

FSharpComposableQuery overview & demo

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

F#unctional Programming Meetup
September 11, 2014

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

employees

dpt	name	salary
"Product"	"Alex"	40,000
"Product"	"Bert"	60,000
"Research"	"Cora"	50,000
"Research"	"Drew"	70,000
"Sales"	"Erik"	200,000
"Sales"	"Fred"	95,000
"Sales"	"Gina"	155,000

tasks

emp	tsk
"Alex"	"build"
"Bert"	"build"
"Cora"	"abstract"
"Cora"	"build"
"Cora"	"call"
"Cora"	"disassemble"
"Cora"	"enthuse"
"Drew"	"abstract"
"Drew"	"enthuse"
"Erik"	"call"
"Erik"	"enthuse"

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

LINQ (F#) example

employees

dpt	name	salary
"Product"	"Alex"	40,000
"Product"	"Bert"	60,000
"Research"	"Cora"	50,000
"Research"	"Drew"	70,000
"Sales"	"Erik"	200,000
"Sales"	"Fred"	95,000
"Sales"	"Gina"	155,000

tasks

emp	tsk
"Alex"	"build"
"Bert"	"build"
"Cora"	"abstract"
"Cora"	"build"
"Cora"	"call"
"Cora"	"disassemble"
"Cora"	"enthuse"
"Drew"	"abstract"
"Drew"	"enthuse"
"Erik"	"call"
"Erik"	"enthuse"

```
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

employees

dpt	name	salary
"Product"	"Alex"	40,000
"Product"	"Bert"	60,000
"Research"	"Cora"	50,000
"Research"	"Drew"	70,000
"Sales"	"Erik"	200,000
"Sales"	"Fred"	95,000
"Sales"	"Gina"	155,000

tasks

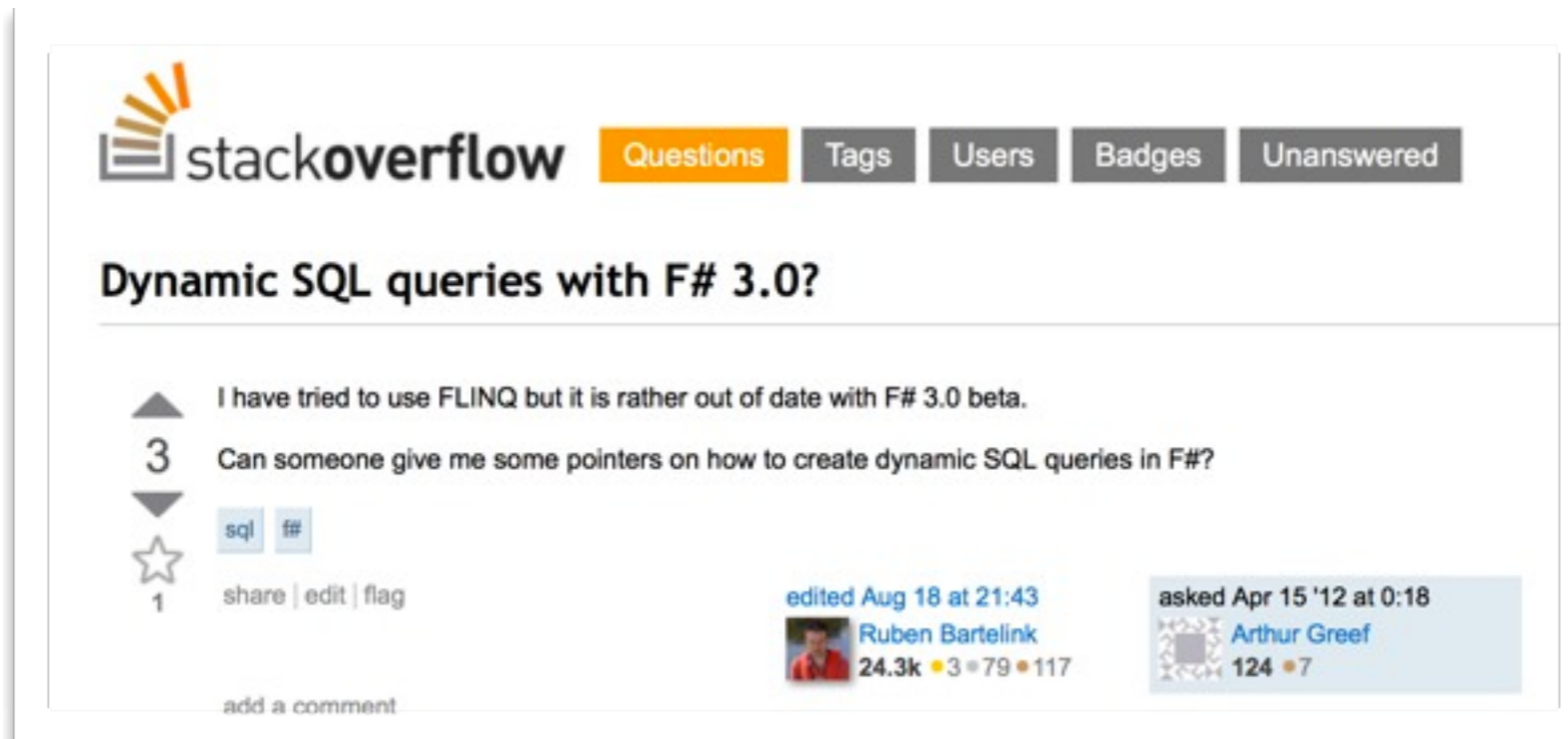
emp	tsk
"Alex"	"build"
"Bert"	"build"
"Cora"	"abstract"
"Cora"	"build"
"Cora"	"call"
"Cora"	"dissemble"
"Cora"	"enthuse"
"Drew"	"abstract"
"Drew"	"enthuse"
"Erik"	"call"
"Erik"	"enthuse"

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

