A practical theory of Language Integrated Query

James Cheney, Sam Lindley, Philip Wadler
University of Edinburgh

Scala Workshop
Montpellier, 2 July 2013
What is the difference between theory and practice?
In theory there is no difference.
But in practice there is.
A tale of two languages

Links
Cooper, Lindley, Wadler, Yallop
(Edinburgh)

LINQ for C#, VB, F#
Hejlsberg, Meijer, Syme
(Microsoft Redmond & Cambridge)
Goals

*Series of examples*
- Join queries
- Abstraction over values (first-order)
- Abstraction over predicates (higher-order)
- Composition of queries
- Dynamic generation of queries
- Nested intermediate data

*Type safety*

*Avoid Scylla and Charybdis*
- Each host query generates one SQL query
  - *Scylla*: failure to generate a query
  - *Charybdis*: multiple queries, avalanche
Britannia between Scylla & Charybdis.

or The Vessel of the Constitution steered clear of the Rock of Democracy, and the Whirlpool of Arbitrary Power.
Limitations

Restrictions on the theory:
We consider only comprehensions, unions, and existence tests.
Future work to extend to grouping, sorting, and aggregation.

Notational convention:
We treat bags (multisets) as lists.
Part I

A first example
A database

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Alex”</td>
<td>60</td>
</tr>
<tr>
<td>“Bert”</td>
<td>56</td>
</tr>
<tr>
<td>“Cora”</td>
<td>33</td>
</tr>
<tr>
<td>“Drew”</td>
<td>31</td>
</tr>
<tr>
<td>“Edna”</td>
<td>21</td>
</tr>
<tr>
<td>“Fred”</td>
<td>60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>her</th>
<th>him</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Alex”</td>
<td>“Bert”</td>
</tr>
<tr>
<td>“Cora”</td>
<td>“Drew”</td>
</tr>
<tr>
<td>“Edna”</td>
<td>“Fred”</td>
</tr>
</tbody>
</table>
A query in SQL

```sql
select w.name as name, w.age - m.age as diff
from couples as c,
    people as w,
    people as m
where c.her = w.name and c.him = m.name and w.age > m.age
```

<table>
<thead>
<tr>
<th>name</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Alex”</td>
<td>4</td>
</tr>
<tr>
<td>“Cora”</td>
<td>2</td>
</tr>
</tbody>
</table>
A database as data

```javascript
{people =
  [ {name = “Alex” ; age = 60} ;
    {name = “Bert” ; age = 56} ;
    {name = “Cora” ; age = 33} ;
    {name = “Drew” ; age = 31} ;
    {name = “Edna” ; age = 21} ;
    {name = “Fred” ; age = 60} ] ;
couples =
  [ {her = “Alex” ; him = “Bert” } ;
    {her = “Cora” ; him = “Drew”} ;
    {her = “Edna” ; him = “Fred” } ] }
```
Importing the database (naive)

```haskell
type DB =
  { people :
    { name : string; age : int } list;
  couples :
    { her : string; him : string } list }
let db' : DB = database("People")
```
A query as a comprehension (naive)

```plaintext
let differences' : {name : string; diff : int} list =
    for c in db'.couples do
        for w in db'.people do
            for m in db'.people do
                if c.her = w.name && c.him = m.name && w.age > m.age then
                    yield {name : w.name; diff : w.age - m.age}

differences'
    [{name = "Alex"; diff = 4}
     {name = "Cora"; diff = 2}]```
Importing the database (quoted)

```plaintext
type DB =
    {people :
        {name : string; age : int} list;
    
couples :
        {her : string; him : string} list
    };

let db : Expr<DB> = <! database("People") @>
A query as a comprehension (quoted)

```plaintext
let differences : Expr< {name : string; diff : int} list > =
  @{
    for c in (%db).couples do
      for w in (%db).people do
        for m in (%db).people do
          if c.her = w.name && c.him = m.name && w.age > m.age then
            yield {name : w.name; diff : w.age - m.age} @
  }

run(differences)

[ {name = “Alex”; diff = 4}
  {name = “Cora”; diff = 2} ]
```
Running a query

1. compute quoted expression
2. simplify quoted expression
3. translate query to SQL
4. execute SQL
5. translate answer to host language

