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

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”

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

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

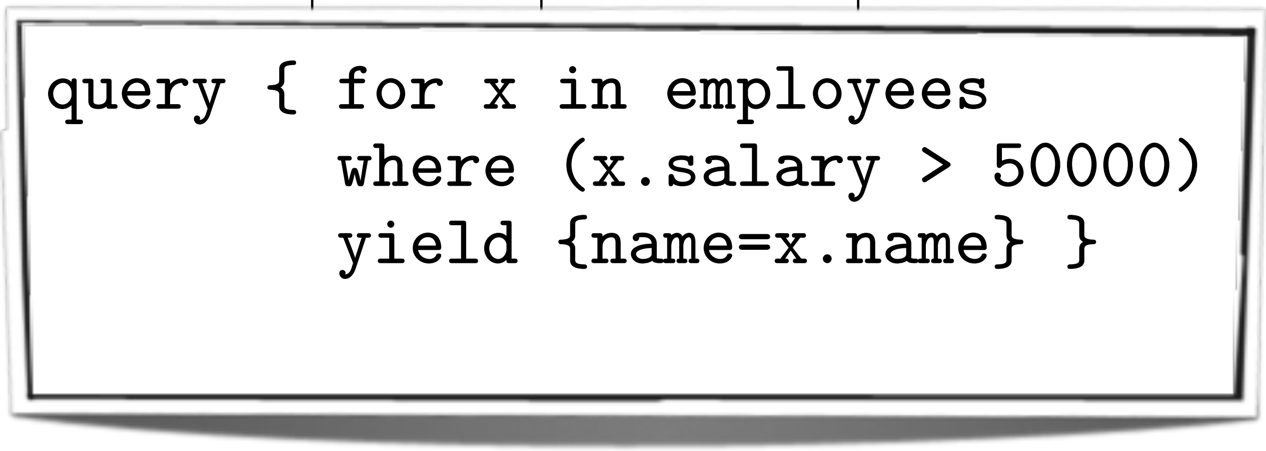
LINQ (F#) example

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”	“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
```

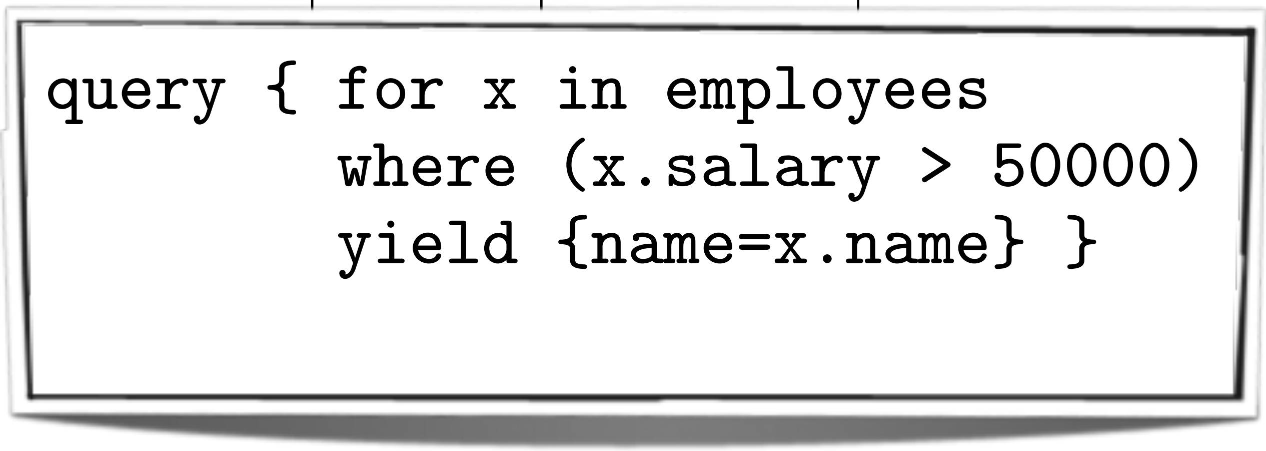
LINQ (F#) example

