.0 SDK tool in the directory

HKLM\SOFTWARE\Microsoft\Microsoft SDKs\Windows\v7.0A\WinSDK-NetFx40Tools\InstallationFolder

This provider also runs **csc.exe**. The binary is located in:

System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory()

### Diagnostics for this provider

Good diagnostics must be given on the following conditions:

* Bad file name

### Debugging uses of this provider

It is not expected that any particular issues will be detected in the step-over, step-through and breakpoint debugging experience for this provider. The internally generated C# stub code will not be available for use during debugging and should never be seen by the use.

### Caching and Liveness for this provider

This provider reacts to changes in the saved file on disk and raises the provider Invalidate event when these changes occur.

## ODataService<…>

This provider embeds a set of generated types to provide strongly typed access to the given OData service. The embedded types are the same as those generated by **svcutil.exe** or accessed via **Add Service Reference**.

[<Generate>] type Northwest = ODataService<"http://services.odata.org/Northwind/Northwind.svc/">

let db = Northwest.NorthwindEntities()

Queries can be used against the service:

query { for c in db.Customers do

yield c }

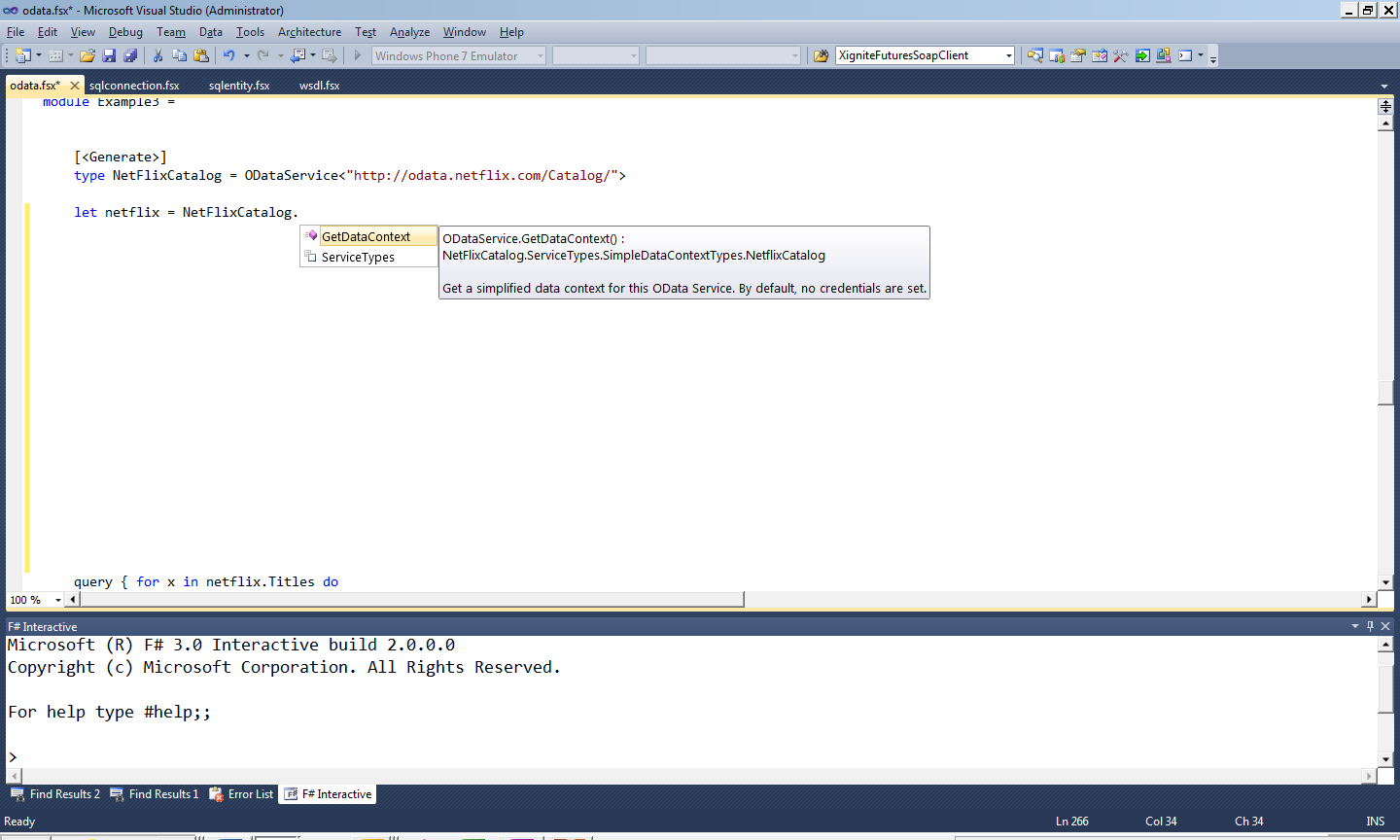
### Static Parameters

The set of static parameters is shown below, and mostly corresponds to those accepted by **datasvcutil.exe**.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Type | Description | Default |
| ServiceUri | String | The URI for the OData service |  |
| LocalSchemaFile | string | Local file in which to store the latest service metadata for the service | empty |
| ForceUpdate | bool | Require that the direct connection to the service be available at design/compile-time and the local service file is refreshed | true |
| DataServiceCollection | bool | Generate collections derived from DataServiceCollection | false |

### The Generated Type Space

For example:



For the declaration

[<Generate>]

type MyService = ODataService<*parameters*>

The generated types are as follows, where ***ServiceTypeName*** is the name of the single type generated by **datasvcutil.exe** which has base type **System.Data.Services.Client.DataServiceContext**. This is usually the name in the schema for the service, e.g. **NetflixCatalog**.

**type** MyService

**Description**: The overall container type

**method** GetDataContext()

**Description**: A method returning a simplified view of the data context. The method creates and returns

**new** MyService.ServiceTypes.SimpleDataContextTypes.*ServiceTypeName* (**new** System.Uri(*serviceUri*))

where *serviceUri* is the valueof the ServiceUri static parameter.

**Return type**: MyService.ServiceTypes.SimpleDataContextTypes.*ServiceTypeName*

**method** GetDataContext(*uri*: System.Uri)

**Description**: A method returning a simplified view of the service context. The method creates and returns

**new** MyService.ServiceTypes.SimpleDataContextTypes.*ServiceTypeName* (*uri*)

**Return type**: MyService.ServiceTypes.SimpleDataContextTypes.*ServiceTypeName*

**type** MyService.ServiceTypes

**Description**: The embedded types generated by datasvcutil.exe.

**type** MyService.ServiceTypes.\*

Contains the full set of types generated by **datasvcutil.exe** for the service mapping as nested types. Precisely one of these will be the full context type named *ServiceTypeName* (e.g.NetflixCatalog).

