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

SHARKS; Dogs of Scylla.

BRITANNIA between SCYLLA & CHARYBDIS.

or — *The Vessel of the Constitution steered clear of the Rock of Democracy, and the Whirlpool of Arbitary-Power.* —

Pub.d April 10th 1793 by H. Humphrey No 18 Old Bond Street

J.s G.y des.n et fec.t pro bono publico —


Somers

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

people

name	age
“Alex”	60
“Bert”	56
“Cora”	33
“Drew”	31
“Edna”	21
“Fred”	60

couples

her	him
“Alex”	“Bert”
“Cora”	“Drew”
“Edna”	“Fred”

A query in 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

name	diff
"Alex"	4
"Cora"	2

A database as data

$$
\begin{aligned}
\{&\mathsf{people} = \\
&\quad [\{\mathsf{name} = \text{“Alex”}\ ;\ \mathsf{age} = 60\}; \\
&\quad \ \ \{\mathsf{name} = \text{“Bert”}\ ;\ \mathsf{age} = 56\}; \\
&\quad \ \ \{\mathsf{name} = \text{“Cora”};\ \mathsf{age} = 33\}; \\
&\quad \ \ \{\mathsf{name} = \text{“Drew”};\ \mathsf{age} = 31\}; \\
&\quad \ \ \{\mathsf{name} = \text{“Edna”};\ \mathsf{age} = 21\}; \\
&\quad \ \ \{\mathsf{name} = \text{“Fred”}\ ;\ \mathsf{age} = 60\}]; \\
&\mathsf{couples} = \\
&\quad [\{\mathsf{her} = \text{“Alex”}\ ;\ \mathsf{him} = \text{“Bert”}\ \}; \\
&\quad \ \ \{\mathsf{her} = \text{“Cora”};\ \mathsf{him} = \text{“Drew”}\}; \\
&\quad \ \ \{\mathsf{her} = \text{“Edna”};\ \mathsf{him} = \text{“Fred”}\ \}]\}
\end{aligned}
$$

Importing the database (naive)

$$
\begin{array}{l}
\textbf{type}\ \mathsf{DB} = \\
\quad \{\mathsf{people} : \\
\qquad \{\mathsf{name} : \textbf{string};\ \mathsf{age} : \textbf{int}\}\ \textbf{list}; \\
\quad\ \ \mathsf{couples} : \\
\qquad \{\mathsf{her} : \textbf{string};\ \mathsf{him} : \textbf{string}\}\ \textbf{list}\} \\
\textbf{let}\ \mathsf{db}' : \mathsf{DB} = \textbf{database}(\text{“People”})
\end{array}
$$

A query as a comprehension (naive)

$$
\begin{array}{l}
\textbf{let}\ \mathsf{differences}' : \{\mathsf{name} : \textbf{string};\ \mathsf{diff} : \textbf{int}\}\ \textbf{list} = \\
\quad \textbf{for}\ \mathsf{c}\ \textbf{in}\ \mathsf{db}'.\mathsf{couples}\ \textbf{do} \\
\quad \textbf{for}\ \mathsf{w}\ \textbf{in}\ \mathsf{db}'.\mathsf{people}\ \textbf{do} \\
\quad \textbf{for}\ \mathsf{m}\ \textbf{in}\ \mathsf{db}'.\mathsf{people}\ \textbf{do} \\
\quad \textbf{if}\ \mathsf{c.her} = \mathsf{w.name}\ \ \&\&\ \ \mathsf{c.him} = \mathsf{m.name}\ \ \&\&\ \ \mathsf{w.age} > \mathsf{m.age}\ \textbf{then} \\
\quad \textbf{yield}\ \{\mathsf{name} : \mathsf{w.name};\ \mathsf{diff} : \mathsf{w.age} - \mathsf{m.age}\}
\end{array}
$$

differences'

$$
\begin{array}{l}
[\,\{\mathsf{name} = \text{“Alex”};\ \mathsf{diff} = 4\} \\
\ \ \{\mathsf{name} = \text{“Cora”};\ \mathsf{diff} = 2\}\,]
\end{array}
$$

Importing the database (quoted)

```
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)

```
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

```
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

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

```
run(<@ (%satisfies)(fun(x) → 30 ≤ x && x < 40) @>)
```

[{name = “Cora”}; {name = “Drew”}]

```
run(<@ (%satisfies)(fun(x) → x mod 2 = 0) @>)
```

[{name = “Alex”}; {name = “Bert”}; {name = “Fred”}]

Composing queries

