FSharpComposableQuery
overview & demo

James Cheney
(with Phil Wadler, Sam Lindley, Yordan Stoyanov)
University of Edinburgh

F#unctional Programming Meetup
September 11, 2014

Work supported by Royal Society, UoE, Google and EPSRC
Motivation

- Database programming involves generating query "code" (SQL) at run time
- Naive approach: compose SQL as strings
  - Maximal control, performance tuning
- But:
  - Type-unsafe
  - can lead to security vulnerabilities (SQL injection)
LINQ

- Language-Integrated Query (LINQ)
  - Microsoft C# (Meijer et al. 2006)
  - and F# (Syme 2006)
- Based on comprehension syntax (a.k.a. "do" notation, computation expressions, etc.)
  - and quotation `<@ @@`
- which explicitly separates query from normal code
- Type-safety inherited from source language
  - Type providers (run-time type information in IDE) make this especially handy
**LINQ (F#) example**

![Table of employees](image)

<table>
<thead>
<tr>
<th>dpt</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Product&quot;</td>
<td>Alex</td>
<td>40,000</td>
</tr>
<tr>
<td>&quot;Product&quot;</td>
<td>Bert</td>
<td>60,000</td>
</tr>
<tr>
<td>&quot;Research&quot;</td>
<td>Cora</td>
<td>50,000</td>
</tr>
<tr>
<td>&quot;Research&quot;</td>
<td>Drew</td>
<td>70,000</td>
</tr>
<tr>
<td>&quot;Sales&quot;</td>
<td>Drew</td>
<td>200,000</td>
</tr>
<tr>
<td>&quot;Sales&quot;</td>
<td>Erik</td>
<td>95,000</td>
</tr>
<tr>
<td>&quot;Sales&quot;</td>
<td>Gina</td>
<td>155,000</td>
</tr>
</tbody>
</table>

**Tasks**

<table>
<thead>
<tr>
<th>emp</th>
<th>tsk</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alex&quot;</td>
<td>&quot;build&quot;</td>
</tr>
<tr>
<td>&quot;Bert&quot;</td>
<td>&quot;build&quot;</td>
</tr>
<tr>
<td>&quot;Cora&quot;</td>
<td>&quot;abstract&quot;</td>
</tr>
<tr>
<td>&quot;Cora&quot;</td>
<td>&quot;call&quot;</td>
</tr>
<tr>
<td>&quot;Cora&quot;</td>
<td>&quot;dissemble&quot;</td>
</tr>
<tr>
<td>&quot;Cora&quot;</td>
<td>&quot;enthusue&quot;</td>
</tr>
<tr>
<td>&quot;Drew&quot;</td>
<td>&quot;abstract&quot;</td>
</tr>
<tr>
<td>&quot;Drew&quot;</td>
<td>&quot;enthusue&quot;</td>
</tr>
<tr>
<td>&quot;Erik&quot;</td>
<td>&quot;call&quot;</td>
</tr>
<tr>
<td>&quot;Erik&quot;</td>
<td>&quot;enthusue&quot;</td>
</tr>
</tbody>
</table>

**Query**

```fsharp
define query 
  for x in employees 
  where (x.salary > 50000) 
  yield {name=x.name} 
```
LINQ (F#) example

query { for x in employees
   where (x.salary > 50000)
yield {name=x.name} }

select name
from employees e
where e.salary > 50000
LINQ (F#) example

```
query { for x in employees
    where (x.salary > 50000)
    yield {name=x.name} }
```
Dynamic/composable queries in F#?

Dynamic SQL queries with F# 3.0?

I have tried to use FLINQ but it is rather out of date with F# 3.0 beta.
Can someone give me some pointers on how to create dynamic SQL queries in F#?

3

sql | #
Dynamic/composable queries in F#?

