Compiler plugins for LINQ

Miguel Garcia

http://www.sts.tu-harburg.de/people/mi.garcia
Agenda

– The long road to DPBLs
– LINQ: what’s new and what’s not
– Denotational semantics, “the missing link in LINQ”
– A tale of two typesafe rewritings:
  • LINQ nested in Java
  • LINQ nested in Scala
– Future work: Scala (with LINQ) as a DBPL
The long road to DPBLs

- Having the best DBMS and the best programming language is not enough

- We need in fact:
  - a persistence-aware compiler, and
  - a programming-language aware DBMS

- DBPL = control flow + DDL + **query language** + DML

- MS decided a new query language was needed LINQ
LINQ: A query language with …

• A functional-object flavor

• Few “built-in” constructs (join, group by, orderby)

• Most functionality provided by
  – Object-based functions
  – In particular, those on collections: sum, min, max, etc.

• Lean design:
  – Comprehensions and closures
  – Shorthand syntax for grouping and sorting
  – And that’s all, really
Warm-up examples (C# syntax where it makes a difference)

<table>
<thead>
<tr>
<th></th>
<th>LINQ</th>
<th>Math</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>from x1 in e1 from x2 in e2</code></td>
<td>`[ e3</td>
</tr>
<tr>
<td></td>
<td>where cond select e3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scala</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>for (x1 ← e1; x2 ← e2; if cond) yield (e3)</code></td>
<td></td>
</tr>
</tbody>
</table>
# Warm-up examples (C# syntax where it makes a difference)

<table>
<thead>
<tr>
<th>LINQ</th>
<th>from x1 in e1 from x2 in e2 where cond select e3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math</td>
<td>[ e3</td>
</tr>
<tr>
<td>Scala</td>
<td>for (x1 &lt;- e1; x2 &lt;- e2; if cond) yield (e3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LINQ</th>
<th>from c in customers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>group c.name by c.country;</td>
</tr>
<tr>
<td>Math</td>
<td>[ (country, [c.name</td>
</tr>
<tr>
<td></td>
<td>c.country == country] )</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Scala</td>
<td>val countries = noDup(customers map (_.country))</td>
</tr>
<tr>
<td></td>
<td>for (country &lt;- countries) yield (country,</td>
</tr>
<tr>
<td></td>
<td>customers filter (<em>.country == country) map (</em>.name) )</td>
</tr>
</tbody>
</table>
What we’re focusing on

• Wouldn’t it be great if LINQ snippets could appear nested in Java or Scala code?
  – That’s what this talk is about
  – We’ll see how LINQ is desugared into functional-style building blocks (the “SQO” operators, or if you will, the more fundamental *flat*, *filter*, and *flatMap* operators)

• What this talk does not cover:
  – Technologies related to LINQ, in particular the Entity Framework (responsible for most of what we perceive as “LINQ power”)
  – Query optimization
  – Communication with a DBMS

• But first of all, some good old denotational semantics
Main points about LINQ syntax

Railroad diagram for the textual syntax of LINQ, reproduced from http://www.albahari.com/nutshell/linqsyntax.html
Denotational semantics in 5 minutes

A sequence of binding sets

\[
\begin{align*}
\{ &a = \bullet, b = \bullet, \ldots, \bullet, \bullet, \bullet, \bullet \} \\
\{ &a = \bullet, b = \bullet, \ldots, \bullet, \bullet, \bullet, \bullet \} \\
\{ &a = \bullet, b = \bullet, \ldots, \bullet, \bullet, \bullet, \bullet \}
\end{align*}
\]

Fed into a Select valuation function

\[\mathbf{[select \ E_{selexp}]envs} \overset{\text{def}}{=} \mathbf{[ [selexp](env) | env \leftarrow envs ]}\]

Results in

A result sequence

(no more binding sets)
Other valuations just enrich an environment

\[
\begin{align*}
\{ & a = \bullet, \ b = \bullet, \ \ldots \ \bullet \ \bullet \ \bullet \ \} \\
\{ & a = \bullet, \ b = \bullet, \ \ldots \ \bullet \ \bullet \ \} \\
\{ & a = \bullet, \ b = \bullet, \ \ldots \ \bullet \ \bullet \ \} \\
\{ & a = \bullet, \ b = \bullet, \ \ldots \ \bullet \ \bullet \ \} \\
\end{align*}
\]

fed into a \textit{Let} valuation function

\[
\left[ \text{let } V_{\text{var}} = E_{\text{exp}} \right] \text{envs} \overset{\text{def}}{=} \left[ \text{env'} \mid \text{env} \leftarrow \text{envs}, \right. \left. \text{let env'} = \text{env} \cup \{ \text{var} \mapsto \llbracket \text{exp} \rrbracket (\text{env}) \} \right]
\]

