Advances in Programming Languages

APL11: Bridging Query and Programming Languages

Ian Stark

School of Informatics
The University of Edinburgh

Monday 16 February 2008
Semester 2 Week 6

Topic: Domain-Specific vs. General-Purpose Languages

This is the second of three lectures on integrating domain-specific languages with general-purpose programming languages. In particular, SQL for database queries.

- Using SQL from Java
- LINQ: .NET Language Integrated Query
- Language integration in F#

Topic: Domain-Specific vs. General-Purpose Languages

This is the second of three lectures on integrating domain-specific languages with general-purpose programming languages. In particular, SQL for database queries.

- Using SQL from Java
- LINQ: .NET Language Integrated Query
- Language integration in F#

Topic: Domain-Specific vs. General-Purpose Languages

This is the second of three lectures on integrating domain-specific languages with general-purpose programming languages. In particular, SQL for database queries.

- Using SQL from Java
- LINQ: .NET Language Integrated Query
  - Overview of Microsoft .NET Framework
  - Integrating queries into C# programming
  - Extensions to the C# language
- Language integration in F#

The Microsoft .NET Framework

Microsoft's *.NET* is a large framework for developing, deploying, and running applications. It now forms a substantial part of the Windows platform, and most additions to Windows arrive as part of .NET.

From the skewed perspective of this course, we can conveniently divide .NET features into two domains:

- Application management infrastructure
- Interesting programming language provision

.NET Application Management

The .NET framework supplies extensive support for building and managing large applications.

- Building:
  - General-purpose base classes: collections, datatypes, text manipulation, networking, crypto, file access, graphics, . . .
  - High-level Windows specials: Forms, Presentation, Communication, Active Directory, Workflow, Cardspace, dots
- Managing:
  - Library control and access
  - Application packaging and deployment
  - Name spaces and versioning

.NET *assemblies* provide rich metadata and other facilities for managing deployment and execution.

.NET Programming Language Support

.NET is comparatively language-neutral, providing a shared platform for multiple programming languages.

The *Common Language Infrastructure* is intended to allow high-level interworking between languages.

- A *Common Language Runtime* (*CLR*) provides memory management, garbage collection, code security and other runtime services.
- The *Common Intermediate Language* (*CIL*, or Microsoft's *MSIL*) is a bytecode that serves as the binary format for .NET components.
- The *Common Type System* (*CTS*) means that applications and libraries written in different languages can sensibly communicate high-level data structures.

MSIL is comparable to the Java virtual machine bytecode, but with a few refinements built in (generics, unboxed datatypes) and better support for different language paradigms.

.NET Programming Languages

Several programming languages are available for .NET, all compiling to MSIL, and all sharing access to the .NET libraries and to each other.

There is good Visual Studio .NET integration for C#, VB.NET (Visual Basic), C/C++, F#, Standard ML, Python and Ruby.

Wikipedia lists another 50 or so .NET languages (right down to LOLcode.net)

For legacy code, and facilities not directly available in the CLR, .NET provides explicit handling of "managed" and "native" code assemblies.

Overall, .NET is similar to Java/JavaEE except for: multiple-language support; symbiotic with Microsoft Windows.

Database Query from Java

```
Connection con = DriverManager.getConnection(url, user, password);

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("SELECT name, id, score FROM Users");

while (rs.next())       // Loop through each row returned by the query
{
  String n  = rs.getString("name");
  int i     = rs.getInt("id");
  float s   = rs.getFloat("score");
  System.out.println(n+i+s);
}
```

Database Query from C#

```
SqlConnection con = new SqlConnection(dataSourceString);

con.Open();

string query = "SELECT name, id, score FROM Users";

SqlCommand command = new SqlCommand(query, con);

SqlDataReader rdr = command.ExecuteReader();

while (rdr.Read())
  { Console.WriteLine("{0} {1} {2}", rdr[0], rdr[1], rdr[2]); }
rdr.Close();
```

Could Do Better

These existing arrangements for database access have good and bad points:

- ✓ Industrial strength: alternative back-end drivers, scalable, supported, familiar.
- ✓ Straightforward: strings are easily to read and edit. (For humans, at least.)

- ✗ Fragile: concatenating and manipulating strings easily goes wrong.
- ✗ Insecure: sanitizing user input becomes essential but also difficult.

- ✗ Unchecked: the strong static checking of Java/C# is abandoned within the query string.
- ✗ Semantically lossy: the high-level abstraction and structure of SQL as a domain-specific declarative programming language is all gone.

Parameterized Queries

Constructions like Java's *prepared statements* can help a little:

```
...
String prequery =
  "SELECT id, name FROM Users WHERE ? < score AND score < ?";

PreparedStatement stmt = con.prepareStatement(prequery);

stmt.setFloat(1,low);    // Fill in the two
stmt.setFloat(2,high);   // missing values

rs = stmt.executeQuery(query); // Now run the completed query
...
```

This is less fragile, and offers opportunities for sanitization: but to go further reinvents features that host programming languages already have.

Could Still Do Better

These existing arrangements for database access have good and bad points:

- ✓ Industrial strength: alternative back-end drivers, scalable, supported, familiar.
- ✓ Straightforward: strings are easily to read and edit. (For humans, at least.)
- ? Fragile: concatenating and manipulating strings easily goes wrong.
- ? Insecure: sanitizing user input becomes essential but also difficult.
- ✗ Unchecked: the strong static checking of Java/C# is abandoned within the query string.
- ✗ Semantically lossy: the high-level abstraction and structure of SQL as a domain-specific declarative programming language is all gone.