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”

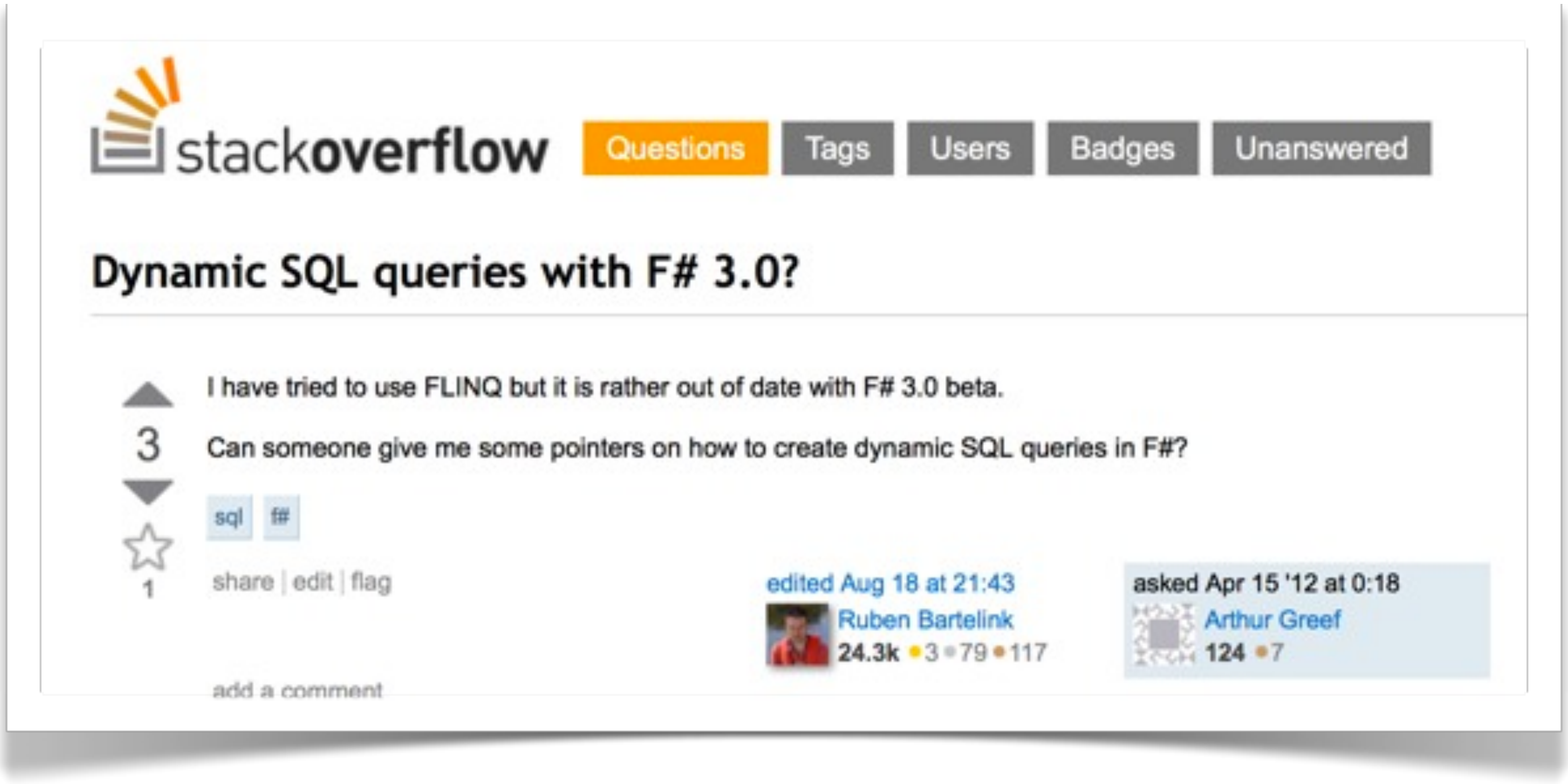


```
query { for x in employees
        where (x.salary > 50000)
        yield {name=x.name} }
```

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

name
Bert
Drew
Erik
Fred
Gina

Dynamic/composable queries in F#?



Dynamic SQL queries with F# 3.0?

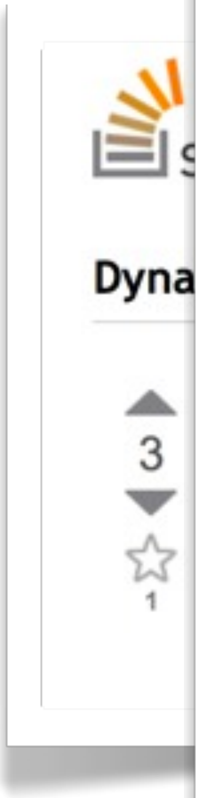
I have tried to use FLINQ but it is rather out of date with F# 3.0 beta.

