LINQ: Language Features for concurrency (among other things)

Neal M Gafter
Microsoft

LINQ Language Features

- Unified Type System (v1: value and reference)

LINQ Language Features

- Unified Type System (v1: value and reference)

  ```
  struct
  default(T)
  ValueType
  int == System.Int32
  ```

LINQ Language Features

- Unified Type System (v1: value and reference)
- Reified Generics (v2: incl runtime specializ’n)

LINQ Language Features

- Unified Type System (v1: value and reference)

- Reified Generics (v2: incl runtime specializ’n)

  List<int>
  Map<string, int>
  x.F<int>(…)

LINQ Language Features

- Unified Type System (v1: value and reference)
- Reified Generics (v2: incl runtime specializ’n)
- Lambda Expressions (v3)

LINQ Language Features

- Unified Type System (v1: value and reference)
- Reified Generics (v2: incl runtime specializ’n)
- Lambda Expressions (v3)

  (int x, int y) => x+y
  x => x+1

Aggregate Operations

```
IEnumerable<string> list1 = …;
```

Aggregate Operations

```
IEnumerable<string> list1 = …;
IEnumerable<int> list2 =
    Utils.Select( // “map”
        list1,
        (string s) => ParseInt(s));
```

Aggregate Operations

```
IEnumerable<string> list1 = …;
IEnumerable<int> list2 =
    Utils.Select( // “map”
        list1,
        (string s) => ParseInt(s));
int sum1 = Utils.Aggregate( // “reduce”
    list2, 0,
    (int i1, int i2) => i1+i2);
```

Aggregate Operations

```
IEnumerable<string> list1 = …;
IEnumerable<int> list2 =
    Utils.Select( // “map”
        list1,
        (string s) => ParseInt(s));
int sum1 = Utils.Aggregate( // “reduce”
    list2, 0,
    (int i1, int i2) => i1+i2);
int sum2 = Utils.Sum( // map/reduce
    list1,
    (string s) => ParseInt(s));
```

Inferred Locals

```
FilteredStreamReader r =
    new FilteredStreamReader(…);

IEnumerable<string> list2 =
    Utils.Select(list1, (int i) => i.ToString());
```

Inferred Locals

```
FilteredStreamReader r =
    new FilteredStreamReader(…);

IEnumerable<string> list2 =
    Utils.Select(list1, (int i) => i.ToString());
```

Becomes

```
var r = new FilteredStreamReader(…); // locals only
IEnumerable<string> list2 =
    Utils.Select(list1, i => i.ToString());
```

Extension Methods

```
int sum1 = Utils.Aggregate( // map then reduce
    Utils.Select(
        list1,
        s => ParseInt(s)),
        0, (i1,i2) => i1+i2);
```

Extension Methods

```
int sum1 = Utils.Aggregate( // map then reduce
    Utils.Select(
        list1,
        s => ParseInt(s)),
        0, (i1,i2) => i1+i2);
```

becomes

```
int sum1 = list1
    .Select(s => ParseInt(s)) // map
    .Aggregate(0, (i1,i2) => i1+i2); // reduce
```

Lots of useful extensions

```
public static IEnumerable<TSource>
    Distinct<TSource>(this IEnumerable<TSource> source);
public static IEnumerable<TSource>
    Distinct<Tsource>(
        this IEnumerable<TSource> source,
        IEqualityComparer<TSource> comparer);
public static IEnumerable<IGrouping<TKey, TSource>>
    GroupBy<TSource, TKey>(
        this IEnumerable<TSource> source,
        Func<TSource, TKey> keySelector);
public static IEnumerable<TResult>
    Select<TSource, TResult>(
        this IEnumerable<TSource> source,
        Func<TSource, int, TResult> selector);
public static IEnumerable<TResult>
    SelectMany<TSource, TResult>(
        this IEnumerable<TSource> source,
        Func<TSource, IEnumerable<TResult>> selector);
public static IOrderedEnumerable<TSource>
    OrderBy<TSource, TKey>(
        this IEnumerable<TSource> source,
        Func<TSource, TKey> keySelector);
```

“Standard” extensions

By standardizing on a set of extension method signatures, the “same” code can be used to query different “kinds” of data.

- XML
- SQL
- Collections
- Arrays

More pieces of the language puzzle

- Anonymous Types
- Object and Collection Initializers

More pieces of the language puzzle

- Anonymous Types
- Object and Collection Initializers

Eliminate huge swaths of boilerplate

Query Expressions

- Encode particular patterns in the language

```
string[] words =
{ "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese" };

var wordGroups =
    from w in words
    group w by w[0] into g
    select new { FirstLetter = g.Key, Words = g };

foreach (var g in wordGroups) {
    Console.WriteLine(
        "Words that start with the letter '{0}':", g.FirstLetter);
    foreach (var w in g.Words) {
        Console.WriteLine(w);
    }
}
```

Query Expressions

- Encode particular patterns in the language

```
var wordGroups =
    from w in words
    group w by w[0] into g
    select new { FirstLetter = g.Key, Words = g };
```

```
var wordGroups = words
  .GroupBy(w => w[0])
  .Select(g => new { FirstLetter = g.Key, Words = g });
```

Examples

- LINQ to Object
  extension methods for **`IEnumerable`**
  stream-based operations

- LINQ to XML

Examples

- LINQ to Object
  extension methods for **`IEnumerable`**
  stream-based operations

- LINQ to XML

- pLINQ
  extension methods for aggregate operations

pLINQ

```
IEnumerable<T> data = ...;
var q = data
    .Where(x => p(x))
    .Orderby(x => k(x))
    .Select(x => f(x));
foreach (var e in q) a(e);
```

pLINQ

```
IEnumerable<T> data = ...;
var q = data
    .AsParallel()
    .Where(x => p(x))
    .Orderby(x => k(x))
    .Select(x => f(x));
foreach (var e in q) a(e);
```

Problem

This facility is not good enough for

- Relational DBs (LINQ to SQL)
  - Analysis and optimization of full query
  - Remote evaluation

Expression Trees

When a lambda is converted to

```
Expression<Func<…>>
```

Instead of

```
Func<…>
```

The result is a tree representation of the expression!

References

- http://blogs.msdn.com/b/pfxteam/archive/tags/plinq/
- C# Language Section on MSDN, http://tinyurl.com/25o8632
- Visual C# 2010 Express Edition http://tinyurl.com/yaoc3c3
- LINQ samples http://tinyurl.com/62j6sb
- Expression Trees: http://tinyurl.com/ycvbpgz
- Matt Warren’s articles on building query provider APIs http://tinyurl.com/2am65pc

Q&A