**How do you compose query expressions in F#?**


And I've been wondering why the following is legitimate:

```fsharp
let testQuery = query {
    for number in netflix.Titles do
    where (number.Name.Contains("Test"))
}
```

But you can't really do something like this:

```fsharp
let christmasPredicate = fun (x:Catalog.ServiceTypes.Title) -> x.Name.Contains("Christmas")
let testQuery = query {
    for number in netflix.Titles do
    where christmasPredicate
}
```

Surely F# allows compositability like this so you can reuse a predicate?? What if I wanted Christmas titles combined with another predicate like before a specific date? I have to copy and paste my entire query? C# is completely unlike this and has several ways to build and combine predicates.
Dynamic/composable queries in F#?

How do you compose query expressions in F#?


And I've been wondering why the following is legitimate

```fsharp
let testQuery = query {
    for number in netflix.Titles do
    where (number.Name.Contains("Test"))
}
```

But you can't really do something like this

```fsharp
let christmasPredicate = fun (x:Catalog.ServiceTypes.Title) -> x.Name.Contains("Christmas")
let testQuery = query {
    for number in netflix.Titles do
    where christmasPredicate
}
```

Surely F# allows composability like this so you can reuse a predicate?? What if I wanted Christmas titles combined with another predicate like before a specific date? I have to copy and paste my entire query? C# is completely unlike this and has several ways to build and combine predicates.
Queries with function "parameters"?

- A way to (de)compose queries into reusable chunks?
  - (avoid repeating yourself)
- This could be very useful
  - a form of staged computation/meta-programming
- Queries could be constructed dynamically
  - including constructing queries of different "shape"
  - goes beyond simple int/string parameters
  - yet still strongly typed
A LINQ example

### Sample Data

<table>
<thead>
<tr>
<th>dpt</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td>Alex</td>
<td>40,000</td>
</tr>
<tr>
<td>Product</td>
<td>Bert</td>
<td>60,000</td>
</tr>
<tr>
<td>Research</td>
<td>Cora</td>
<td>50,000</td>
</tr>
<tr>
<td>Research</td>
<td>Drew</td>
<td>70,000</td>
</tr>
<tr>
<td>Sales</td>
<td>Erik</td>
<td>200,000</td>
</tr>
<tr>
<td>Sales</td>
<td>Fred</td>
<td>95,000</td>
</tr>
<tr>
<td>Sales</td>
<td>Gina</td>
<td>155,000</td>
</tr>
</tbody>
</table>

### Code Example

```csharp
let elem = <@ fun x xs ->
    query { for y in xs
        exists(y = x) } @>

let canDo = <@ fun name tsk ->
    (elem) tsk (for t in tasks
        where (t.emp = name)
        yield t.tsk) @>

query { for x in employees
    where ((canDo name "build")
        yield {name=x.name} } }
```
LINQ example

### Table: Employees

<table>
<thead>
<tr>
<th>dpt</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Product&quot;</td>
<td>&quot;Alex&quot;</td>
<td>40,000</td>
</tr>
<tr>
<td>&quot;Product&quot;</td>
<td>&quot;Bert&quot;</td>
<td>60,000</td>
</tr>
<tr>
<td>&quot;Research&quot;</td>
<td>&quot;Cora&quot;</td>
<td>50,000</td>
</tr>
<tr>
<td>&quot;Research&quot;</td>
<td>&quot;Drew&quot;</td>
<td>70,000</td>
</tr>
<tr>
<td>&quot;Sales&quot;</td>
<td>&quot;Erik&quot;</td>
<td>200,000</td>
</tr>
<tr>
<td>&quot;Sales&quot;</td>
<td>&quot;Fred&quot;</td>
<td>95,000</td>
</tr>
<tr>
<td>&quot;Sales&quot;</td>
<td>&quot;Gina&quot;</td>
<td>155,000</td>
</tr>
</tbody>
</table>