**type** MyService.ServiceTypes.SimpleDataContextTypes

**Description**: Contains the generated, relocated types representing the simplified view of the context types.

**type** MyService.ServiceTypes.SimpleDataContextTypes.*ServiceTypeName*

**Description**: Represents the simplified view of the data context returned by GetDataContext(). This wraps a single instance of MyService.ServiceTypes.*ServiceTypeName*.

**property \***

**Description**: Contains one property for each property on the full context type which returns a System.Data.Services.Client.DataServiceQuery<\_>, e.g. the Netflix service type contains the property **Titles**.

**property** Credentials

**Description**: Gets or sets the authentication information used dynamically by each query for this data context object. Does not affect the credentials used during type-checking.

**Example**:

let netflix = NetFlixCatalog.GetDataContext()

netflix.Credentials <- NetworkCredentials("user","password")

Gets or sets the authentication information used by each query for this data context object

**property** DataContext

**Description**: Gets the full data context, of type System.Data.Linq.DataContext

### Permitted Queries

See the MSDN documentation for OData LINQ queries for the query forms that are allowed by OData. Operations such as grouping and joining are not supported, and many sorting operations are highly restricted.

Any restrictions are not F#-specific. Queries should in general return either individual entities, or tuples of projections. This means that:

query { for t in netflix.Titles do

yield t }

and

query { for t in netflix.Titles do

yield (t.Name, t.AverageRating) }

are ok, however this is not:[[17]](#footnote-17)

query { for t in netflix.Titles do

// OData queries do not allow you to return a single property projection

// System.NotSupportedException: Navigation properties can only be selected from a single resource. Specify a key predicate to restrict the entity set to a single instance

//

// Instead, place this in a tuple, e.g. – yield (t, t.Name)

yield t.Name }

### Background process execution

This provider runs **datasvcutil.exe**. The binary is located as a .NET Framework 4.0 SDK tool in the directory

System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory()

This provider also runs **csc.exe**. The binary is located in:

System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory()

### Diagnostics for this provider

Good diagnostics must be given on the following conditions:

* Invalid service URI
* Unreachable service URI
* Lack of permissions to access URI
* Invalid local schema file name
* The local schema file can’t be update because it is read-only or can’t otherwise be written
* Running datasvcutil.exe fails for some reason
* datasvcutil.exe can’t be found in the expected place or directory name doesn’t exist
* Incorrect file name or extension for LocalSchemaFile (.csdl is required)

### Debugging uses of this provider

It is not expected that any particular issues will be detected in the step-over, step-through and breakpoint debugging experience for this provider. The internally generated C# stub code will not be available for use during debugging and should never be seen by the use.

The REST requests used dynamically can be seen by using the db.DataContext.SendingRequest event, for example:

let db = Northwest.GetDataContext()

db.DataContext.SendingRequest.Add (fun x -> printfn "requesting %A" x.Request.RequestUri)

### Caching and Liveness for this provider

See LocalSchemaFile and ForceUpdate

This provider does not react to schema changes on the service.

## WsdlService<…>

This provider embeds a set of generated types to provide strongly typed access to the given WSDL web service. The embedded types are the same as those generated by **svcutil.exe** or accessed via **Add Service Reference**.

open Microsoft.FSharp.Data.TypeProviders

[<Generate>]

type XigniteFuturesService = WsdlService<"http://www.xignite.com/xFutures.asmx?WSDL">

let service = new XigniteFuturesService.XigniteFuturesSoapClient()

### Static Parameters

The set of static parameters is shown below, and mostly corresponds to those accepted by **svcutil.exe**.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Type | Description | Default |
| ServiceUri | string | The URI for the WSDL service |  |
| LocalSchemaFile | string | Local file in which to store the .wsdl for the service |  | |
| ForceUpdate | bool | Require that the direct connection to the service be available at design/compile-time and the local service file is refreshed |  | |
| DataContractOnly | bool | Generate code for Data Contract types only. Service Contract types will not be generated | false |
| MessageContract | bool | Generate Message Contract types | false |
| EnableDataBinding | bool | Implement the INotifyPropertyChanged interface on all DataContract types to enable data binding. | false |
| Serializable | bool | Generate classes marked with the Serializable Attribute. | false |
| Async | bool | Generate both synchronous and asynchronous method signatures. | false |
| CollectionType | string | A fully-qualified or assembly-qualified name of the type to use as a collection data type when code is generated from schemas. | empty |
| DataContractSerializer | bool | Generate data types that use the Data Contract Serializer for serialization and deserialization | false |

### The Generated Type Space

For example:



For the declaration

[<Generate>]

type MyService = WsdlService<*parameters*>

The generated types are as follows, where ***ServiceTypeName*** is the name of the single type generated by **svcutil.exe** which has base type **System.Data.Services.Client.DataServiceContext**. This is usually the name in the WSDL for the client for the service, e.g. **XigniteFuturesSoapClient**.

**type** MyService

**Description**: The overall container type

**method** GetHttp*SeriviceTypeName*()

**Description**: A method returning a simplified view of the data context. The method creates and returns

**new** *ServiceTypeName*(**new** BasicHttpBinding(), EndpointAddress(serviceUri))

where *serviceUri* is the valueof the ServiceUri static parameter.

**Return type**: MyService.ServiceTypes.SimpleDataContextTypes.*ServiceTypeName*

**type** MyService.ServiceTypes

**Description**: The embedded types generated by svcutil.exe.

**type** MyService.ServiceTypes.\*

**Description**: Contains the full set of types generated by svcutil.exe for the service mapping as nested types.

### Permitted Queries

LINQ queries can’t be used against WSDL web services.

### Background process execution

This provider runs **svcutil.exe**. The binary is located as a .NET Framework 4.0 SDK tool in the directory

HKLM\SOFTWARE\Microsoft\Microsoft SDKs\Windows\v7.0A\WinSDK-NetFx40Tools\InstallationFolder

This provider also runs **csc.exe**. The binary is located in:

System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory()

### Diagnostics for this provider

Good diagnostics must be given on the following conditions:

* Invalid service URI
* Unreachable service URI
* Lack of permissions to access URI
* Invalid local schema file name
* The local schema file can’t be update because it is read-only or can’t otherwise be written
* Running svcutil.exe fails for some reason
* Svcutil.exe can’t be found in the expected place or registry key doesn’t exist

Incorrect extension for LocalSchemaFile

### Debugging uses of this provider

It is not expected that any particular issues will be detected in the step-over, step-through and breakpoint debugging experience for this provider. The internally generated C# stub code will not be available for use during debugging and should never be seen by the use.

### Caching and Liveness for this provider

See LocalSchemaFile and ForceUpdate. Use a local schema file if the 'LocalSchemaFile' parameter is present (IsNullOrWhiteSpace). Otherwise use a temporary file to hold the metadata description of the service. Always force the update of the local schema file if 'forceUpdate' is true and the downloaded contents have changed.

This provider does not react to schema changes on the service.