*Scylla and Charybdis:*

Each **run** generates one query if

- A. answer type is flat (bag of record of scalars)
- B. only permitted operations (e.g., no recursion)
- C. only refers to one database
Scala (naive)

val differences:
List[{ val name: String; val diff: Int }] =
for {
  c <- db.couples
  w <- db.people
  m <- db.people
  if c.her == w.name && c.him == m.name && w.age > m.age
} yield new Record {
  val name = w.name
  val diff = w.age - m.age
}
Scala (quoted)

val differences: 
  Rep[List[{ val name: String; val diff: Int }]] = 
  for {
    c <- db.couples
    w <- db.people
    m <- db.people
    if c.her == w.name && c.him == m.name && w.age > m.age
  } yield new Record {
    val name = w.name
    val diff = w.age - m.age
  }
Part II

Abstraction, composition, dynamic generation
Abstracting over values

```plaintext
type Names = {name : string} list
let range : Expr<(int, int) -> Names> =
  <@ fun(a, b) -> for w in (%db).people do
    if a <= w.age && w.age < b then
      yield {name : w.name} @>

run(<@ (%range)(30, 40) @>)
  [{name = “Cora”}; {name = “Drew”}]
```
Abstracting over a predicate

\[
\text{let satisfies : Expr< (int → bool) → Names >} = \\
\text{@ fun(p) → for w in (%db).people do} \\
\text{  if p(w.age) then} \\
\text{  yield \{name : w.name\}} \text{@>}
\]

\[
\text{run(@ (%satisfies)(fun(x) → 30 ≤ x && x < 40) @>)} \\
[ \{\text{name = “Cora”}\}; \{\text{name = “Drew”}\} ]
\]

\[
\text{run(@ (%satisfies)(fun(x) → x mod 2 = 0) @>)} \\
[ \{\text{name = “Alex”}\}; \{\text{name = “Bert”}\}; \{\text{name = “Fred”}\} ]
\]
Composing queries

```ocaml
let getAge : Expr< string → int list > =
  <@ fun(s) → for u in (%db).people do
  if u.name = s then
  yield u.age @>

let compose : Expr< (string, string) → Names > =
  <@ fun(s, t) → for a in (%getAge)(s) do
  for b in (%getAge)(t) do
  (%range)(a, b) @>

run(<@ (%compose)(“Edna”, “Bert”) @>)

[ {name = “Cora”}; {name = “Drew”}; {name = “Edna”} ]
```
Dynamically generated queries (1)

```
type Predicate =
  | Above of int
  | Below of int
  | And of Predicate × Predicate
  | Or of Predicate × Predicate
  | Not of Predicate

let t_0 : Predicate = And(Above(30), Below(40))
let t_1 : Predicate = Not(Or(Below(30), Above(40))))
```
Dynamically generated queries (2)

```ocaml
let rec P(t : Predicate) : Expr< int → bool > =
  match t with
  | Above(a) → @fun(x) → (%lift(a)) ≤ x @>
  | Below(a) → @fun(x) → x < (%lift(a)) @>
  | And(t, u) → @fun(x) → (%P(t))(x) && (%P(u))(x) @>
  | Or(t, u) → @fun(x) → (%P(t))(x) || (%P(u))(x) @>
  | Not(t) → @fun(x) → not(%P(t))(x) @>
```
Dynamically generated queries (3)

\[
P(t_0)\]
\[
<@ \text{fun}(x) \mapsto (\text{fun}(x) \mapsto 30 \leq x)(x) \land (\text{fun}(x) \mapsto x < 40)(x) @>
\]
\[
<@ \text{fun}(x) \mapsto 30 \leq x \land x < 40 @>
\]

\[
\text{run}(<@ (%satisfies)(%P(t_0)) @>)
\]
\[
[\{\text{name} = \text{“Cora”}\}; \{\text{name} = \text{“Drew”}\}]
\]

\[
\text{run}(<@ (%satisfies)(%P(t_1)) @>)
\]
\[
[\{\text{name} = \text{“Cora”}\}; \{\text{name} = \text{“Drew”}\}]
\]
Part III