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

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

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

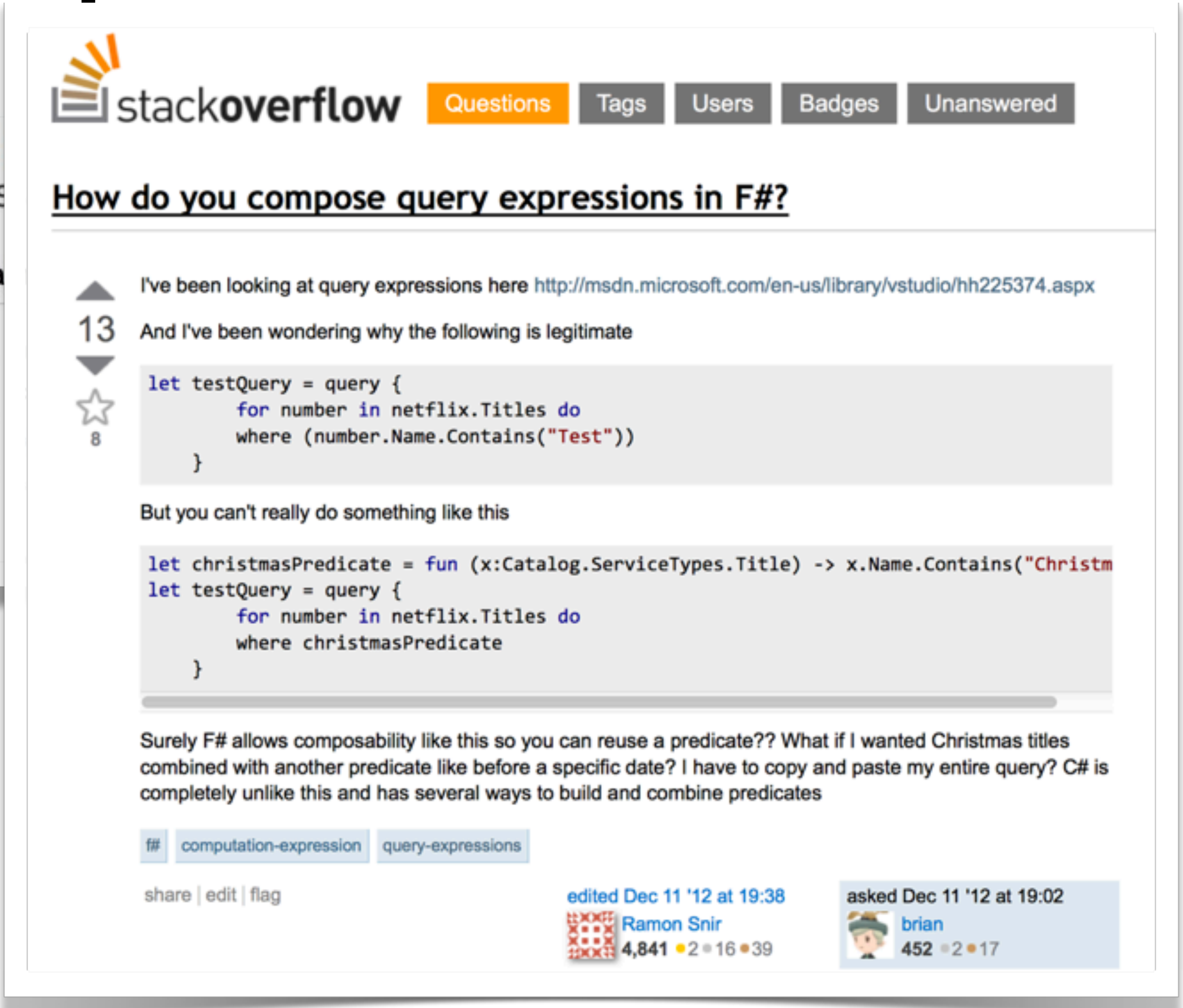
Dynamic/composable queries in F#?



Dyna

3

1



How do you compose query expressions in F#?

13

8

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

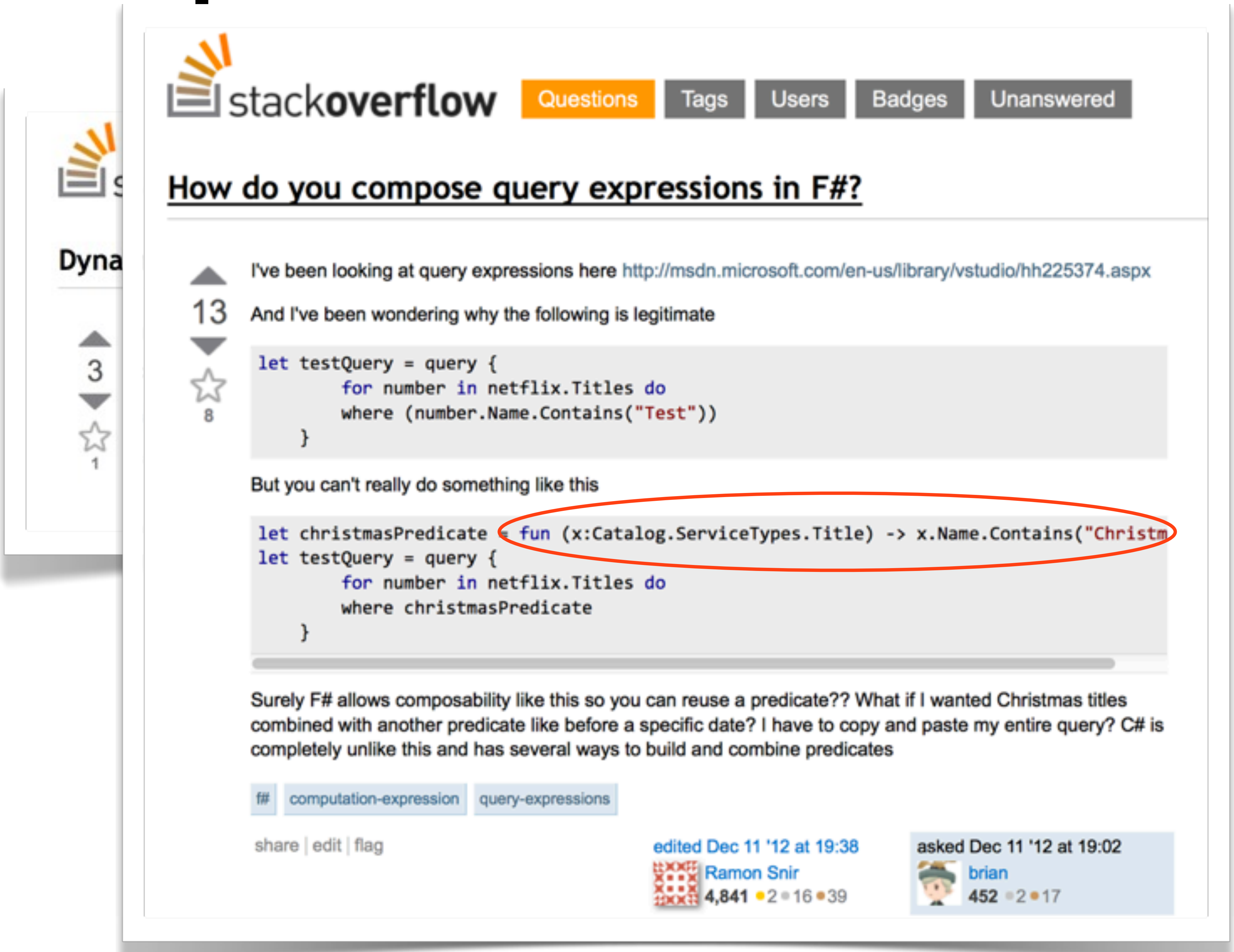
f# computation-expression query-expressions