## ResxFile<…>

This provider embeds the .resources formed by compiling the given .resx file. The embedded types are the same as those generated by **resgen.exe** or by the single file generator used from C#.

open Microsoft.FSharp.Data.TypeProviders

[<Generate>]

type ResourceSet = ResxFile<"Resources.resx">

printfn "ABC = %s" ResourceSet.Resources.String1

printfn "DEF = %s" ResourceSet.Resources.String2

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Type | Description | Default |
| File | string | The name of the RESX resource file to embed |  |

### Diagnostics for this provider

Good diagnostics must be given on the following conditions:

* Invalid file name
* File doesn’t exist or is unreadable

### Debugging uses of this provider

It is not expected that any particular issues will be detected in the step-over, step-through and breakpoint debugging experience for this provider. The internally generated C# stub code will not be available for use during debugging and should never be seen by the use.

### Caching and Liveness for this provider

This provider reacts to changes in the saved file on disk and raises the provider Invalidate event when these changes occur.

# Language Support for Queries

## Lexing

We now allow operators to start with “?”. This skips the character to compute precedence, as for “.”. This is to allow nullable support for F#.

The operator ? is still special (and infix), and the operator ?? is still reserved.

## Parsing

There are no changes to parsing rules for F#.

## Language Support For Calling LINQ Methods

The existing implicit conversion of functions to delegates at member calls in F# 2.0 is extended to include implicit conversion of functions to Expression<DelegateType> at member calls.

A call to **Microsoft.FSharp.Linq.RuntimeHelpers.LeafExpressionConverter.QuotationToLambdaExpression** is inserted implicitly around an implicit quotation of the argument expression.

## Computation Expression Custom Operations

A computation expressionhas *custom operations* if, for an expression

*builder* { *expr* }

the *builder* is a value whose type has one or more members or extension members with a CustomOperationAttribute.

Additionally, a computation expression may now also have a Quote member indicating that the expression is automatically quoted after all translation, and the member applied.

The parse trees in an extended computation expression are given special meaning in the pre-processing phase are as follows:

query { *qexpr*}

*qexpr* :=

| for *patpvs* in *expr* do *qexpr*

| let *patpvs* = *expr* in *qexpr*

| let! *patpvs* = *expr* in *qexpr*

| *qexpr*; into *pat; qexpr*

| *qexpr*; *qexpr*

| join (for *patpvs* in *expr* -> *expr*); *qexpr*

| groupJoin (for *patpvs* in *expr* -> *expr*) into *pat*; *qexpr*

| leftOuterJoin (for *patpvs* in *expr* -> *expr*) into *pat*; *qexpr*

| zip *expr* into *pat*; *qexpr*

| *ident* *arg1* … *argN*

| *expr*

In the case of “into pat”, the “pat” is, in syntax terms, an expression interpreted as an F# pattern. The expression must be of the following syntactic from:

*expr-as-pat* :=

| *ident*

| (*ident*)

| (*ident*, …, *ident*)

The syntactic rewrite for computation expressions is replaced by the following. The translation **Q** is defined recursively according to the following rules:

**Q*vs*** *inp* qexpr =

match qexpr with

| let binds in cexpr

🡪 **Q*vs*** *inp*[let binds in ◊] cexpr

| let! pat vs2 = expr in cexpr

🡪 **Q*vs+vs2*** *inp*[b.Bind(*expr*, (fun pat -> ◊)] cexpr

| do expr in cexpr

🡪 **Q*vs*** *inp*[expr; ◊] cexpr

| do! expr in cexpr

🡪 **Q*vs*** *inp*[b.Bind(*expr*, (fun () -> ◊)] cexpr

| yield expr

🡪 *inp*[b.Yield(expr)]

| yield! expr

🡪 *inp*[b.YieldFrom(expr)]

| return expr

🡪 *inp*[b.Return(expr)]

| return! expr

🡪 *inp*[b.ReturnFrom(expr)]

| use pat vs2 = expr in cexpr

🡪 **Q*vs+vs2*** *inp*[b.Using(expr, (fun pat -> ◊)] cexpr

| use! v = expr in cexpr

🡪 **Q*vs*** *inp* “let! v = expr in use v = v in cexpr”

| if expr then cexpr

🡪 **Q*vs*** *inp*[if expr then ◊] cexpr

| if expr then cexpr*0* else cexpr*1*

🡪 *inp*[if expr then **C** ◊ cexpr*0* else **C** ◊ cexpr*1*]

| match expr with pat*i* -> cexpr*i*

🡪 *inp*[match expr with pat*i* -> **C** ◊ cexpr*i*]

| for patvs2 in expr do cexpr

🡪 **Q*vs+vs2*** *inp*[b.For({| expr |}E, (fun pat -> ◊))] cexpr

| for ident = expr*1* to expr*2* do *c*expr*3*

🡪 **Q*vs*** *inp* “for *ident* in seq { expr*1* .. expr*2* } do *cexpr3* “

| while expr do cexpr

🡪 **Q*vs*** *inp*[b.While((fun () -> expr), b.Delay(fun () -> ◊))] cexpr

| try cexpr with pat*i* -> cexpr*i* |

🡪 *inp*[b.TryWith(b.Delay(fun () -> **C** ◊ cexpr ), (fun v ->

match v with

| (pat*i*:exn) -> **C** ◊ cexpri

| \_ -> raise exn)

| try cexpr finally expr

🡪 *inp*[b.TryFinally(b.Delay(fun () -> **C** ◊ cexpr ), (fun () -> expr))

| qop; cexpr when maintains-var-space *qop*

🡪 **Q*vs***“for vs in **Q*vs*** *inp* qop do yield ◊” cexpr

| qop; cexpr when not maintains-var-space *qop*

🡪 **Q*0***“**Q*vs*** *inp* qop” cexpr

| cexpr*0*; cexpr

🡪 **Q*vs*** *inp*[b.Combine(**C** ◊ cexpr*0*, (fun () -> ◊) cexpr

| expr; cexpr

🡪 **Q*vs*** *inp*[expr; ◊] cexpr

// Custom Query Operator rules: e.g. “distinct”

| *ident* *arg1* when is-custom-query-op *ident* && has-ProjectionParameterAttribute *arg1*

🡪 *b.CustomOperation* (*inp*@*vs,* fun *vs1* -> *arg1*)

A similar translation applies for custom operators with 0 or more arguments

where none, some or all of the arguments are ProjectionParameterAttribute

| into *pat1* ; *cexpr*

🡪 **Q*vs*** (for *pat* in *inp*@*vs* do *qexpr* )

| other-expr

🡪 *inp*[other-expr; b.Zero()]

Plus these cases for “join”, “groupJoin”, “zip” and “leftOuterJoin”

| join (for *p2*vs2in *inp2* -> *key1 = key2*); *qexpr*

🡪 b.Join*(inp1*@*vs1, inp2*, (fun *vs1* -> *key1*), (fun *p2* -> *key2*), (fun *vs1* *p2* -> qtrans*vs1+vs2* *inp0* *qexpr*))

| groupJoin (for *p2* in *inp2* -> *key1 = key2*) into *grouppat*; *qexpr*

🡪 b.GroupJoin(*inp1*@*vs1,inp2*,(fun vs1 -> *key1*),(fun *p2* -> *key2*),(fun *vs1 grouppat* -> …))

| zip *inp2* into *pat*; *qexpr*

🡪 b.Zip(*inp1*@*vs1,inp2*,(fun *vs1 grouppat* -> …))

| leftOuterJoin (for *p2* in *inp2* -> *key1 = key2*) into *group*; *qexpr*

🡪 b.LeftOuterJoin*(inp1*@*vs1,inp2*,(fun vs1 -> *key1*),(fun *p2* -> *key2*),(fun *vs1 group* -> …))

## Attributes for Specifying Computation Expression Custom Operations

The following two attributes are used to specify custom operations: **ProjectionParameterAttribute** and **MaintainsVariableSpaceAttribute**.