Nesting
Flat data

{departments =
    [ {dpt = “Product”} ;
    {dpt = “Quality”} ;
    {dpt = “Research”} ;
    {dpt = “Sales”} ] ;

employees =
    [ {dpt = “Product”; emp = “Alex”} ;
    {dpt = “Product”; emp = “Bert”} ;
    {dpt = “Research”; emp = “Cora”} ;
    {dpt = “Research”; emp = “Drew”} ;
    {dpt = “Research”; emp = “Edna”} ;
    {dpt = “Sales”; emp = “Fred”} ] ;
Flat data (continued)

tasks =

    [
        {emp = “Alex”; tsk = “build”};
        {emp = “Bert”; tsk = “build”};
        {emp = “Cora”; tsk = “abstract”};
        {emp = “Cora”; tsk = “build”};
        {emp = “Cora”; tsk = “design”};
        {emp = “Drew”; tsk = “abstract”};
        {emp = “Drew”; tsk = “design”};
        {emp = “Edna”; tsk = “abstract”};
        {emp = “Edna”; tsk = “call”};
        {emp = “Edna”; tsk = “design”};
        {emp = “Fred”; tsk = “call”} ] }
Importing the database

```plaintext
type Org = {departments : {dpt : string} list;
            employees : {dpt : string; emp : string} list;
            tasks : {emp : string; tsk : string} list };

let org : Expr<Org> = <$ database("Org") $>
```
Departments where every employee can do a given task

```haskell
let expertise' : Expr< string → {dpt : string} list > =
  @@ fun(u) →
    for d in (%org).departments do
      if not(exists(
        for e in (%org).employees do
          if d.dpt = e.dpt && not(exists(
            for t in (%org).tasks do
              if e.emp = t.emp && t.tsk = u then yield { }
          ))) then yield { }
      ))) then yield {dpt = d.dpt} @@
run(@@ (%expertise')(“abstract”) @@)
[
  {dpt = “Quality”};
  {dpt = “Research”}
]```
Nested data

```javascript

[ {dpt = "Product"; employees =
    [ {emp = "Alex"; tasks = [ "build" ] }
    {emp = "Bert"; tasks = [ "build" ] } ] },

{dpt = "Quality"; employees = [ ] },

{dpt = "Research"; employees =
    [ {emp = "Cora"; tasks = [ "abstract"; "build"; "design" ] }
    {emp = "Drew"; tasks = [ "abstract"; "design" ] }
    {emp = "Edna"; tasks = [ "abstract"; "call"; "design" ] } ] },

{dpt = "Sales"; employees =
    [ {emp = "Fred"; tasks = [ "call" ] } ] } ]
```
Nested data from flat data

```plaintext
type NestedOrg = [ {dpt: string; employees:
    [{emp: string; tasks: [string]}]} ]

let nestedOrg : Expr<NestedOrg> =
  @@ for d in (%org).departments do
    yield {dpt = d.dpt; employees =
      for e in (%org).employees do
        if d.dpt = e.dpt then
          yield {emp = e.emp; tasks =
            for t in (%org).tasks do
              if e.emp = t.emp then
                yield t.tsk}}}} @@
```
Higher-order queries

```plaintext
let any : Expr< (A list, A → bool) → bool > =
@@ fun(xs, p) →
   exists(for x in xs do
     if p(x) then
       yield { } ) @@

let all : Expr< (A list, A → bool) → bool > =
@@ fun(xs, p) →
   not( not( (%any)(xs, fun(x) → not(p(x)))) ) @@

let contains : Expr< (A list, A) → bool > =
@@ fun(xs, u) →
   (%any)(xs, fun(x) → x = u ) @@
```
Departments where every employee can do a given task

```haskell
let expertise : Expr< string → {dpt : string} list > =
  <$> fun(u) → for d in (%nestedOrg)
    if (%all)(d.employees,
        fun(e) → (%contains)(e.tasks, u) then
    yield {dpt = d.dpt} @>

run <$> (%expertise)(“abstract”) @>
[ {dpt = “Quality”}; {dpt = “Research”} ]
```
Part IV

Quotations vs. functions
Abstracting over values

```plaintext
let range : Expr<(int, int) -> Names> =
    <@ fun(a, b) -> for w in (%db).people do
           if a <= w.age && w.age < b then
               yield {name : w.name} @>
run(<@ (%range)(30, 40) @>)

vs.