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#?



How do you compose query expressions in F#?

13

8

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

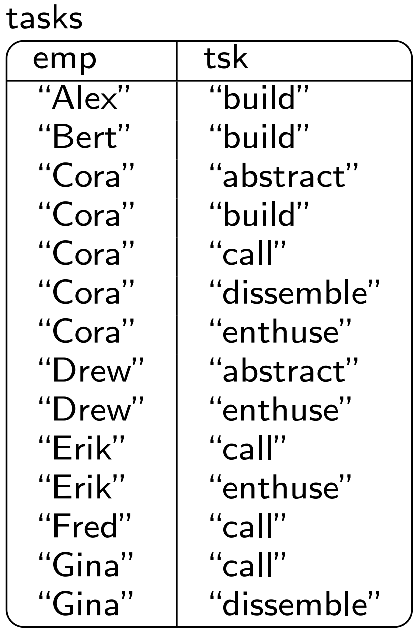
edited Dec 11 '12 at 19:38
Ramon Snir
4,841 •2 •16 •39

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

Queries with function "parameters"?

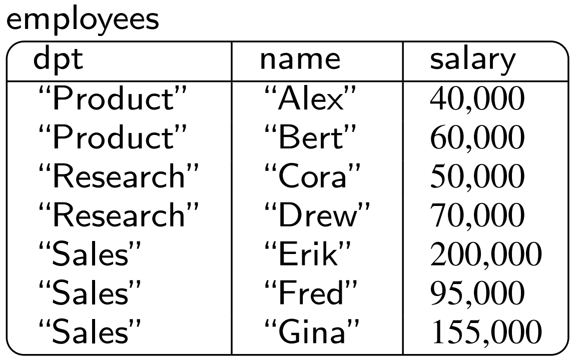
- A way to (de)compose queries into reusable chunks?