The attributes are only significant on custom operations and are otherwise ignores.

The following shows how these are orthogonal and can be present in any combination.

Example:

module QueryOperators =

let head (x:seq<'T>) = Seq.head x

[<MaintainsVariableSpace>]

let where ([<ProjectionParameter>] f) x = Seq.filter f x

let select ([<ProjectionParameter>] f) x = Seq.map f x

[<MaintainsVariableSpace>]

let skip n x = Seq.skip n x

### CustomOperationAttribute

Indicates that a member on a builder type implements a custom operation in a computation expression.

The parameter is the name of the custom operation.

### ProjectionParameterAttribute

Indicates that, when a custom operation is used in a computation expression, a parameter is automatically parameterized by the variable space of the computation expression

[<AttributeUsage(AttributeTargets.Parameter,AllowMultiple=false)>]

type ProjectionParameterAttribute =

new : unit -> ProjectionParameterAttribute

inherit System.Attribute

### MaintainsVariableSpaceAttribute

Indicates that, when a custom operation is used in a computation expression, it produces a computation with the same variable space as the input computation.

[<AttributeUsage(AttributeTargets.Method ||| AttributeTargets.Property,AllowMultiple=false)>]

type MaintainsVariableSpaceAttribute =

new : unit -> MaintainsVariableSpaceAttribute

inherit System.Attribute

## Example: Specifying Your Own Query Builder

This example uses Event/IObservable programming in a mini-Rx fashion. It does not use quotations.

Definitions:

type EventBuilder() =

member \_\_.For(ev:IObservable<'T>, loop:('T -> #IObservable<'U>)) : IObservable<'U> =

failwith "NYI"

member \_\_.Yield(v:'T) : IObservable<'T> = failwith ""

member \_\_.Run(x:Expr<'T>) = failwith "NYI"

[<MaintainsVariableSpace; CustomOperation("where")>]

member \_\_.Where (x, [<ProjectionParameter>] f) = Observable.filter f x

[<CustomOperation("select")>]

member \_\_.Select (x, [<ProjectionParameter>] f) = Observable.map f x

[<CustomOperation("choose")>]

member \_\_.Choose (x, [<ProjectionParameter>] f) = Observable.choose f x

Usage:

let f = new System.Windows.Forms.Form()

let progressiveSumOfAllClicksOnLeftPartOfForm() =

eventQuery { for x in f.MouseClick do

where (x.X < 100) }

# Leaf Expression Translation to LINQ

FSharp.Core includes an API to convert F# quotations to LINQ expression trees, for a subset of F# quotations corresponding to C# expressions. These helpers are made public to make it easier to write alternative translators.

## The Translation API

The primary portion of the API is as follows:

namespace Microsoft.FSharp.Linq.RuntimeHelpers

module LeafExpressionConverter =

val QuotationToExpression : Quotations.Expr -> Expression

val QuotationToLambdaExpression : Quotations.Expr<'T> -> Expression<'T>

val EvaluateQuotation : Quotations.Expr -> obj

Each of the above perform a translation from F# quotations to LINQ expression trees. The translation only accepts F# quotations which have a translation which is in the subset of LINQ expressions generated by the C# compiler. In particular, this means that the translation functions do NOT accept

* Quotations whose translations involve a call to a method returning a “void” type
* Sequential execution, for-loops, while-loops, “Set” operations
* “let rec”

Otherwise a NotSupportedException is raised.

### Microsoft.FSharp.Linq.RuntimeHelpers. LeafExpressionConverter.QuotationToExpression

Calls to this method are made implicitly by the implementation of QueryBuilder and query { … }.

### Microsoft.FSharp.Linq.RuntimeHelpers. LeafExpressionConverter.QuotationToLambdaExpression

Calls to this are implicitly inserted when calling a method that expects an Expression<T> argument.

Type variable T is assumed to be a delegate type.

This is a strongly typed translation.

### Microsoft.FSharp.Linq.RuntimeHelpers.LeafExpressionConverter.EvaluateQuotation

Calls to this method are made implicitly by the implementation of QueryBuilder and query { … }.

## The Translation

The translation cases are as follows:

|  |  |
| --- | --- |
| Quotation Input | LINQ Output |
| Patterns.Var | the corresponding LINQ ParameterExpression in the translation context |
| Patterns.AndAlso/OrElse | Expression.AndAlso/OrElse |
| Patterns.Value | Expression.Constant |
| Patterns.Coerce | Expression.TypeAs, if this is a true “boxing” or “downcast” coercion |
| Patterns.TypeTest | Expression.TypeIs |
| Patterns.FieldGet | Expression.TypeIs |
| Patterns.TupleGet | Expression.Property (perhaps nested) |
| Patterns.PropertyGet | Expression.Property |
| Patterns.NewRecord | Expression.New |
| Patterns.NewArray | Expression.NewArrayInit |
| Patterns.DefaultValue | Expression.New |
| Patterns.NewUnionCase | Expression.Call on the corresponding method returned by PreComputeUnionConstructorInfo. |
| Patterns.UnionCaseTest | Expression.Equal on the integers returned by the Tag property on the union case and by calling the member returned by PreComputeUnionTagMemberInfo |
| Patterns.NewObject | Expression.New. If the type is System.Nullable, then Expression.Convert. |
| Patterns.NewDelegate | Expression.Lambda. The variables are translated to fresh LINQ parameters and added to the translation context. |
| Patterns.NewTuple | Expression.New (if necessary nested) |
| Patterns.IfThenElse | Expression.Condition |
| Patterns.Quote | A call to the Subst function with values and variables corresponding to the set of variables in the current translation context. |
| Patterns.Let | Expression.Call(“Invoke”, Expression.Lambda(System.Func<\_,\_>, value). The variable is translated to a fresh LINQ parameters and added to the translation context. |
| Patterns.Lambda | Expression.Call(“FuncConvertToFSharpFunc”, Expression.Lambda(typeof<Converter<Domain,Range>>,…) |
| Patterns.Application   * A quotation corresponding to “f x1 x2”, “f x1 x2 x3” or “f x1 x2 x3 x4” | Expression.Call on the static InvokeFast member for the corresponding function type accepting 2,3 or 4 arguments. |
| * A quotation corresponding to “f x1” | Expression.Call on the Invoke member for the corresponding function type. |
| Patterns.Call | Expression.Call, except as noted below |
| * A use of F# core operators <, <=, <>, >=, >, not | Expression.Equal/Greater/GreaterThan/GreaterThanOrEqual/LessThan/LessThanOrEqual/NotEqual/Not |
| * A use of nullable operators ?<, ?<=, ?=, ?>, ?>=, ?<>, <?, <=?, =?, >?, >=?, <>?, ?<?, ?<=?, ?=?, ?>?, ?>=?, ?<>? | Expression.Equal/Greater/GreaterThan/GreaterThanOrEqual/LessThan/LessThanOrEqual/NotEqual/Not, with coercions added as needed |
| * A use of LanguagePrimitives.IntrinsicFunctions.MakeDecimal on constant inputs | Expression.Constant (This corresponds to the quotation form of decimal constant literals.) |
| * A use of F# core operators ~-, +, /, -, \*, %, <<, >>, &&&, |||, ^^^, ~~~ | Expression.Negate/Add/Divide/Subtract/Multiply/Modulo/LeftShift/RightShift/And/Or/ExclusiveOr/Not |
| * A use of F# core checked operators ~-, +, -, \* | Expression.NegateChecked/AddChecked/ SubtractChecked/MultiplyChecked |
| * A use of F# core operators char, decimal, float, float32, sbyte, int16, int32, int, int64, byte, uint16, uint32, uint64 | Expression.Convert |
| * A use of F# core checked operators for the same | Expression.ConvertChecked |
| * A use of MemberInitialization helper | Expression.MemberInit |
| * A use of ImplicitExpressionConversionHelper | The translation of the argument |
| * A use of LanguagePrimitives.IntrinsicFunctions.GetArray | Expression.ArrayIndex |