let range'(a : Expr<int>, b : Expr<int>) : Names =
    <@ for w in (%db).people do
        if (%a) <= w.age && w.age < (%b) then
            yield {name : w.name} @>
run(range'(<@ 30 @>, <@ 40 @>))
```
Composing queries

\[
\textbf{let} \ \text{compose} : \ \text{Expr}<\text{(string, string)} \rightarrow \text{Names}> = \\
<@ \ \text{fun}(s, t) \rightarrow \text{for a in } (%\text{getAge})(s) \text{ do} \\
\quad \text{for b in } (%\text{getAge})(t) \text{ do} \\
\quad (%\text{range})(a, b) @> \\
\]

to

\[
\textbf{let} \ \text{compose}' : \ \text{Expr}<\text{(string, string)} \rightarrow \text{Names}> = \\
<@ \ \text{fun}(s, t) \rightarrow \text{for a in } (%\text{getAge})(s) \text{ do} \\
\quad \text{for b in } (%\text{getAge})(t) \text{ do} \\
\quad (%\text{range}'(<@ a @>, <@ b @>)) @>
\]
Prefer

*closed quotations*

to

*open quotations.*

Prefer

*quotations of functions*

to

*functions of quotations.*
Part V

From XPath to SQL
Part VI

Idealised LINQ
Terms

\[ \text{VAR} \]
\[ \Gamma, x : A \vdash x : A \]

\[ \text{FUN} \]
\[ \frac{\Gamma, x : A \vdash N : B}{\Gamma \vdash \text{fun}(x) \to N : A \to B} \]

\[ \text{APP} \]
\[ \frac{\Gamma \vdash L : A \to B \quad \Gamma \vdash M : A}{\Gamma \vdash L \ M : B} \]

\[ \text{SINGLETON} \]
\[ \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{yield} \ M : A \ \text{list}} \]

\[ \text{FOR} \]
\[ \frac{\Gamma \vdash M : A \ \text{list} \quad \Gamma, x : A \vdash N : B \ \text{list}}{\Gamma \vdash \text{for} \ x \ \text{in} \ M \ \text{do} \ N : B \ \text{list}} \]

\[ \text{REC} \]
\[ \frac{\Gamma, f : A \to B, x : A \vdash N : B}{\Gamma \vdash \text{rec} \ f(x) \to N : A \to B} \]
Quoted terms

\[
\begin{align*}
\text{VARQ} & \quad \frac{}{\Gamma; \Delta, x : A \vdash x : A} \\
\text{FUNQ} & \quad \frac{\Gamma; \Delta, x : A \vdash N : B}{\Gamma; \Delta \vdash \text{fun}(x) \to N : A \to B} \\
\text{APPQ} & \quad \frac{\Gamma; \Delta \vdash L : A \to B}{\Gamma; \Delta \vdash L \ M : B} \\
\text{SINGLETONQ} & \quad \frac{\Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash \text{yield} \ M : A \ \text{list}} \\
\text{FORQ} & \quad \frac{\Gamma; \Delta \vdash M : A \ \text{list}}{\Gamma; \Delta \vdash \text{for} \ x \ \text{in} \ M \ \text{do} \ N : B \ \text{list}} \\
\text{DATABASE} & \quad \frac{\Sigma(db) = \{\ell : T\}}{\Gamma; \Delta \vdash \text{database}(db) : \{\ell : T\}}
\end{align*}
\]
Quotation and anti-quotation

**QUOTE**
\[
\Gamma; \cdot \vdash M : A \\
\Gamma \vdash \langle@ M \rangle : \text{Expr}<A>
\]

**ANTIQUOTE**
\[
\Gamma \vdash M : \text{Expr}<A> \\
\Gamma; \Delta \vdash (\% M) : A
\]

**RUN**
\[
\Gamma \vdash M : \text{Expr}<T> \\
\Gamma \vdash \text{run}(M) : T
\]

**LIFT**
\[
\Gamma \vdash M : O \\
\Gamma \vdash \text{lift}(M) : \text{Expr}<O>
\]
Normalisation: symbolic evaluation

\[
\begin{align*}
(fu_n(x) \rightarrow N) M & \rightsquigarrow N[x := M] \\
\{\ell = M\}.\ell_i & \rightsquigarrow M_i \\
\text{for } x \text{ in (yield } M \text{) do } N & \rightsquigarrow N[x := M] \\
\text{for } y \text{ in (for } x \text{ in } L \text{ do } M \text{) do } N & \rightsquigarrow \text{for } x \text{ in } L \text{ do (for } y \text{ in } M \text{ do } N) \\
\text{for } x \text{ in (if } L \text{ then } M \text{) do } N & \rightsquigarrow \text{if } L \text{ then (for } x \text{ in } M \text{ do } N) \\
\text{for } x \text{ in [ ] do } N & \rightsquigarrow [ ] \\
\text{for } x \text{ in } (L @ M) \text{ do } N & \rightsquigarrow (\text{for } x \text{ in } L \text{ do } N) @ (\text{for } x \text{ in } M \text{ do } N) \\
\text{if } \text{true then } M & \rightsquigarrow M \\
\text{if } \text{false then } M & \rightsquigarrow [ ]
\end{align*}
\]
Normalisation: *ad hoc* rewriting

\[
\begin{align*}
\text{for } x \text{ in } L \text{ do } (M @ N) & \leftrightarrow (\text{for } x \text{ in } L \text{ do } M) @ (\text{for } x \text{ in } L \text{ do } N) \\
\text{for } x \text{ in } L \text{ do } [ ] & \leftrightarrow [ ] \\
\text{if } L \text{ then } (M @ N) & \leftrightarrow (\text{if } L \text{ then } M) @ (\text{if } L \text{ then } N) \\
\text{if } L \text{ then } [ ] & \leftrightarrow [ ] \\
\text{if } L \text{ then } (\text{for } x \text{ in } M \text{ do } N) & \leftrightarrow \text{for } x \text{ in } M \text{ do } (\text{if } L \text{ then } N) \\
\text{if } L \text{ then } (\text{if } M \text{ then } N) & \leftrightarrow \text{if } (L \&\& M) \text{ then } N
\end{align*}
\]
Properties of reduction

On well-typed terms, the relations $\rightsquigarrow$ and $\rightarrow$

- preserve typing,
- are strongly normalising, and
- are confluent.

Terms in normal form under $\rightsquigarrow$ satisfy the subformula property: with the exception of predicates (such as $<$ or exists), the type of a subterm must be a subformula of either the type of a free variable or of the type of the term.
Rewriting in Scala

/*
  for x in [] do N --> []
  for x in (yield Q) do R --> R[x:=Q]
  for x in (P @ Q) do R -->
  (for x in P do R) @ (for x in Q do R)
  for x in (if P then Q) do R -->
  if P then (for x in Q do R)
  for y in (for x in P do Q) do R -->
  for x in P do (for y in Q do R)
*/

  case Empty() => List()
  case Yield(a) => f(a)
  case Concat(a,b) => a.flatMap(f) ++ b.flatMap(f)
  case IfThen(c,a) =>
    if (c) for (x <- a; y <- f(x)) yield y else List()
  case For(l2,f2) =>
    for (x <- l2; y <- f2(x); z <- f(y)) yield z
  ...
}
Example (1): query