results in

\[
\begin{align*}
\{ & a = \bullet, \ b = \bullet, \ \ldots \ \bullet \ \bullet \ \bullet \ \} \\
\{ & a = \bullet, \ b = \bullet, \ \ldots \ \bullet \ \bullet \ \} \\
\{ & a = \bullet, \ b = \bullet, \ \ldots \ \bullet \ \bullet \ \} \\
\{ & a = \bullet, \ b = \bullet, \ \ldots \ \bullet \ \bullet \ \} \\
\end{align*}
\]

another sequence of binding sets, for consumption by the next LINQ construct

\[
\begin{align*}
\{ & a = \bullet, \ b = \bullet, \ \ldots \ \bullet \ \bullet \ \bullet \ \} \\
\{ & a = \bullet, \ b = \bullet, \ \ldots \ \bullet \ \bullet \ \} \\
\{ & a = \bullet, \ b = \bullet, \ \ldots \ \bullet \ \bullet \ \} \\
\{ & a = \bullet, \ b = \bullet, \ \ldots \ \bullet \ \bullet \ \} \\
\end{align*}
\]
Desugaring LINQ into SQO: the big picture

“LINQ textual syntax”

```csharp
var q = from p in people where p.Age > 20
    orderby p.Age descending, p.Name
    group new {
    } by p.CanCode;
```

↓

```csharp
var q = people.Where(p => p.Age > 20)
    .GroupBy(p => p.CanCode,
        p => new { p.Name, Senior = p.Age > 30,
            FirstJob = p.Jobs[0]
        });
```

“Standard Query Operators” (SQO)
SQO: 50 operators with 94 overloaded signatures (4 shown below)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static IQueryable&lt;IGrouping&lt;TKey, TSource&gt;&gt; GroupBy&lt;TSource,TKey&gt;(this IQueryable&lt;TSource&gt; source, Expression&lt;Func&lt;TSource, TKey&gt;&gt; keySelector)</td>
<td>Groups the elements of a sequence according to a specified key selector function.</td>
</tr>
<tr>
<td>public static IQueryable&lt;IGrouping&lt;TKey, TSource&gt;&gt; GroupBy&lt;TSource,TKey&gt;(this IQueryable&lt;TSource&gt; source, Expression&lt;Func&lt;TSource, TKey&gt;&gt; keySelector, IEqualityComparer&lt;TKey&gt; comparer)</td>
<td>Groups the elements of a sequence according to a specified key selector function and compares the keys by using a specified comparer.</td>
</tr>
<tr>
<td>public static IQueryable&lt;IGrouping&lt;TKey, TElement&gt;&gt; GroupBy&lt;TSource,TKey,TElement&gt;(this IQueryable&lt;TSource&gt; source, Expression&lt;Func&lt;TSource, TKey&gt;&gt; keySelector, Expression&lt;Func&lt;TSource, TElement&gt;&gt; elementSelector)</td>
<td>Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function.</td>
</tr>
<tr>
<td>public static IQueryable&lt;TResult&gt; GroupBy&lt;TSource, TKey, TResult&gt;(this IQueryable&lt;TSource&gt; source, Expression&lt;Func&lt;TSource, TKey&gt;&gt; keySelector, Expression&lt;Func&lt;TKey, IResultEnumerable&lt;TSource&gt;, TResult&gt;&gt; resultSelector)</td>
<td>Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key.</td>
</tr>
</tbody>
</table>

How to pick the correct ones when desugaring from LINQ?
18 transformation rules to the rescue

(details in the paper)

Implemented in LINQExpand4Java, an open-source JSR-269 compiler plugin

<table>
<thead>
<tr>
<th>ID</th>
<th>Phase</th>
<th>Description</th>
<th>§ in C# spec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1</td>
<td>Inline query continuation</td>
<td>7.15.2.1</td>
</tr>
<tr>
<td>T2</td>
<td>2</td>
<td><code>FromClause</code> <code>JoinClause</code> <code>JoinIntoClause</code></td>
<td>7.15.2.2</td>
</tr>
<tr>
<td>T3</td>
<td>3</td>
<td>Identity query</td>
<td>7.15.2.3</td>
</tr>
<tr>
<td>T4</td>
<td>4</td>
<td><code>FromClause</code> <code>FromClause</code> <code>SelectClause</code> otherwise</td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>4</td>
<td><code>FromClause</code> <code>JoinClause</code> <code>SelectClause</code> otherwise</td>
<td></td>
</tr>
<tr>
<td>T6</td>
<td>4</td>
<td><code>FromClause</code> <code>JoinIntoClause</code> <code>SelectClause</code> otherwise</td>
<td>7.15.2.4</td>
</tr>
<tr>
<td>T7</td>
<td>4</td>
<td><code>OrderByClause</code> <code>WhereClause</code> <code>LetClause</code></td>
<td></td>
</tr>
<tr>
<td>T8</td>
<td>4</td>
<td>non-identity <code>SelectClause</code> identity <code>SelectClause</code></td>
<td>7.15.2.5</td>
</tr>
<tr>
<td>T9</td>
<td>4</td>
<td>non-identity <code>GroupByClause</code> identity <code>GroupByClause</code></td>
<td>7.15.2.6</td>
</tr>
</tbody>
</table>
T1: Inline query continuations