The following C# LINQ expression trees do not have a translation from F# quotations:

|  |  |
| --- | --- |
| F# construct | C# construct |
| *None* | Expression.ArrayLength |
| *None* | Expression.ListBind |
| *None* | Expression.ListInit |
| *None* | Expression.ElementInit |

## Helper functions that indicate specific translations when used in input quotations

Two additional “fake” values are exposed in the public surface area of this API. When used in a quotation, these functions indicate a specific conversion should be performed when converting the quotation to a LINQ expression.

namespace Microsoft.FSharp.Linq.RuntimeHelpers

module LeafExpressionConverter =

/// When used in a quotation, this function indicates a specific conversion

/// should be performed when converting the quotation to a LINQ expression.

val ImplicitExpressionConversionHelper : 'T -> Expression<'T>

/// When used in a quotation, this function indicates a specific conversion

/// should be performed when converting the quotation to a LINQ expression.

val MemberInitializationHelper : 'T -> 'T

## Helper function for nested quotation literals

A helper function is used to evaluate nested quotation literals in the variable context where they occur.

/// A runtime helper used to evaluate nested quotation literals.

val SubstHelper : Expr \* Var[] \* obj[] -> Expr<'T>

# Performance for Queries

## Client-side CPU costs of executing queries

Apart from “query correctness” (e.g. good SQL), the main performance concern is CPU costs on the client side (and perhaps memory and garbage costs too, though these should reveal themselves as CPU costs).

The F# client-side CPU costs for query evaluation will be higher than C#/VB:

* F# translates from quotations to LINQ expressions, rather than building LINQ expressions directly.
* Evaluating quotation literals <@ … @> (or their implicit form in query { … }) in F# is already slower than constructing LINQ expressions in C# in the first place.
* F# converts “whole” query { … } expressions to quotations, rather than just the leaf fragments as in C#.

We don’t yet know how much damage this will do collectively: I could imagine 10x or more.

As a result, a performance goal of 5x of the CPU cost of equivalent queries in C#/VB seems reasonable.

Ideally we would be closer, but we’re not sure it’s realistic. An important thing to remember is that CPU costs aren’t the most important thing – the real point of queries is to do work on the database, and that will still be achieved. That said, we have heard anecdotally that C# take client side CPU costs of IQueryable query evaluation very seriously, it would be nice to get more info on what customer feedback is driving this. However the comparison point for C#/VB will be with Java, whereas for F# the comparison point should ideally be with OCaml/Haskell/Scala/Clojure, the first three of which at least have no LINQ-like feature at all.

If we are way off our perf goals I think there will be ways for us to improve:

Two small things to consider here:

* We could amortize these static array creations:

IL\_0072: ldc.i4 0xa96

IL\_0077: newarr [mscorlib]System.Byte

IL\_007c: dup

IL\_007d: ldtoken field valuetype '<PrivateImplementationDetails$test>'/T13731\_2710Bytes@ Test/MiscTestsForImplicitExpressionConversion/TechnicalReportExamplesOption2/PageRank2::field13732@

IL\_0082: call void [mscorlib]System.Runtime.CompilerServices.RuntimeHelpers::InitializeArray(class [mscorlib]System.Array,

valuetype [mscorlib]System.RuntimeFieldHandle)

* We could take an array as a parameter here, rather than a list:

Microsoft.FSharp.Quotations.FSharpExpr::Deserialize(class [mscorlib]System.Type, LIST, LIST)

## Size of FSharp.Core

The addition of functionality to FSharp.Core has increased its size from 938K to 1100K. This is within acceptable limits.

# Syntax Highlighting, IntelliSense, Diagnostics and Debugging for Queries

## Syntax Highlighting

All query custom operations must be syntax highlighted within query fragments in “normal” circumstances

This will be implemented as a VSIX extension shipped out-of-band.

## IntelliSense

Character-by-character typing of queries must give reliable intellisense results at all steps of the all the queries shown in the primary example queries in this document.

* Ensure completion of custom operations in queries

## QuickInfo

* Add quick info for custom operations
* Add bespoke quick info text for join and groupJoin

## Diagnostics

Special compiler diagnostics are needed for:

* “where x = y” in a query without parentheses:

## Debugging

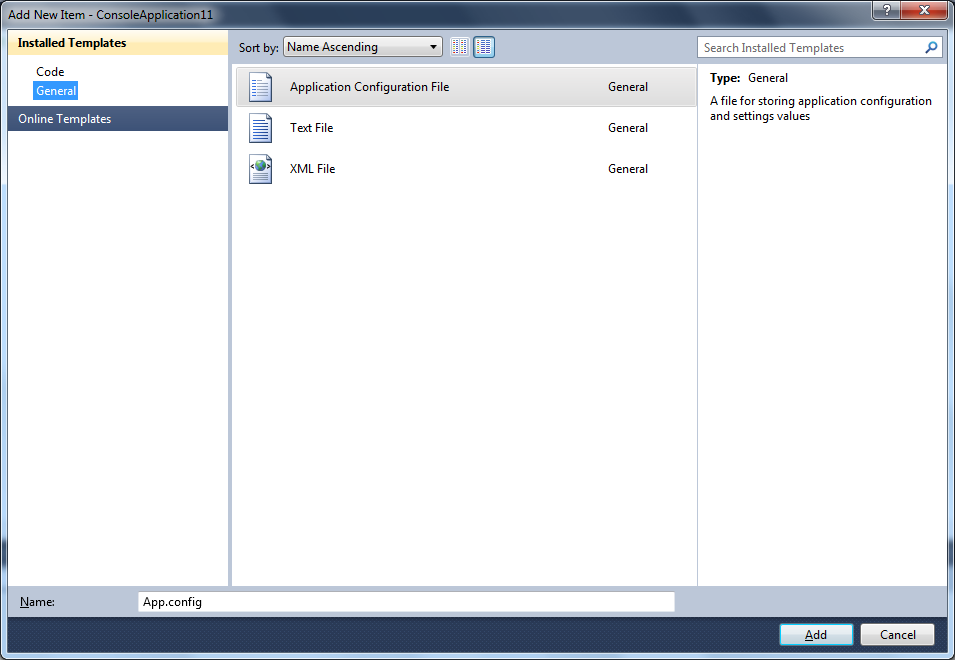
Step-based debugging and breakpoints in queries will not be a major focus, but we should lock down whatever behavior we achieve, e.g. via micro code-generation tests.

User Education material must highlight the end-to-end debugging switched and options that can be provided to help debug LINQ queries, e.g.

* The Log option for Linq-to-SQL databases, which displays the SQL queries executed.
* We should determine other options as appropriate

# Additions to Standard F# Project and Item Templates

We will include templates for each built-in type provider



The following additions will be made to the standard F# project and item templates as part of this work

## RESX Item Template

This is as for C#.

The item is **not** automatically added to the generated assembly – a “ResxFile<…>” reference is needed first.

## DBML Item Template

This is as for C#.

The item is **not** automatically added to the generated assembly – a “DbmlFile<…>” reference is needed first.

## EDMX Item Template

This is as for C#. In C# it triggers a launch of a designer to connect to a database etc. We have to consider what we want here for F#, if anything.

The item is **not** automatically added to the generated assembly – a “EdmxFile<…>” reference is needed first.

## F# Data Consumption Project (SQL, Entity Framework or ADO.NET)

This is under consideration. The template would include the various options and allow you to comment out the ones you did not need. It would initially be configured to produce an .EXE, and have an associated script.

## F# Service Consumption Project (OData or WSDL)

This is under consideration. The template would include the various options and allow you to comment out the ones you did not need. It would initially be configured to produce an .EXE, and have an associated script.

# Out of Band Work

## Query Keyword Highlighting

## F# Power Pack gets EasyChart library

Charting is an essential addition to the F# data programming experience. We should strongly consider adding an EasyChart-like library to the F# power pack. Ideally this should have been part of our dev11 deliverable – it would have made a very “complete” package.

## Ship Additional Providers in F# Power Pack

### Performance Counter Provider

This is a user suggestion.

This could be part of a WMI provider, and the information may actually be available via WMI.

### Registry Provider

This is a team suggestion.

### XmlFile<…> and XsdFile<…>

An XmlFile provider would embed a set of generated types to one specific XML file according to a deduced schema. The XsdFile provider would do the same for any XML file following the given schema. In both cases the types would be the same as those generated by **xsd.exe**.

However, the models produces by xsd.exe aren’t particularly compelling for F# usage because they use so many type tests and type cases for different node types. It’s not clear we should rely on them and it would be wrong to bake in support for them at this stage.

[<Generate>] type SampleXmlDocFile = XmlFile<"Fsc.xml">

SampleXmlDocFile.doc ()

SampleXmlDocFile.docAssembly().name

[<Generate>] type SampleXsdSchema = XsdFile<"wix.xsd">

### Microsoft.Management.TypeProvider

The .NET libraries include a tool to do codegen for Microsoft.Management classes based on WMI information. It is tempting to include a generative provider that uses this tool under-the-hood.

The F# team have also done a prototype of a codegen-free WMI implementation. Below is some sample code. However, the generated code is not as complete w.r.t. required functionality as the generated implementation.

open Microsoft.Management.TypeProvider

[ for b in LocalMachine.Win32\_Battery -> b.Name, b.TimeToFullCharge]

[ for b in LocalMachine.Win32\_DiskDrive -> b.Name, b.Description]

[ for b in LocalMachine.CIM\_Battery -> b.Name, b.BatteryStatus, b.Availability, b.Description, b.ExpectedLife, b.TimeOnBattery, b.SmartBatteryVersion ]

### XamlFile

The F# team have done a prototype of a Xaml provider. It would be nice to include this for completeness, but equally it could be a sample.

open FSharp.Xaml.TypeProvider

type MainWindow = XamlFile<"MainWindow.xaml">

let mainWindow = MainWindow(Visibility=Visibility.Visible, Topmost=true)

let button = mainWindow.thebutton

## Other Major Uses of LINQ – DryadLINQ

DryadLINQ is both an important system in its own right (e.g. for HPC server) and an indicator of a class major potential future uses of LINQ and query-based programming. We expect to cooperate with the DryadLINQ team to ship support for using DryadLINQ with F#.

A set of good DryadLINQ query examples are in the MSR Academic release of DryadLINQ. (Do not use the CTP version from the TC team, which is being thrown away and rewritten starting with the MSR Academic release)

DryadLINQ queries tend to use combinatory form more than query syntax. For example, in C#:

var words = input.SelectMany(x => x.Split(' '));

var groups = words.GroupBy(x => x);

var counts = groups.Select(x => new Pair(x.Key, x.Count()));

var ordered = counts.OrderByDescending(x => x.Count);

var top = ordered.Take(k);

This is a serious challenge for the current F# design – there is no good query { … } encoding of this functionality – all look pretty bad.

For this example, the natural thing for the F# programmer is to write a combinator pipeline:

input

|> QuerySeq.selectMany (fun x -> x.Split(' '))

|> QuerySeq.groupBy (fun x -> x)

|> QuerySeq.select (fun x -> x.Key, x.Count())

|> QuerySeq.sortByDescending (fun (key,count) -> count)

|> QuerySeq.take k

Where there is implicit lambda 🡪 Linq Expression lifting for all hypothetical “QuerySeq” operators, or to write as in C#:

let words = input.SelectMany(fun x -> x.Split(' '));

let groups = words.GroupBy(fun x -> x);

let counts = groups.Select(fun x -> x.Key, x.Count());

let ordered = counts.OrderByDescending(fun (key,count) -> count);

let top = ordered.Take(k);

DryadLINQ also allows the use of **Associative** and **Homomorphic** attributes on called methods. This may again cause problems for F#.

DryadLINQ makes several extensions to the LINQ operators, e.g.

*First, we have to use special* ***DryadLINQ*** *extensions for Aggregate and Count which return IQueryables and not values: AggregateAsQuery and CountAsQuery. These two operators return an IQueryable which will always contain a single element when evaluated.*

Note we would expect this API to only be used via extension-member notation with F#, so these should not pose a problem.