```
select name
from employees e
where e.salary > 50000
```

name
Bert
Drew
Erik
Fred
Gina

Dynamic/composable queries in F#?



The screenshot shows a Stack Overflow question page. At the top, the Stack Overflow logo is on the left, and navigation buttons for 'Questions', 'Tags', 'Users', 'Badges', and 'Unanswered' are on the right. The question title is 'Dynamic SQL queries with F# 3.0?'. The question body contains the text: 'I have tried to use FLINQ but it is rather out of date with F# 3.0 beta.' Below the text, there are icons for upvotes (3), downvotes, and a star (1). There are also tags for 'sql' and 'f#'. At the bottom left, there are links for 'share', 'edit', and 'flag', and a link to 'add a comment'. On the right side, there are two user profiles: 'Ruben Bartelink' (edited Aug 18 at 21:43, 24.3k reputation, 3 answers, 79 votes, 117 comments) and 'Arthur Greef' (asked Apr 15 '12 at 0:18, 124 reputation, 7 votes).

stackoverflow Questions Tags Users Badges Unanswered

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.

3 Can someone give me some pointers on how to create dynamic SQL queries in F#?

sql f#

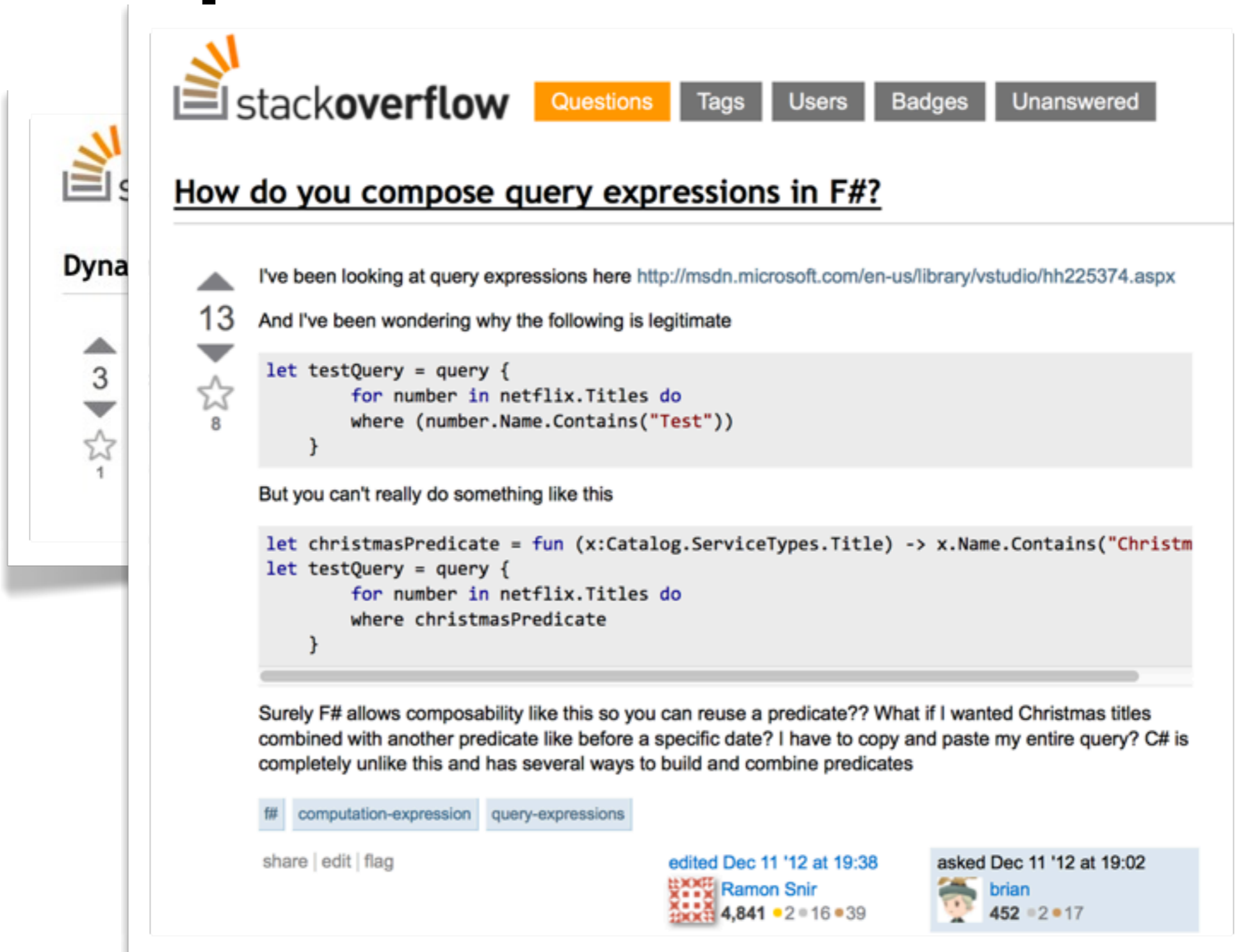
1 share | edit | flag

edited Aug 18 at 21:43
Ruben Bartelink
24.3k • 3 • 79 • 117

asked Apr 15 '12 at 0:18
Arthur Greef
124 • 7

add a comment

Dynamic/composable queries in F#?



The image shows a screenshot of a Stack Overflow question page. The page title is "How do you compose query expressions in F#?". The question is asked by user "brian" and edited by "Ramon Snir". The question text discusses the difficulty of composing query expressions in F# and includes two code snippets. The first snippet shows a query expression for finding titles containing "Test". The second snippet shows a more complex query expression that uses a predicate function to filter titles containing "Christmas". The page also shows the Stack Overflow logo, navigation tabs (Questions, Tags, Users, Badges, Unanswered), and a sidebar with navigation icons.

stackoverflow Questions Tags Users Badges Unanswered

How do you compose query expressions in F#?

I've been looking at query expressions here <http://msdn.microsoft.com/en-us/library/vstudio/hh225374.aspx>

And I've been wondering why the following is legitimate

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

But you can't really do something like this