```
run(@(compose)(“Edna”, “Bert”))
```
Example (2): after splicing

```
(fun(s, t) →
  for a in (fun(s) →
    for u in database(“People”).people do
      if u.name = s then yield u.age)(s) do
  for b in (fun(s) →
    for u in database(“People”).people do
      if u.name = s then yield u.age)(t) do
  (fun(a, b) →
    for w in database(“People”).people do
      if a ≤ w.age && w.age < b then
        yield {name : w.name})(a, b))

(“Edna”, “Bert”)```
Example (3): beta reduction

```python
for a in (for u in database("People").people do
    if u.name == "Edna" then yield u.age) do
for b in (for u in database("People").people do
    if u.name == "Bert" then yield u.age) do
for w in database("People").people do
    if a <= w.age && w.age < b then
        yield {name : w.name}
```
Example (4): other rewriting

```python
for u in database("People").people do
    if u.name == "Edna" then
        for v in database("People").people do
            if v.name == "Bert" then
                for w in database("People").people do
                    if u.age <= w.age && w.age < v.age then
                        yield {name : w.name}
```
Example (5): *ad hoc* reductions ↦

```plaintext
for u in database(“People”).people do
for v in database(“People”).people do
for w in database(“People”).people do
if u.name = “Edna” && v.name = “Bert” &&
    u.age ≤ w.age && w.age < v.age then
  yield {name : w.name}
```
Example (6): SQL

```sql
select w.name as name
from people as u,
    people as v,
    people as w
where u.name = "Edna" and v.name = "Bert" and
    u.age <= w.age and w.age < v.age
```
Part VII