# Risks

* **People will find the dissonance between Seq.\*, “seq { .. }” and “query { … }” too strong** 
  + e.g. “where” v. “filter”, “select” v. “map”
  + F# users can always define additional custom operations as macros, e.g.

module QueryOperators =

[<ReflectedDefinition; MaintainsVariableSpace>]

let select ([<ProjectionParameter>] f) x = QueryOperators.map f x

* **The query translation from F# quotations to LINQ expressions must be correct**
  + This has to be tested carefully
* **Lack of templates**
  + F# 2.0 has a striking lack of templates. Hence the inclusion of adding templates as part of this work
* **F# learning material (books etc.) may not be updated for F# 3.0** 
  + This may be a real problem.

# Resolved Issues

### Limiting Nested “For”

One of the primary “practical problems” with LINQ queries is that it makes it far too easy to construct very expensive cross-join operations by using two nested for loops. Furthermore, nested for-loops are not allowed for many query data sources, e.g. OData.

It would be nice if the fundamental language mechanism had some way to limit the use of nested for loops, even if the default “query” operator did not support this.

The natural way to do this is to translate a single “for” to a form that uses some specific builder method like builder.Select or builder.Map or builder.ForFinal.

**This is not supported.**

### OfType

Is this operator surfaced in F# queries? E.g.

query { for c in db.Customers do

match c with

| :? SpecialCustomer as sc -> ....

| \_ -> () }

This operator would be useful in a Freebase/ontology provider. **This is not currently supported.**

## No Nullable arithmetic operators

We will not have “+”, “-“ etc. on nullable values

## No Nullable default conversions

We will not have “int”, “int64” etc. work on nullable values

## No Nullable user-defined conversions

We will not have anything that helps with “nullable” and user-defined op\_Implicit/op\_Explicit conversions.

## No syntax for array/list

Should query syntax include “query [ …]” and “query [| … |]”? No.

## No Seq operators

We considered adding operators corresponding to Query operators to **Microsoft.FSharp.Collections.Seq**. where they don’t already exist. This does not seem appropriate beyond “where” and “select”

## No Concat operator

This operator will not be surfaced in F# query syntax beyond the existing seq { … } support for in-memory concatenation.

## No query operators for array/list

We do not include these query operators:

toArray: source:seq<'T> -> 'T[]

toList: source:seq<'T> -> 'T list

## No Direct Emit to LINQ Expression Trees

We will not directly generate LINQ Expressions directly from the F# compiler at all.

* This avoids massively complicating the F# language specification, compiler and codegen.

## No Direct Translation to LINQ Query Pattern

Shouldn’t we just emit the LINQ code gen directly and not go through Quotations?

One motivation for this is that reflection support is limited on the Compact Framework and Phone, and quotations expose a couple of extra reflection requirements.

However, we will do not do LINQ-pattern translation in the F# compiler. This avoids the F# language becoming yet-another “shell” for C#, and sets expectations correctly for our support for additional C# pattern-based translations

This means a new builder & quotation-to-expression-tree translation implementation is needed for each instance of the LINQ query pattern.

We will not do this in F# 3.0.

# Appendix

## F# 2.0 LINQ support

LINQ support in F# 2.0 ships in F# PowerPack. The typical LINQ queries look like this:

query <@ seq { for c in db.Customers do yield c.Address}

|> Seq.sortBy (fun s -> s.Length)

@>

query <@ seq { for c in db.Customers do

for e in db.Employees do

if c.ContactName = e.LastName then

yield c.ContactName } @>

The cornerstones of LINQ support in F# are the two functions: ToLinqExpression:

ToLinqExpression : Microsoft.FSharp.Quotations.Expr -> Expressions.Expression

which translates F# quotation into Linq expression, and a query function.

The query function is defined thusly:

val query : Quotations.Expr<'T> -> 'T

Typically, 'T is a seq<\_>. The argument to query is a quotation of an expression expressing query in terms of Seq operators and seq computation expressions. query function translates the given expression into calls to Queryable operators, building a new quotation. For example, the first query would translate into:

<@ Queryable.SortBy(Queryable.Select(db.Customers, <|@fun c -> c.Address @|>, <|@ fun s -> s.Length @|>)) @>

Where <|@ . @|> is an operator translating F# quotations into Linq expressions.

query then translates the resulting quotation into a Linq expression using ToLinqExpression and compiles it.

## Problems F# 2.0 Query Support

F# 2.0 includes reasonable support for IQueryable LINQ queries using a combination of F# programming and the F# power pack. The support is effective and vaguely palatable, but definitely not “syntactically fluent”, because:

* Aggregation operations like Sum and Average must be performed outside the query comprehension syntax. This is also true in C#, where .Sum() or .Max() is used, but the problem tends to be worse in F#.
* Transformation operations like SortBy must be performed outside the query comprehension syntax.
* “Group into” and “Join” operations must be performed outside the query comprehension syntax.
* Queries must be enclosed in both query <@ … @>  and seq { … }, which gives syntactic dissonance, in particular when translating from C#
* Tuples can’t be used as intermediate collection results in queries.
* Queries are not supported in the F# standard library, rather in the power pack

Further, no effective support exists for pattern-based manifestations of the LINQ pattern in F# 2.0.

This approach suffers from a number of drawbacks:

* It’s in the PowerPack, not in FSharp.Core
* Explicit quotations are awkward: users need to master a concept extraneous to their task.
* No “group by”, “sort by” or “join” in the comprehension syntax
* Lack of type safety #1: the approach only statically checks that the query is IEnumerable (indeed it doesn’t even check that). It doesn’t check that the query is IQueryable.
* The syntax is more syntactically distant from SQL (in comparison to C#). This can be a conceptual barrier for people.
* Lack of type safety #2: it translates any quotation of type ‘T into a query. Therefore users are left to discover what is translatable and what is not by trial and error.[[18]](#footnote-18)
* The fact that the syntax is “just like a program” leads people to think they can write anything in the queries (a strange effect: the more uniform the syntax, the more likely they are to hit the lack of type safety inherent in meta-programming-based approaches to queries)
* Nullables are very awkward (both in queries and out)

For example, consider sorting

“Sort by” can be applied as a combinator:

query <@ seq { for c in db.Customers do yield c.Address }

|> Seq.sortBy (fun s -> s.Length) @>

However in general this requires packing the tuple space, sorting, then unpacking…. Even with a query { … } syntax the general pattern would have to be:

query { for ...[v1,...vN]... in yield [v1....vN] }

|> Query.sortBy (fun [v1...vN] -> s.Length)

|> Query.map (fun [v1...vN] -> ...)

let periodicTable =

seq { for elem in Data.Freebase.Chemistry.``Chemical Element`` do

if elem.``Atomic number``.HasValue then

yield elem }

|> Seq.distinctBy (fun e -> e.``Atomic number``.GetValueOrDefault() )

|> Seq.sortBy (fun e -> e.``Atomic number``.GetValueOrDefault() )

The benefits of the approach are

* Fewer dependencies:
  + No dependency on IQueryable in the F# library
  + No dependency on .NET 3.5 in the library/language
* Simpler language definition (keeping “.NET” at arm’s length in the name of hygiene):
  + no special language rules for IQueryable
  + no special syntax for join, group by, sort by
  + no special rules for nullable

The correspondence with C# syntax is as follows:

query { // from

for c in db.Customers do

// from

for e in db.Employees do

// where

if c.ContactName = e.LastName then

// select

yield c.ContactName }

Freebase (and presumably other type-based semantic stores) force the issue of “type discovery” in queries. For example:

query { for loc in Freebase.Locations do

match loc with

| :? LocationWithNeighbourhoods as nlh ->

yield loc, nlh.Neighbourhoods

| \_ ->

() }

There may be more issues lurking here, we should implement Freebase query support….