### Tasks

<table>
<thead>
<tr>
<th>emp</th>
<th>tsk</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alex&quot;</td>
<td>&quot;build&quot;</td>
</tr>
<tr>
<td>&quot;Bert&quot;</td>
<td>&quot;build&quot;</td>
</tr>
<tr>
<td>&quot;Cora&quot;</td>
<td>&quot;abstract&quot;</td>
</tr>
<tr>
<td>&quot;Cora&quot;</td>
<td>&quot;build&quot;</td>
</tr>
<tr>
<td>&quot;Cora&quot;</td>
<td>&quot;call&quot;</td>
</tr>
<tr>
<td>&quot;Drew&quot;</td>
<td>&quot;dissemble&quot;</td>
</tr>
<tr>
<td>&quot;Drew&quot;</td>
<td>&quot;enthuse&quot;</td>
</tr>
<tr>
<td>&quot;Erik&quot;</td>
<td>&quot;abstract&quot;</td>
</tr>
<tr>
<td>&quot;Erik&quot;</td>
<td>&quot;enthuse&quot;</td>
</tr>
<tr>
<td>&quot;Erik&quot;</td>
<td>&quot;call&quot;</td>
</tr>
<tr>
<td>&quot;Fred&quot;</td>
<td>&quot;enthuse&quot;</td>
</tr>
<tr>
<td>&quot;Gina&quot;</td>
<td>&quot;call&quot;</td>
</tr>
<tr>
<td>&quot;Gina&quot;</td>
<td>&quot;dissemble&quot;</td>
</tr>
</tbody>
</table>

```fsharp
let elem = <@ fun x xs -> query { for y in xs exists(y = x) } @>
let canDo = <@ fun name tsk -> (%elem) tsk (for t in tasks where (t.emp = name) yield t.tsk) @>
query { for x in employees
    where ((%canDo) x.name "build")
    yield {name=x.name} }
```
Normalization

- Monadic comprehensions (including nonrecursive higher-order functions) can be normalized
  - Worked out by Wong for Kleisli system, extended to higher-order in Links by Cooper
  - Translation to SQL then straightforward
- However (surprisingly), LINQ (F#) doesn't fully support this normalization
  - our ICFP '13 paper shows how to add this
Normalisation: symbolic evaluation

\[(\text{fun}(x) \rightarrow N) \ M \leadsto N[x := M]\]

\[\{\ell = M\}.\ell_i \leadsto M_i\]

\[\text{for } x \text{ in } \text{(yield } M) \text{ do } N \leadsto N[x := M]\]

\[\text{for } y \text{ in } \text{(for } x \text{ in } L \text{ do } M) \text{ do } N \leadsto \text{for } x \text{ in } L \text{ do } (\text{for } y \text{ in } M \text{ do } N)\]

\[\text{for } x \text{ in } \text{(if } L \text{ then } M) \text{ do } N \leadsto \text{if } L \text{ then } (\text{for } x \text{ in } M \text{ do } N)\]

\[\text{for } x \text{ in } [ ] \text{ do } N \leadsto [ ]\]

\[\text{for } x \text{ in } (L @ M) \text{ do } N \leadsto (\text{for } x \text{ in } L \text{ do } N) @ (\text{for } x \text{ in } M \text{ do } N)\]

\[\text{if true then } M \leadsto M\]

\[\text{if false then } M \leadsto [ ]\]

slide stolen from Phil Wadler's talk
Normalisation: *ad hoc* rewriting

\[
\begin{align*}
\text{for } x \text{ in } L \text{ do } (M \oplus N) & \leftrightarrow (\text{for } x \text{ in } L \text{ do } M) \oplus (\text{for } x \text{ in } L \text{ do } N) \\
\text{for } x \text{ in } L \text{ do } \text{[ ]} & \leftrightarrow \text{[ ]} \\
\text{if } L \text{ then } (M \oplus N) & \leftrightarrow (\text{if } L \text{ then } M) \oplus (\text{if } L \text{ then } N) \\
\text{if } L \text{ then } \text{[ ]} & \leftrightarrow \text{[ ]} \\
\text{if } L \text{ then } (\text{for } x \text{ in } M \text{ do } N) & \leftrightarrow \text{for } x \text{ in } M \text{ do } (\text{if } L \text{ then } N) \\
\text{if } L \text{ then } (\text{if } M \text{ then } N) & \leftrightarrow \text{if } (L \land M) \text{ then } N
\end{align*}
\]
let elem = <@ fun x xs ->
    query { for y in xs
        exists (y=x) } @>
let canDo = <@ fun name tsk ->
    (%elem) tsk (for t in tasks
        where (t.emp = name)
        yield t.tsk ) @>
query { for x in employees
    where ((%canDo) x.name "build")
    yield {name = x.name}
let elem = <@ fun x xs ->
    query { for y in xs
        exists (y=x) }
    @>