Results
<table>
<thead>
<tr>
<th>Example</th>
<th>F# 2.0</th>
<th>F# 3.0</th>
<th>us</th>
<th>(norm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>differences</td>
<td>17.6</td>
<td>20.6</td>
<td>18.1</td>
<td>0.5</td>
</tr>
<tr>
<td>range</td>
<td>×</td>
<td>5.6</td>
<td>2.9</td>
<td>0.3</td>
</tr>
<tr>
<td>satisfies</td>
<td>2.6</td>
<td>×</td>
<td>2.9</td>
<td>0.3</td>
</tr>
<tr>
<td>satisfies</td>
<td>4.4</td>
<td>×</td>
<td>4.6</td>
<td>0.3</td>
</tr>
<tr>
<td>compose</td>
<td>×</td>
<td>×</td>
<td>4.0</td>
<td>0.8</td>
</tr>
<tr>
<td>P(t₀)</td>
<td>2.8</td>
<td>×</td>
<td>3.3</td>
<td>0.3</td>
</tr>
<tr>
<td>P(t₁)</td>
<td>2.7</td>
<td>×</td>
<td>3.0</td>
<td>0.3</td>
</tr>
<tr>
<td>expertise'</td>
<td>7.2</td>
<td>9.2</td>
<td>8.0</td>
<td>0.6</td>
</tr>
<tr>
<td>expertise</td>
<td>×</td>
<td>66.7&lt;sup&gt;av&lt;/sup&gt;</td>
<td>8.3</td>
<td>0.9</td>
</tr>
<tr>
<td>xp₀</td>
<td>×</td>
<td>8.3</td>
<td>7.9</td>
<td>1.9</td>
</tr>
<tr>
<td>xp₁</td>
<td>×</td>
<td>14.7</td>
<td>13.4</td>
<td>1.1</td>
</tr>
<tr>
<td>xp₂</td>
<td>×</td>
<td>17.9</td>
<td>20.7</td>
<td>2.2</td>
</tr>
<tr>
<td>xp₃</td>
<td>×</td>
<td>3744.9</td>
<td>3768.6</td>
<td>4.4</td>
</tr>
</tbody>
</table>