## Additional LINQ implementations to investigate

e.g. see <http://en.wikipedia.org/wiki/LINQ>

* Continuous LINQ
* LINQ To Sharepoint
* LINQ To Active Directory
* LINQ for Novell.Directory.Ldap
* LinqToWikipedia
* Obtics (Observable Object LINQ)
* LINQ to SNMP
* LINQ to DataSets
* The LINQ to SQL provider works only with Microsoft SQL Server databases; to support any generic database, LINQ also includes the LINQ to DataSets, which uses ADO.NET to handle the communication with the database. Once the data is in ADO.NET Datasets, LINQ to DataSets execute queries against these datasets.
* Data Services: LINQ to ADO.NET Data Services
* dotConnect: LINQ to Oracle, MySQL, PostgreSQL, and SQLite
* Entity Framework: LINQ to Entities
* Windows Search: LINQ to System Search
* Google search: LINQ to Google
* DbLinq: LINQ to MySQL, PostgreSQL, Oracle, Ingres, SQLite and Microsoft SQL Server
* NHibernate:
  + LINQ to NHibernate contribution project
  + Native LINQ support in the upcoming 3.0 release based on re-linq
* DataObjects.NET: LINQ to DataObjects.NET
* LLBLGen Pro: LINQ to LLBLGen
* OpenMapi: LINQ to MAPI
* CSV: LINQ to CSV
* Twitter: LINQ to Twitter
* db4o: LINQ to db4o
* Wikipedia: LINQ to Wikipedia
* LINQ to XSD: LINQ to XML Schema

## Splicing Into query { … }

Is it possible to “splice” into quotation queries? E.g.

let frag() =

<@ QueryOperators.where (fun (c:NorthwndSchema.Customer) -> c.ContactName.StartsWith "B") @>

let customerWithNamesStartingWithB =

query { for c in db.Customers do

%(frag())

select c }

|> showGrid

This typechecks, and we can’t technically stop it happening, but we should test and explore the limits.

## Added Random Notes and Jottings

Binary operators?

| zipWith *inp2* into *y* ; *qexpr*

🡪 zipWith *inp1*@*vs1 inp2* (vs1 y -> qtrans*vs1+vs2* *inp0* *qexpr*)

| binOpWith *inp2* into *y* ; *qexpr*

🡪 binOpWith *inp1*@*vs1 inp2* (vs1 y -> qtrans*vs1+vs2* *inp0* *qexpr*)

| binOpWith (for *p2*vs2in *inp2* -> *key1 = key2*) ; *qexpr*

🡪 binOpWith *inp1*@*vs1 inp2* (*vs1* -> *key1*) (*p2* -> *key2*) (*vs1* *p2* -> qtrans*vs1+vs2* *inp0* *qexpr*)

1. Not just “a LINQ style of programming, the weasel words we use for F# 2.0 [↑](#footnote-ref-1)
2. Remember, SQL still occupies 50% of industry jobs…. Even if F# was relevant to 1% of those, that’s huge number of F# jobs [↑](#footnote-ref-2)
3. This is like groupJoin but implicitly adds “let prodGroup = System.Linq.Enumerable.DefaultIfEmpty prodGroup” [↑](#footnote-ref-3)
4. The thenBy operators are not statically checked to follow a “sortBy” operator. However, a runtime error occurs if this is not the case. [↑](#footnote-ref-4)
5. Head, last, single, minBy, maxBy, averageBy raise InvalidOperationException when used on empty result sets [↑](#footnote-ref-5)
6. **maxBy** in a query returns the maximum value, but the existing **Seq.maxBy** returns an element associated with the maximum value. We considered using **maxValBy** for the former, but on review decided that the extra confusion this would add would not be worth the cost, especially since the types tell the story. [↑](#footnote-ref-6)
7. **yield** is an alternative to select to achieve terminological consistency with seq { ... }. **select** does look much nicer to the LINQ user, but allowing consistency and smooth transitions with seq { ... } is also important to the overall F# coding experience. [↑](#footnote-ref-7)
8. Inside query { …}, skip and take are “forgiving”, they do not raise exceptions if the list runs out of elements. Like other operators inside query { … }, this follows LINQ semantics. [↑](#footnote-ref-8)
9. groupBy has a different signature in query { … }, it returns a sequence of IGrouping, rather than a sequence of tuples. We’ll live with this dissonance. [↑](#footnote-ref-9)
10. These are not included in seq { } and Seq.\* since they are very specific to enable representation of certain LINQ queries, and are not typical in normal F# in-memory programming. [↑](#footnote-ref-10)
11. These are not included in seq { } and Seq.\* since they are very specific to enable representation of certain LINQ queries, and are not typical in normal F# in-memory programming. [↑](#footnote-ref-11)
12. A record type is partially immutable if any of its fields are not marked mutable. [↑](#footnote-ref-12)
13. This second case is for fsi.exe scripting where ConnectionStringConfigFile is used, e.g. consider the case where we have a single config file containing useful connection strings, shared between multiple scripts [↑](#footnote-ref-13)
14. A connection string may be present in a DBML file through the following XML appearing at the top of a tile. This indicates the class name and property name for a .NET variable that holds the connection string setting:

    <Database Name="NORTHWND.MDF" Class="NORTHWND\_MDF" xmlns="http://schemas.microsoft.com/linqtosql/dbml/2007">

    <Connection Provider="System.Data.SqlClient" Mode="AppSettings" SettingsObjectName="…" SettingsPropertyName="…"/>

    ...

    </Database> [↑](#footnote-ref-14)
15. This second case is for fsi.exe scripting where ConnectionStringConfigFile is used, e.g. consider the case where we have a single config file containing useful connection strings, shared between multiple scripts [↑](#footnote-ref-15)
16. As with all generative type providers, static linking is not performed for F# Interactive code. [↑](#footnote-ref-16)
17. This is a frustrating limitation of OData queries which is difficult to work-around from the F# perspective. [↑](#footnote-ref-17)
18. This problem is fairly inherent in meta-programming approaches – C# suffers from it as well. [↑](#footnote-ref-18)