let canDo = <@ fun name tsk ->
    (fun x xs ->
        query { for y in xs
            exists (y=x) }
        tsk (for t in tasks
            where (t.emp = name)
            yield t.tsk ) @>
    query { for x in employees
        where ((canDo) x.name "build")
        yield {name = x.name}
Example

```ml
let elem = <@ fun x xs ->
  query { for y in xs
    exists (y=x) } @>

let canDo = <@ fun name tsk ->
  query { for x in employees
    where ((%elem) x.name "build")
    yield {name = x.name} } @>

query { for x in employees
  where ((%canDo) x.name "build")
  yield {name = x.name} }
```
Example

This is what LINQ normally sees. Note β-redexes!

X (failure or query avalanche)
Example

```ml
let elem = <@ fun x xs ->
    query { for y in xs
        exists (y=x) } @>

let canDo = <@ fun name tsk ->
    (fun x xs ->
        query { for y in xs
            exists (y=x) })
    tsk (for t in tasks
        where (t.emp = name)
        yield t.tsk ) @>

query { for x in employees
    where ((fun name tsk ->
        (fun x xs ->
            query { for y in xs
                exists (y=x) }))
        tsk (for t in tasks
            where (t.emp = name)
            yield t.tsk ) ) x.name "build"
    yield {name = x.name}
```
Example

```plaintext
let elem = fun x xs ->
    query { for y in xs
        exists (y=x) }

let canDo = fun name tsk ->
    (elem tsk (for t in tasks
        where (t.emp = name)
        yield t.tsk ))

query { for x in employees
    where ((canDo x.name "build")
    yield {name = x.name}}
```
Example

let elem = <@ fun x xs ->
  query { for y in xs
    exists (y=x) }
@>

let canDo = <@ fun name tsk ->
  (fun x xs ->
    query { for y in xs
      exists (y=x) }
    tsk (for t in tasks
      where (t.emp = name)
      yield t.tsk )
  ) x.name "build"
@>

query { for x in employees
  where ((canDo) x.name "build")
  yield {name = x.name}
Example

```ocaml
class = <
  fun x xs ->
    query { for y in xs
      exists (y=x) }
>

class canDo = <
  fun name tsk ->
    (class elem) tsk (for t in tasks
      where (t.emp = name)
        yield t.tsk
    )
>

query { for x in employees
  where ((class canDo) x.name "build")
    yield {name = x.name}
```

Example

```
let elem = <@ fun x xs ->
  query { for y in xs
    exists (y=x) }
@>

let canDo = <@ fun name tsk ->
  (fun x xs -> query { for y in xs
    exists (y=x) } tsk (for t in tasks
      where (t.emp = name)
      yield t.tsk )
@>

query { for x in employees
  where ((canDo) x.name "build")
  yield {name = x.name}
```
• A library that implements normalization from our ICFP paper

• "No assembly required"
  • Replaces standard QueryBuilder query operator
  • including (subtle) overloading tricks (thanks to Don Syme for helping with this)

• Tested on a wide range of query expressions
  • Should preserve or improve on default behavior
Demo

- Tutorial examples from ICFP paper
Conclusions

- F# 3.0's LINQ capabilities are powerful, but have some (ad hoc?) limitations
  - Quotation and higher-order functions can be used to compose queries
  - But, existing LINQ implementation doesn't always handle these correctly or efficiently
- Normalization techniques developed in other contexts can help
- Presented FSharpComposableQuery
  - a drop-in library that augments F#'s LINQ facilities with better support for query composition and higher-order functions
- http://www.nuget.org/packages/FSharpComposableQuery/