  - (avoid repeating yourself)
- This could be very useful
  - a form of staged computation/meta-programming
- Queries could be constructed dynamically
  - including constructing queries of different "shape"
  - goes beyond simple int/string parameters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

LINQ example



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”

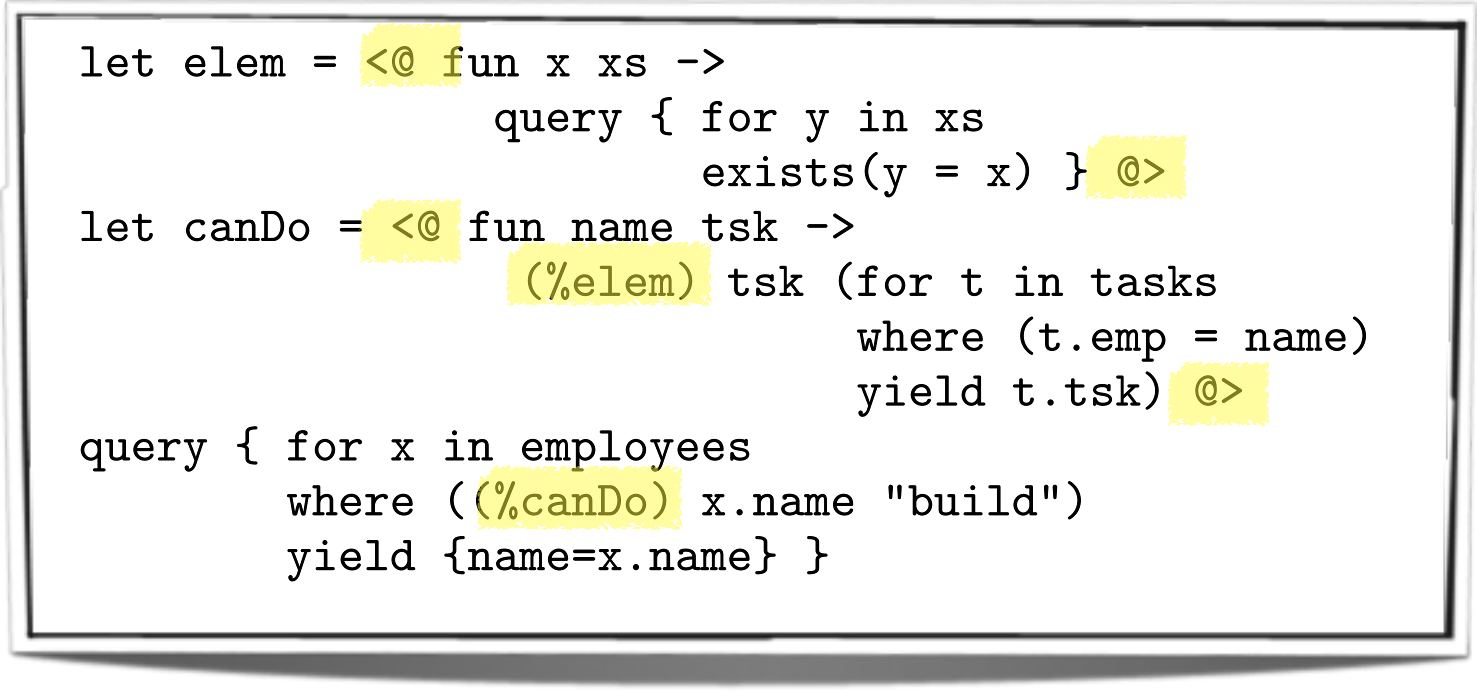


employees

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

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

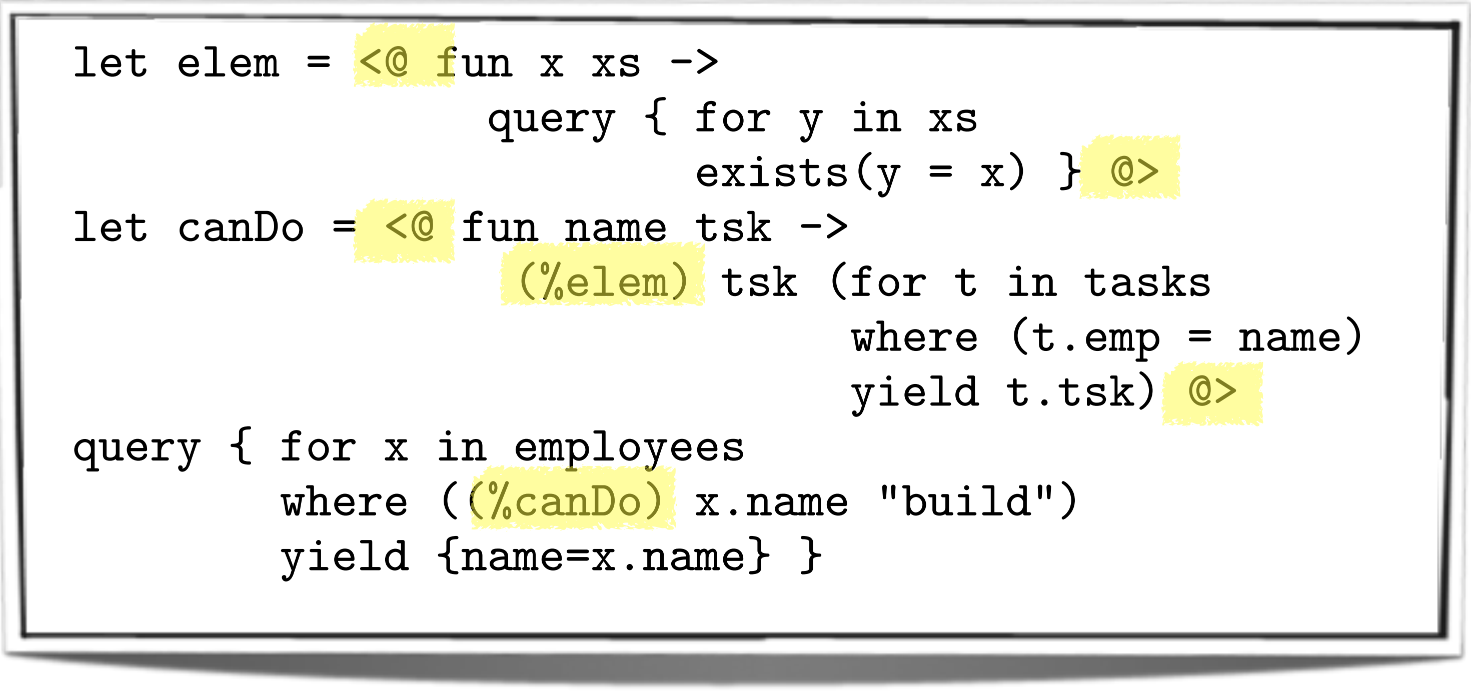
employees

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
<@ @>



```
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} }
```

antiquote
(%)

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

$$
\begin{aligned}
(\textbf{fun}(x) \to N)\ M &\leadsto N[x := M] \\
\{\overline{\ell = M}\}.\ell_i &\leadsto M_i \\
\textbf{for}\ x\ \textbf{in}\ (\textbf{yield}\ M)\ \textbf{do}\ N &\leadsto N[x := M] \\
\textbf{for}\ y\ \textbf{in}\ (\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ M)\ \textbf{do}\ N &\leadsto \textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ (\textbf{for}\ y\ \textbf{in}\ M\ \textbf{do}\ N) \\