<sup>av</sup> marks query avalanche. All times in milliseconds.
<table>
<thead>
<tr>
<th>Q#</th>
<th>F# 3.0</th>
<th>us</th>
<th>(norm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>2.0</td>
<td>2.4</td>
<td>0.3</td>
</tr>
<tr>
<td>Q2</td>
<td>1.5</td>
<td>1.7</td>
<td>0.2</td>
</tr>
<tr>
<td>Q5</td>
<td>1.7</td>
<td>2.1</td>
<td>0.3</td>
</tr>
<tr>
<td>Q6</td>
<td>1.7</td>
<td>2.1</td>
<td>0.3</td>
</tr>
<tr>
<td>Q7</td>
<td>1.5</td>
<td>1.8</td>
<td>0.2</td>
</tr>
<tr>
<td>Q8</td>
<td>2.3</td>
<td>2.4</td>
<td>0.2</td>
</tr>
<tr>
<td>Q9</td>
<td>2.3</td>
<td>2.7</td>
<td>0.3</td>
</tr>
<tr>
<td>Q10</td>
<td>1.4</td>
<td>1.7</td>
<td>0.2</td>
</tr>
<tr>
<td>Q11</td>
<td>1.4</td>
<td>1.7</td>
<td>0.2</td>
</tr>
<tr>
<td>Q12</td>
<td>4.4</td>
<td>4.9</td>
<td>0.4</td>
</tr>
<tr>
<td>Q13</td>
<td>2.5</td>
<td>2.9</td>
<td>0.4</td>
</tr>
<tr>
<td>Q14</td>
<td>2.5</td>
<td>2.9</td>
<td>0.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q#</th>
<th>F# 3.0</th>
<th>us</th>
<th>(norm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q15</td>
<td>3.5</td>
<td>4.0</td>
<td>0.5</td>
</tr>
<tr>
<td>Q16</td>
<td>3.5</td>
<td>4.0</td>
<td>0.5</td>
</tr>
<tr>
<td>Q17</td>
<td>6.2</td>
<td>6.7</td>
<td>0.4</td>
</tr>
<tr>
<td>Q18</td>
<td>1.5</td>
<td>1.8</td>
<td>0.2</td>
</tr>
<tr>
<td>Q19</td>
<td>1.5</td>
<td>1.8</td>
<td>0.2</td>
</tr>
<tr>
<td>Q20</td>
<td>1.5</td>
<td>1.8</td>
<td>0.2</td>
</tr>
<tr>
<td>Q21</td>
<td>1.6</td>
<td>1.9</td>
<td>0.3</td>
</tr>
<tr>
<td>Q22</td>
<td>1.6</td>
<td>1.9</td>
<td>0.3</td>
</tr>
<tr>
<td>Q23</td>
<td>1.6</td>
<td>1.9</td>
<td>0.3</td>
</tr>
<tr>
<td>Q24</td>
<td>1.8</td>
<td>2.0</td>
<td>0.3</td>
</tr>
<tr>
<td>Q25</td>
<td>1.4</td>
<td>1.6</td>
<td>0.2</td>
</tr>
<tr>
<td>Q27</td>
<td>1.8</td>
<td>2.1</td>
<td>0.2</td>
</tr>
<tr>
<td>Q#</td>
<td>F# 3.0</td>
<td>us</td>
<td>(norm)</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>-----</td>
<td>--------</td>
</tr>
<tr>
<td>Q29</td>
<td>1.5</td>
<td>1.7</td>
<td>0.2</td>
</tr>
<tr>
<td>Q30</td>
<td>1.8</td>
<td>2.0</td>
<td>0.2</td>
</tr>
<tr>
<td>Q32</td>
<td>2.7</td>
<td>3.1</td>
<td>0.3</td>
</tr>
<tr>
<td>Q33</td>
<td>2.8</td>
<td>3.1</td>
<td>0.3</td>
</tr>
<tr>
<td>Q34</td>
<td>3.1</td>
<td>3.6</td>
<td>0.5</td>
</tr>
<tr>
<td>Q35</td>
<td>3.1</td>
<td>3.6</td>
<td>0.4</td>
</tr>
<tr>
<td>Q36</td>
<td>2.2</td>
<td>2.4</td>
<td>0.2</td>
</tr>
<tr>
<td>Q37</td>
<td>1.3</td>
<td>1.6</td>
<td>0.2</td>
</tr>
<tr>
<td>Q38</td>
<td>4.2</td>
<td>4.9</td>
<td>0.6</td>
</tr>
<tr>
<td>Q39</td>
<td>4.2</td>
<td>4.7</td>
<td>0.4</td>
</tr>
<tr>
<td>Q40</td>
<td>4.1</td>
<td>4.6</td>
<td>0.4</td>
</tr>
<tr>
<td>Q41</td>
<td>6.3</td>
<td>7.3</td>
<td>0.6</td>
</tr>
<tr>
<td>Q42</td>
<td>4.7</td>
<td>5.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Q43</td>
<td>7.2</td>
<td>6.9</td>
<td>0.7</td>
</tr>
<tr>
<td>Q44</td>
<td>5.4</td>
<td>6.2</td>
<td>0.7</td>
</tr>
<tr>
<td>Q45</td>
<td>2.2</td>
<td>2.6</td>
<td>0.3</td>
</tr>
<tr>
<td>Q46</td>
<td>2.3</td>
<td>2.7</td>
<td>0.4</td>
</tr>
<tr>
<td>Q47</td>
<td>2.1</td>
<td>2.5</td>
<td>0.3</td>
</tr>
<tr>
<td>Q48</td>
<td>2.1</td>
<td>2.5</td>
<td>0.3</td>
</tr>
<tr>
<td>Q49</td>
<td>2.4</td>
<td>2.7</td>
<td>0.3</td>
</tr>
<tr>
<td>Q50</td>
<td>2.2</td>
<td>2.5</td>
<td>0.3</td>
</tr>
<tr>
<td>Q51</td>
<td>2.0</td>
<td>2.4</td>
<td>0.3</td>
</tr>
<tr>
<td>Q52</td>
<td>6.1</td>
<td>5.9</td>
<td>0.4</td>
</tr>
<tr>
<td>Q53</td>
<td>11.9</td>
<td>11.2</td>
<td>0.6</td>
</tr>
</tbody>
</table>
Comparison of F# 3.0 and us (using F# 3.0 as a back-end) on the 62 example database queries in the F# 3.0 documentation. Five query expressions (Q3, Q4, Q26, Q28, Q31) are excluded because they are executed on in-memory lists rather than generating SQL. All times in milliseconds.

<table>
<thead>
<tr>
<th>Q#</th>
<th>F# 3.0</th>
<th>us</th>
<th>(norm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q54</td>
<td>4.4</td>
<td>4.8</td>
<td>0.4</td>
</tr>
<tr>
<td>Q55</td>
<td>5.2</td>
<td>5.6</td>
<td>0.4</td>
</tr>
<tr>
<td>Q56</td>
<td>4.6</td>
<td>5.1</td>
<td>0.5</td>
</tr>
<tr>
<td>Q57</td>
<td>2.5</td>
<td>2.9</td>
<td>0.4</td>
</tr>
<tr>
<td>Q58</td>
<td>2.5</td>
<td>2.9</td>
<td>0.4</td>
</tr>
<tr>
<td>Q59</td>
<td>3.1</td>
<td>3.6</td>
<td>0.5</td>
</tr>
<tr>
<td>Q60</td>
<td>3.6</td>
<td>4.4</td>
<td>0.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q#</th>
<th>F# 3.0</th>
<th>us</th>
<th>(norm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q61</td>
<td>5.8</td>
<td>6.3</td>
<td>0.3</td>
</tr>
<tr>
<td>Q62</td>
<td>5.4</td>
<td>5.9</td>
<td>0.2</td>
</tr>
<tr>
<td>Q63</td>
<td>3.4</td>
<td>3.8</td>
<td>0.4</td>
</tr>
<tr>
<td>Q64</td>
<td>4.3</td>
<td>4.9</td>
<td>0.6</td>
</tr>
<tr>
<td>Q65</td>
<td>10.2</td>
<td>10.1</td>
<td>0.4</td>
</tr>
<tr>
<td>Q66</td>
<td>8.9</td>
<td>8.7</td>
<td>0.6</td>
</tr>
<tr>
<td>Q67</td>
<td>14.7</td>
<td>13.1</td>
<td>1.1</td>
</tr>
</tbody>
</table>
Goals