```
let christmasPredicate = fun (x:Catalog.ServiceTypes.Title) -> x.Name.Contains("Christm
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

f# computation-expression query-expressions

share | edit | flag

edited Dec 11 '12 at 19:38
Ramon Snir
4,841 ● 2 ● 16 ● 39

asked Dec 11 '12 at 19:02
brian
452 ● 2 ● 17

Dynamic/composable queries in F#?



The screenshot shows a Stack Overflow question page. At the top, the Stack Overflow logo is on the left, and navigation buttons for 'Questions', 'Tags', 'Users', 'Badges', and 'Unanswered' are on the right. The question title is 'How do you compose query expressions in F#?'. Below the title, the user's profile picture and name 'Dyna' are visible on the left. The question text reads: 'I've been looking at query expressions here <http://msdn.microsoft.com/en-us/library/vstudio/hh225374.aspx> And I've been wondering why the following is legitimate'. Below this is a code block containing F# query expressions. The second code block shows a lambda function being used as a predicate, which is circled in red. The question has 13 votes and 8 stars. At the bottom, there are tags for 'f#', 'computation-expression', and 'query-expressions', along with user information for the asker and the editor.

stackoverflow Questions Tags Users Badges Unanswered

How do you compose query expressions in F#?

I've been looking at query expressions here <http://msdn.microsoft.com/en-us/library/vstudio/hh225374.aspx>

And I've been wondering why the following is legitimate

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

But you can't really do something like this

```
let christmasPredicate = fun (x:Catalog.ServiceTypes.Title) -> x.Name.Contains("Christm
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

f# computation-expression query-expressions

share | edit | flag

edited Dec 11 '12 at 19:38
Ramon Snir
4,841 ● 2 ● 16 ● 39

asked Dec 11 '12 at 19:02
brian
452 ● 2 ● 17

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

LINQ example

employees

dpt	name	salary
"Product"	"Alex"	40,000
"Product"	"Bert"	60,000
"Research"	"Cora"	50,000
"Research"	"Drew"	70,000
"Sales"	"Erik"	200,000
"Sales"	"Fred"	95,000
"Sales"	"Gina"	155,000

tasks

emp	tsk
"Alex"	"build"
"Bert"	"build"
"Cora"	"abstract"
"Cora"	"build"
"Cora"	"call"
"Cora"	"dissemble"
"Cora"	"enthuse"
"Drew"	"abstract"
"Drew"	"enthuse"
"Erik"	"call"
"Erik"	"enthuse"
"Fred"	"call"
"Gina"	"call"
"Gina"	"dissemble"

quotation

<@ @>

antiquote

(%)

```
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} }
```

LINQ example

employees

dpt	name	salary
"Product"	"Alex"	40,000
"Product"	"Bert"	60,000
"Research"	"Cora"	50,000
"Research"	"Drew"	70,000
"Sales"	"Erik"	200,000
"Sales"	"Fred"	95,000
"Sales"	"Gina"	155,000

tasks

emp	tsk
"Alex"	"build"
"Bert"	"build"
"Cora"	"abstract"
"Cora"	"build"
"Cora"	"call"
"Cora"	"dissemble"
"Cora"	"enthuse"
"Drew"	"abstract"
"Drew"	"enthuse"
"Erik"	"call"
"Erik"	"enthuse"
"Fred"	"call"
"Gina"	"call"
"Gina"	"dissemble"

quotation

<@ @>

antiquote

(%)