(a) \[ \text{T1: inline query continuation} \]
\[
\text{from } x_1 \text{ in } e_1 \ldots \text{ into } x_2 \ldots \\
\rightarrow \text{from } x_2 \text{ in } (\text{from } x_1 \text{ in } e_1 \ldots) \ldots
\]

(b) \[
\text{function T1 ( q : QueryExp ) : QueryExp} \\
\text{when } q.qbody.qcont \neq \emptyset
\]
\[
\text{new QueryExp} \{ \\
\text{from} = \text{new FromClause} \{ \\
\text{type} = q.qbody.qcont.type, \\
\text{var} = q.qbody.qcont.var, \\
\text{in} = \text{new QueryExp} \{ \\
\text{from} = q.from \\
qbody = \text{new QueryBody} \{ \\
\text{qbclauses} = q.qbody.qbclauses, \\
\text{sel_gby} = q.qbody.sel_gby \\
\} \} \}
\]
qbody = q.qbody.qcont.qbody
\]
### How does Scala compare to SQO as translation target?

- **Pros:** Avoids learning the 90+ overloaded SQO operators
- **Cons:** Grouping and sorting (where Scala lacks syntax shorthands) result in less-than-compact expressions

<table>
<thead>
<tr>
<th>T10</th>
<th>From followed by a join with into followed by a select</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LINQ</strong></td>
<td>from x1 in e1</td>
</tr>
<tr>
<td></td>
<td>join x2 in e2 on k1 equals k2 into g</td>
</tr>
<tr>
<td></td>
<td>select e3</td>
</tr>
<tr>
<td><strong>SQO</strong></td>
<td>e1.GroupJoin( e2,</td>
</tr>
<tr>
<td></td>
<td>x1 =&gt; k1, x2 =&gt; k2,</td>
</tr>
<tr>
<td></td>
<td>(x1, g) =&gt; e3</td>
</tr>
<tr>
<td><strong>Scala</strong></td>
<td>for ( x1 &lt;- e1; val outerKey = k1;</td>
</tr>
<tr>
<td></td>
<td>val g = e2 filter { x2 =&gt; outerKey == k2 }</td>
</tr>
<tr>
<td></td>
<td>) yield e3</td>
</tr>
</tbody>
</table>
More pros

- The Scala compiler performs its own desugaring
- Additional processing (e.g., optimization) possible via flexible compiler plugin architecture (better than Java’s)

<table>
<thead>
<tr>
<th>Scala</th>
<th>val res = for (x1 &lt;- e1; x2 &lt;- e2) yield (x1 + x2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desugar</td>
<td>val res: Array[Double] = e1flatMap[Double] (((x1: Double) =&gt; e2.map[Double] (((x2: Double) =&gt; x1.+(x2)))));</td>
</tr>
</tbody>
</table>

- AST rewriting can rely on the Kiama library
  - http://code.google.com/p/kiama
- Allows expressing in Scala two program-transformation paradigms (in a manner faithful to their original notations):
  - strategic term rewriting, i.e. Stratego
  - (circular) attribute grammars, i.e. JastAdd
... and a few (transient) cons too

- As of Scala 2.7.5, no partial evaluation performed by default
  - Either implement it yourself, or
  - Try compiling with -optimise, -specialize, -experimental, or
  - Wait for Scala 2.8

<table>
<thead>
<tr>
<th>Scala</th>
<th>val res2 = Set(1, 2, 3) map (x =&gt; x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desugar</td>
<td>Not reduced to « Set(1, 2, 3) »</td>
</tr>
<tr>
<td></td>
<td>The identity function will be applied at runtime, somewhere in the snippet below J</td>
</tr>
</tbody>
</table>

```scala
def desugar: scala.collection.immutable.Set = 
scala.this.Predef.Set().apply(
new scala.runtime.BoxedIntArray(
Array[Int]{1,2,3})
).map(
{(new QueryDesugar$$anonfun$2(): Function1)}))
```
Going the last mile to make Scala a DBPL

• Our first prototype:
  – Typesafe rewriting of nested LINQ into (main-memory only) Scala

• Further down the road
  – Deciding on a rich subset of Scala comprehensions for translation into a DB query language
  – Using a comprehensions-aware query planner:
    • lambda-DB, http://lambda.uta.edu/lambda-DB/manual/
    • Ferry & Pathfinder, http://www-db.informatik.uni-tuebingen.de/research/ferry

• And the laundry list goes on
  – DML, caching, events, materialized views, parallelism
Conclusions

• A “proof of concept” DPBL can be grown one component at a time (working hypothesis)
  – Some sub-problems have great solutions done by others

• Be wary of any new query language lacking denotational semantics

• Scala is here to stay, also in the field of OODBs
  – At least, for as long as LINQ remains relevant in the field of OODBs