\textbf{for}\ x\ \textbf{in}\ (\textbf{if}\ L\ \textbf{then}\ M)\ \textbf{do}\ N &\leadsto \textbf{if}\ L\ \textbf{then}\ (\textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ N) \\
\textbf{for}\ x\ \textbf{in}\ \texttt{[ ]}\ \textbf{do}\ N &\leadsto \texttt{[ ]} \\
\textbf{for}\ x\ \textbf{in}\ (L\ \texttt{@}\ M)\ \textbf{do}\ N &\leadsto (\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ N)\ \texttt{@}\ (\textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ N) \\
\textbf{if true then}\ M &\leadsto M \\
\textbf{if false then}\ M &\leadsto \texttt{[ ]}
\end{aligned}
$$

slide stolen from Phil Wadler's talk

Normalisation: *ad hoc* rewriting

$$
\begin{aligned}
\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ (M\ \mathtt{@}\ N) &\hookrightarrow (\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ M)\ \mathtt{@}\ (\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ N) \\
\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ \mathtt{[\ ]} &\hookrightarrow \mathtt{[\ ]} \\
\textbf{if}\ L\ \textbf{then}\ (M\ \mathtt{@}\ N) &\hookrightarrow (\textbf{if}\ L\ \textbf{then}\ M)\ \mathtt{@}\ (\textbf{if}\ L\ \textbf{then}\ N) \\
\textbf{if}\ L\ \textbf{then}\ \mathtt{[\ ]} &\hookrightarrow \mathtt{[\ ]} \\
\textbf{if}\ L\ \textbf{then}\ (\textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ N) &\hookrightarrow \textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ (\textbf{if}\ L\ \textbf{then}\ N) \\
\textbf{if}\ L\ \textbf{then}\ (\textbf{if}\ M\ \textbf{then}\ N) &\hookrightarrow \textbf{if}\ (L\ \ \&\&\ \ M)\ \textbf{then}\ N
\end{aligned}
$$

slide stolen from Phil Wadler's talk

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 ->
              query { for y in xs
l
q
```

```
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 ->
                 query { for y in xs

l

q
```

```
let canDo = <@ fun name tsk ->
                 (fun x xs ->



qu
```

```
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

```
let elem = <@ fun x xs ->
```

```
let canDo = <@ fun name 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}
```

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

X (failure or query avalanche)