```
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} }
```

name

Alex

Bert

Cora

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

$(\mathbf{fun}(x) \rightarrow N) M \rightsquigarrow N[x := M]$

$\{\overline{\ell = M}\}.l_i \rightsquigarrow M_i$

$\mathbf{for } x \mathbf{ in (yield } M) \mathbf{ do } N \rightsquigarrow N[x := M]$

$\mathbf{for } y \mathbf{ in (for } x \mathbf{ in } L \mathbf{ do } M) \mathbf{ do } N \rightsquigarrow \mathbf{for } x \mathbf{ in } L \mathbf{ do (for } y \mathbf{ in } M \mathbf{ do } N)$

$\mathbf{for } x \mathbf{ in (if } L \mathbf{ then } M) \mathbf{ do } N \rightsquigarrow \mathbf{if } L \mathbf{ then (for } x \mathbf{ in } M \mathbf{ do } N)$

$\mathbf{for } x \mathbf{ in } [] \mathbf{ do } N \rightsquigarrow []$

$\mathbf{for } x \mathbf{ in } (L @ M) \mathbf{ do } N \rightsquigarrow (\mathbf{for } x \mathbf{ in } L \mathbf{ do } N) @ (\mathbf{for } x \mathbf{ in } M \mathbf{ do } N)$

$\mathbf{if true then } M \rightsquigarrow M$

$\mathbf{if false then } M \rightsquigarrow []$

Normalisation: *ad hoc* rewriting

for x **in** L **do** $(M @ N)$ \hookrightarrow (**for** x **in** L **do** M) @ (**for** x **in** L **do** N)

for x **in** L **do** $[\]$ \hookrightarrow $[\]$

if L **then** $(M @ N)$ \hookrightarrow (**if** L **then** M) @ (**if** L **then** N)

if L **then** $[\]$ \hookrightarrow $[\]$

if L **then** (**for** x **in** M **do** N) \hookrightarrow **for** x **in** M **do** (**if** L **then** N)

if L **then** (**if** M **then** N) \hookrightarrow **if** $(L \ \&\& \ M)$ **then** N

Example

```
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 ->  
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}}
```

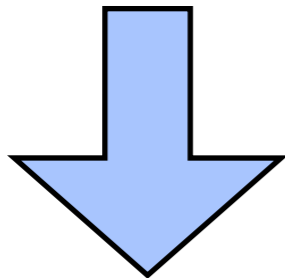
Example

```
let elem = <@ fun x xs ->  
let canDo = <@ fun name tsk ->  
1 query { for x in employees  
q   where ((fun name tsk ->  
qu  (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

```
let elem = <@ fun x xs ->  
let canDo = <@ fun name tsk ->  
1 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")  
qu  
qu  
yield {name = x.name}
```

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



X (failure or query avalanche)

Example

```
let elem = <@ fun x xs ->  
let canDo = <@ fun name tsk ->  
1 query { for x in employees  
q   where ((fun name tsk ->  
qu  (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

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

Example

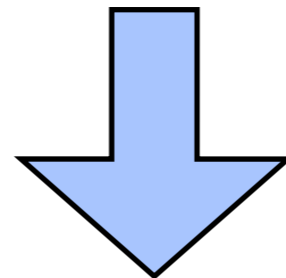
```
let elem = <@ fun x xs ->  
  let canDo = <@ fun name tsk ->  
    query { for x in employees  
      where ((fun name tsk ->  
        query { for x in employees  
          where ((fun name tsk ->  
            query { for x in employees  
              where (query { for y in (for t in tasks 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

Example

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

Example

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



```
SELECT x.name  
FROM employees x  
WHERE EXISTS (SELECT t.tsk FROM tasks t WHERE t.emp = x.name)
```


FSharpComposableQuery library

- 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
- <https://github.com/fsprojects/FSharp.Linq.ComposableQuery>
- <http://www.nuget.org/packages/FSharpComposableQuery/>