Series of examples
  Join queries
  Abstraction over values (first-order)
  Abstraction over predicates (higher-order)
  Composition of queries
  Dynamic generation of queries
  Nested intermediate data

Type safety

Avoid Scylla and Charybdis
  Each host query generates one SQL query
  *Scylla*: failure to generate a query
  *Charybdis*: multiple queries, avalanche
Theory and Practice

*Host and quoted languages*

Theory: different (recursion, database)
Practice: identical

*Coverage*

Theory: doesn’t cover sorting, grouping, aggregation—work for tomorrow
Practice: covers all of LINQ—put it to work today
What is the difference between theory and practice?

In theory there is a difference.
But in practice there isn’t.
Daddy as a lambda man
Appendix A7

Problems with F#
Problems with F# PowerPack

(Notes from James Cheney)

Problems fixed in F# PowerPack code:

- F# 2.0/PowerPack lacked support for singletons in nonstandard places (i.e. other than in a comprehension body).

- F# 2.0/PowerPack also lacked support for Seq.exists in certain places because it was assuming that expressions of base types (eg. booleans) did not need to be further translated.

F# 3.0:

- Did not exhibit the above problems

- But did exhibit translation bug where something like

  query if 1 = 2 then yield 3

  leads to a run-time type error.
Appendix A7

From XPath to SQL
Representing XML

type Node =
    {id: int, parent: int, name: string, pre: int, post: int}
Abstract syntax of XPath

**type** Axis =

- Self
- Child
- Descendant
- DescendantOrSelf
- Following
- FollowingSibling
- Rev of Axis

**type** Path =

- Seq of Path × Path
- Axis of Axis
- NameTest of string
- Filter of Path
An evaluator for XPath: axis

```
let rec axis(ax : Axis) : Expr<(Node, Node) → bool> =

match ax with
| Self → @ fun(s, t) → s.id = t.id @>
| Child → @ fun(s, t) → s.id = t.parent @>
| Descendant → @ fun(s, t) →
    s.pre < t.pre && t.post < s.post @>
| DescendantOrSelf → @ fun(s, t) →
    s.pre ≤ t.pre && t.post ≤ s.post @>
| Following → @ fun(s, t) → s.pre < t.pre @>
| FollowingSibling → @ fun(s, t) →
    s.post < t.pre && s.parent = t.parent @>
| Rev(axis) → @ fun(s, t) → (%axis(ax))(t, s) @>
```
An evaluator for XPath: path

```
let rec path(p : Path) : Expr<(Node, Node) -> bool> =
match p with
| Seq(p, q) -> <@ fun(s, u) -> (%any)((%db).xml, fun(t) -> (%path(p))(s, t) && (%path(q))(t, u)) @>
| Axis(ax) -> axis(ax)
| NameTest(name) -> <@ fun(s, t) -> s.id = t.id && s.name = name @>
| Filter(p) -> <@ fun(s, t) -> s.id = t.id && (%any)((%db).xml, fun(u) -> (%path(p))(s, u)) @>
```
An evaluator for XPath: xpath

```plaintext
let xpath(p : Path) : Expr<Node list> =
  <@
  for root in (%db).xml do
    for s in (%db).xml do
      if root.parent = -1 && (%path(p))(root, s) then
        yield s @>
```
Examples

```xml
/* */

run(xpath(Seq(Axis(Child), Axis(Child))))
[2; 4]

/* */ [following-sibling::d]
run(xpath(Seq(Axis(Descendant),
    Filter(Seq(Axis(FollowingSibling),
        NameTest("d"))))))
[2]
```