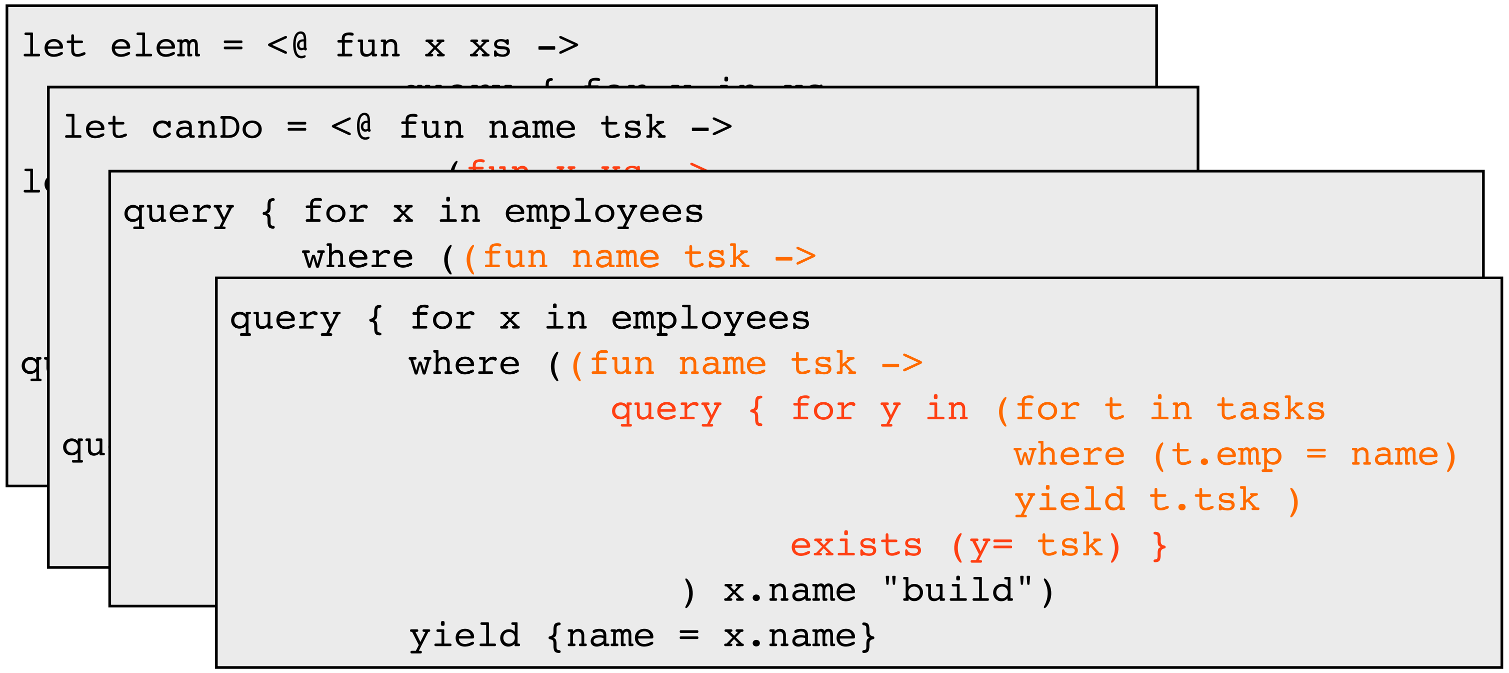
Example

```
let elem = <@ fun x xs ->
              query { for y in xs
l
q
```

```
let canDo = <@ fun name tsk ->
               (fun x xs ->
qu
```

```
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



```
let elem = <@ fun x xs ->
              query { for y in xs
l
q
```

```
let canDo = <@ fun name tsk ->
               (fun x xs ->
qu
```

```
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 ->
                query { for y in xs
l
q
```

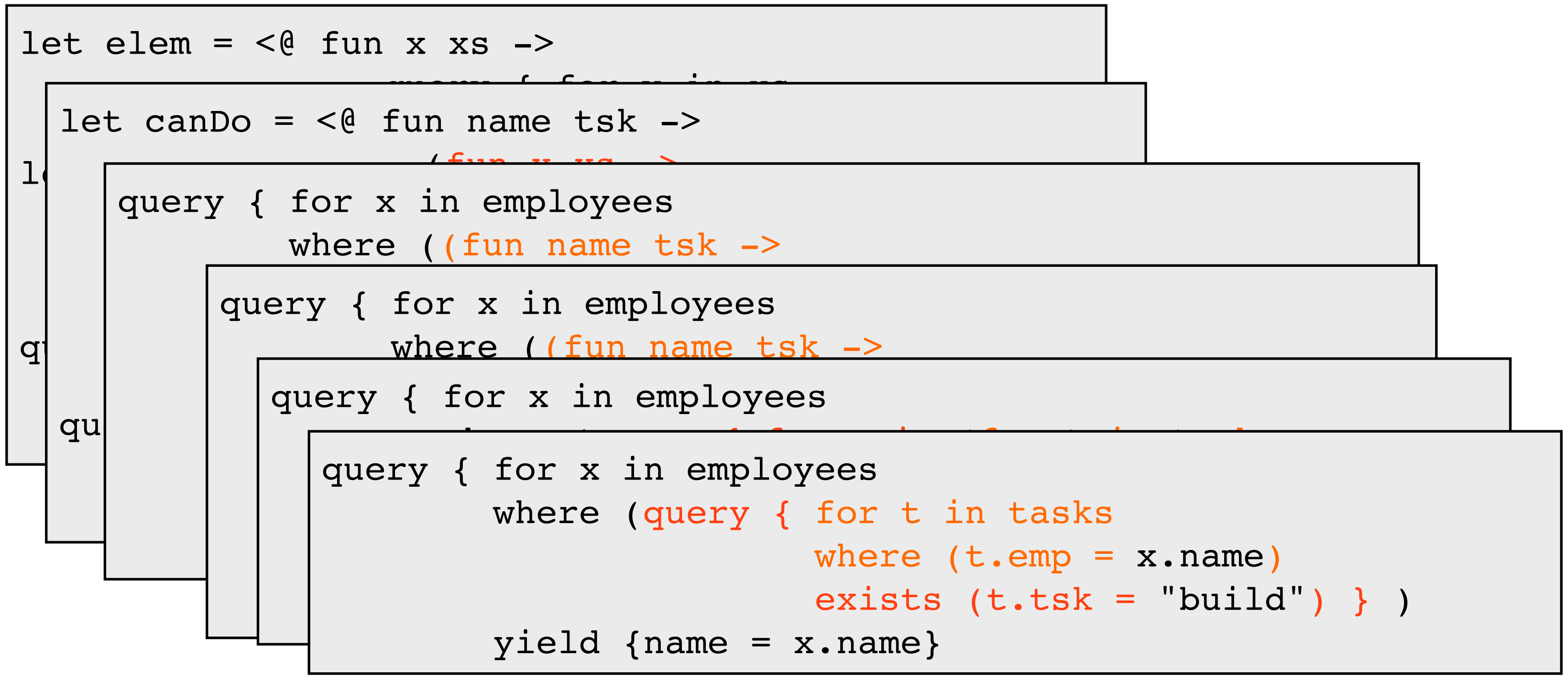
```
let canDo = <@ fun name tsk ->
                (fun x xs ->
qu
```

```
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 ->
                query { for x in xs
l
q
```

```
let canDo = <@ fun name tsk ->
                (fun x xs ->
qu
```

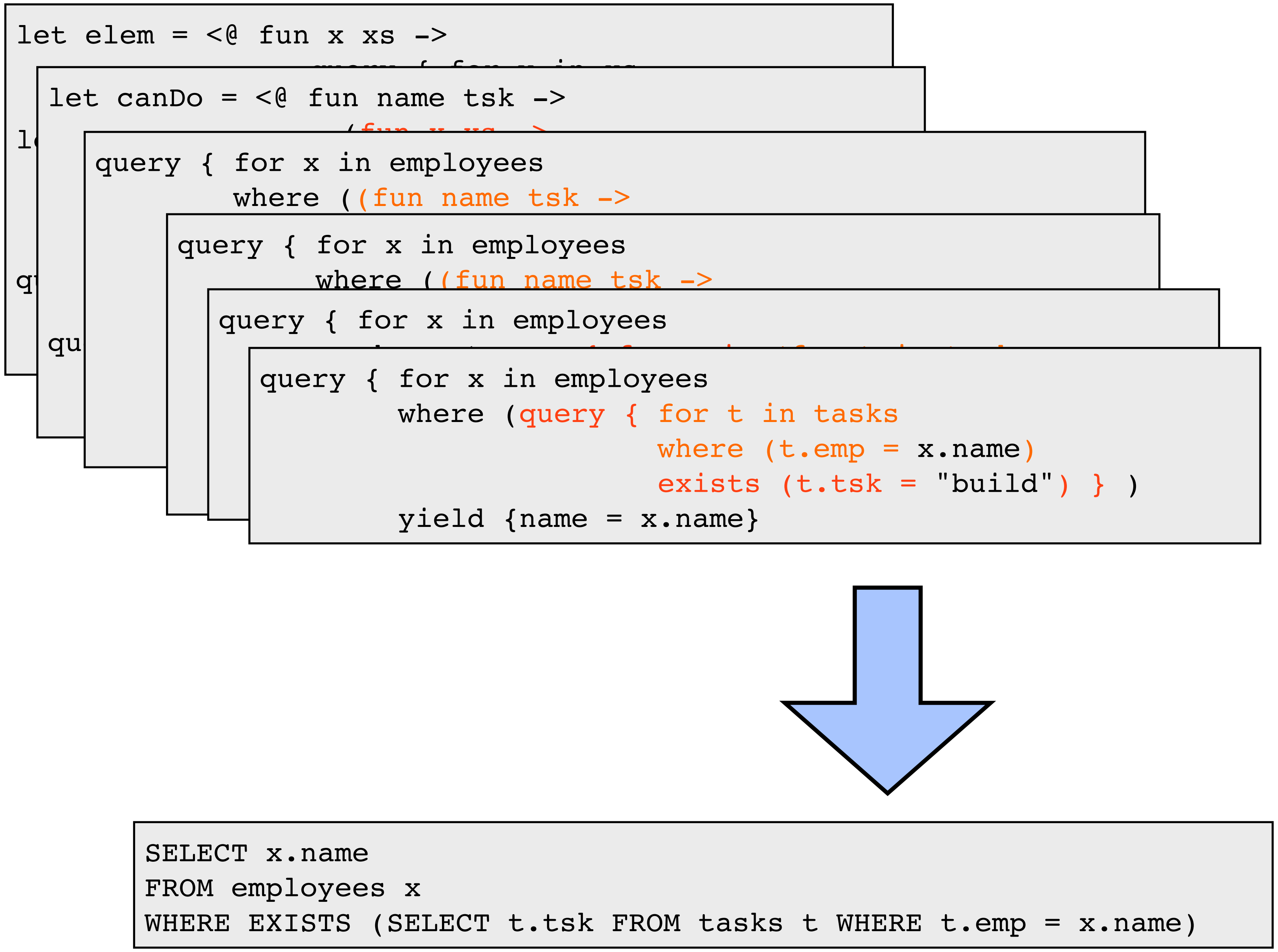
```
query { for x in employees
        where ((fun name tsk ->
```

```
query { for x in employees
        where ((fun name tsk ->
```

```
query { for x in employees
```

```
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
```

```
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/