LINQ

LINQ, *Language Integrated Query*, aims to improve the alignment between programming languages and query languages.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query = from u in users
              where low < u.Score && u.Score < high
              select new { u.Id, u.Name };

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

LINQ

There is more here than just extra SQL-like keywords. The Table<Person> has typed records, field selection u.Score can be checked at compile time, and each item has a correct static type.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query = from u in users
              where low < u.Score && u.Score < high
              select new { u.Id, u.Name };

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

LINQ

Note also that while **var** query = **from** ... builds a query, here of type IEnumerable<...>, it need not necessarily execute it; this can be deferred until the data itself is required by the **foreach**(...) statement.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query = from u in users
              where low < u.Score && u.Score < high
              select new { u.Id, u.Name };

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

LINQ

The special SQL-like syntax is sugar that expands into a sequence of method invocations, each of which returns an IEnumerable<...> object.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query =  users.Where(u => (low < u.Score && u.Score < high))
                    .Select(u => new { u.Id, u.Name }) ;

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

LINQ

In this case, the Where and Select methods act much like filter and map do on (lazy) lists in a functional language.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query = users.Where(u => (low < u.Score && u.Score < high))
                   .Select(u => new { u.Id, u.Name }) ;

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

LINQ

Although the SQL-like syntax is natural for requesting records from a database, in fact the expansion to regular methods means that it can be used for any kind of IEnumerable<...> objects.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query = users.Where(u => (low < u.Score && u.Score < high))
                   .Select(u => new { u.Id, u.Name }) ;

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

LINQ

This expansion into standard method calls also opens up query handling to compiler optimisation: we are no longer just executing an SQL string, but building a structured query.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query = users.Where(u => (low < u.Score && u.Score < high))
                   .Select(u => new { u.Id, u.Name }) ;

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

Language Support for LINQ

Beyond these small examples, LINQ is a general technique for managing data queries in .NET programming languages: currently supported for {Object, SQL, XML} queries in {C# 3, Visual Basic 9}.

LINQ maps the structure of queries into the host programming language, which allows rich possibilities for manipulation and optimization. However, to do this requires several language extensions, including:

- Lambda expressions
- Free-standing method declarations
- Structural datatypes
- Anonymous record types
- Type inference

These are new to C#, but based on well-established concepts from other existing languages.

Language Support for LINQ

Lambda expressions

Java *inner classes* and C# *delegates* allow for local declaration of methods:

```
int max = 100;
...
Func test = delegate(int id){ return id < max }

... now use test ...
```

A *lambda* expression elides the declaration so that anonymous functions become first-class values:

```
... just use (id => (id<max)) ...
```

Language Support for LINQ

Extension methods

Object-oriented programming allows related classes to implement methods in different ways. With *extension methods*, a third party can add further methods to an existing class.

```
// Extension to String class
public static String Bracket(this String source, String pre, String post)
  { return pre+source+post; }

 ...
s.Bracket( "[", "]" );      // Invokes method Bracket(s, "[", "]" )
```

This is used for Where, Select and other LINQ methods.

Language Support for LINQ

Structural datatypes

Using *data-centric* programming in LINQ means that many classes serve only to hold structured values, without object-style state or behaviour.

To support this a new *object initialization* constructor creates a structured data value with an *anonymous type*:

```
object v = new { title = "OED", volumes = 20, mass = 65.68 };
```

For precise static typing in these cases, a new **var** keyword instructs the compiler to infer an appropriate type from the value provided.

```
var i = 42          // i is an int
var s = "Foo"       // s is a string
var v = new { left = 50, right = 100 } // v has an anonymous type
```

This means that later uses of the object v can be typechecked correctly.

Language Support for LINQ

Metaprogramming

In a final programming technology twist, LINQ to SQL and LINQ to XML pass on full details of how a query was constructed, to help with efficient evaluation. This is in the form of an *expression tree*, which can also include details of C# source code. For example:

```
Expression<Func<int,bool>> test = (id => (id<max));
```

Now test is not an executable function, but a data structure representing the given lambda expression.

LINQ presents the information needed to evaluate a query as an expression tree. By analyzing this, a complex expression combining several query operations might be executed in a single SQL call to the database.

This is a limited form of structured *metaprogramming*, where a program may inspect and work with code itself in a type-safe way.

Homework

Browse the Visual Studio developer documentation. Start here:

- .NET Programming in Visual Studio 2008: Language-Integrated Query (LINQ)
  http://msdn.microsoft.com/library/bb397926.aspx

and be sure to look at one of these:

- C# 3.0 Features That Support LINQ
  http://msdn.microsoft.com/library/bb397909.aspx
- Visual Basic Features That Support LINQ
  http://msdn.microsoft.com/library/bb384991.aspx

Thursday's lecture will be about language integration in F#. If you haven't already programmed in F#, then find out about it. I'll post some links on the course blog.

Summary

- .NET is a large application development framework, with a common virtual machine, type system, and support for interlanguage working.
- LINQ manages queries from within the programming language, not as strings but as first-class entities.
- This uses a number of programming language features new to .NET.
- The integration goes deep: queries are semantic, not syntactic, objects.
- LINQ also introduces first-class expressions, the beginnings of structured reflection and metaprogramming.

Type Safe Linking and Modularity for C

Michael Hicks
University of Maryland

Room 4.31/4.33, Informatics Forum
4pm Tuesday, 17th of February, 2009

Laboratory for Foundations of Computer Science
Seminar Series 2008–2009
http://www.lfcs.ed.ac.uk/seminar