```
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 $=$

$\mid$ Above **of int**

$\mid$ Below **of int**

$\mid$ And **of** Predicate $\times$ Predicate

$\mid$ Or **of** Predicate $\times$ Predicate

$\mid$ 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)

```
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)

$$\mathsf{P}(t_0)$$

$$\texttt{<@}\ \ \textbf{fun}(x) \to (\textbf{fun}(x) \to 30 \leq x)(x)\ \ \&\&\ \ (\textbf{fun}(x) \to x < 40)(x)\ \ \texttt{@>}$$

$$\texttt{<@}\ \ \textbf{fun}(x) \to 30 \leq x\ \ \&\&\ \ x < 40\ \ \texttt{@>}$$

$$\textbf{run}(\texttt{<@}\ (\%\mathsf{satisfies})(\%\mathsf{P}(t_0))\ \texttt{@>})$$

$$[\{\mathsf{name} = \text{“Cora”}\};\ \{\mathsf{name} = \text{“Drew”}\}]$$

$$\textbf{run}(\texttt{<@}\ (\%\mathsf{satisfies})(\%\mathsf{P}(t_1))\ \texttt{@>})$$

$$[\{\mathsf{name} = \text{“Cora”}\};\ \{\mathsf{name} = \text{“Drew”}\}]$$

Part III

Nesting

Flat data

$$
\begin{aligned}
\{&\text{departments} = \\
&\quad [\{\text{dpt} = \text{“Product”}\}; \\
&\quad\ \ \{\text{dpt} = \text{“Quality”}\}; \\
&\quad\ \ \{\text{dpt} = \text{“Research”}\}; \\
&\quad\ \ \{\text{dpt} = \text{“Sales”}\}]; \\
&\text{employees} = \\
&\quad [\{\text{dpt} = \text{“Product”}; \text{emp} = \text{“Alex”}\}; \\
&\quad\ \ \{\text{dpt} = \text{“Product”}; \text{emp} = \text{“Bert”}\}; \\
&\quad\ \ \{\text{dpt} = \text{“Research”}; \text{emp} = \text{“Cora”}\}; \\
&\quad\ \ \{\text{dpt} = \text{“Research”}; \text{emp} = \text{“Drew”}\}; \\
&\quad\ \ \{\text{dpt} = \text{“Research”}; \text{emp} = \text{“Edna”}\}; \\
&\quad\ \ \{\text{dpt} = \text{“Sales”}; \text{emp} = \text{“Fred”}\}];
\end{aligned}
$$

Flat data (continued)

$$
\begin{aligned}
&\mathsf{tasks} = \\
&\quad [\{\mathsf{emp} = \text{“Alex”}; \mathsf{tsk} = \text{“build”}\}; \\
&\quad\ \{\mathsf{emp} = \text{“Bert”}; \mathsf{tsk} = \text{“build”}\}; \\
&\quad\ \{\mathsf{emp} = \text{“Cora”}; \mathsf{tsk} = \text{“abstract”}\}; \\
&\quad\ \{\mathsf{emp} = \text{“Cora”}; \mathsf{tsk} = \text{“build”}\}; \\
&\quad\ \{\mathsf{emp} = \text{“Cora”}; \mathsf{tsk} = \text{“design”}\}; \\
&\quad\ \{\mathsf{emp} = \text{“Drew”}; \mathsf{tsk} = \text{“abstract”}\}; \\
&\quad\ \{\mathsf{emp} = \text{“Drew”}; \mathsf{tsk} = \text{“design”}\}; \\
&\quad\ \{\mathsf{emp} = \text{“Edna”}; \mathsf{tsk} = \text{“abstract”}\}; \\
&\quad\ \{\mathsf{emp} = \text{“Edna”}; \mathsf{tsk} = \text{“call”}\}; \\
&\quad\ \{\mathsf{emp} = \text{“Edna”}; \mathsf{tsk} = \text{“design”}\}; \\
&\quad\ \{\mathsf{emp} = \text{“Fred”}; \mathsf{tsk} = \text{“call”}\}]\}
\end{aligned}
$$

Importing the database

$$
\begin{aligned}
&\textbf{type}\ \mathsf{Org} = \{\mathsf{departments} : \{\mathsf{dpt} : \textbf{string}\}\ \textbf{list};\\
&\qquad\qquad\quad \mathsf{employees} : \ \{\mathsf{dpt} : \textbf{string}; \mathsf{emp} : \textbf{string}\}\ \textbf{list};\\
&\qquad\qquad\quad \mathsf{tasks} : \qquad\ \{\mathsf{emp} : \textbf{string}; \mathsf{tsk} : \textbf{string}\}\ \textbf{list}\ \}\\
&\textbf{let}\ \mathsf{org} : \mathsf{Expr}\!<\mathsf{Org}> = \texttt{<@}\ \textbf{database}(\text{“Org”})\ \texttt{@>}
\end{aligned}
$$

Departments where every employee can do a given task

```
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

[{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

```
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

```
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((%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

```
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

```
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

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

vs.

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

VAR

$$\frac{}{\Gamma, x : A \vdash x : A}$$

FUN

$$\frac{\Gamma, x : A \vdash N : B}{\Gamma \vdash \textbf{fun}(x) \to N : A \to B}$$

APP

$$\frac{\Gamma \vdash L : A \to B \qquad \Gamma \vdash M : A}{\Gamma \vdash L\ M : B}$$

SINGLETON

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \textbf{yield}\ M : A\ \textbf{list}}$$

FOR

$$\frac{\Gamma \vdash M : A\ \textbf{list} \qquad \Gamma, x : A \vdash N : B\ \textbf{list}}{\Gamma \vdash \textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ N : B\ \textbf{list}}$$

REC

$$\frac{\Gamma, f : A \to B, x : A \vdash N : B}{\Gamma \vdash \textbf{rec}\ f(x) \to N : A \to B}$$

Quoted terms

VarQ

$$\frac{}{\Gamma; \Delta, x : A \vdash x : A}$$

FunQ

$$\frac{\Gamma; \Delta, x : A \vdash N : B}{\Gamma; \Delta \vdash \textbf{fun}(x) \to N : A \to B}$$

AppQ

$$\frac{\Gamma; \Delta \vdash L : A \to B \qquad \Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash L\ M : B}$$

SingletonQ

$$\frac{\Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash \textbf{yield}\ M : A\ \textbf{list}}$$

ForQ

$$\frac{\Gamma; \Delta \vdash M : A\ \textbf{list} \qquad \Gamma; \Delta, x : A \vdash N : B\ \textbf{list}}{\Gamma; \Delta \vdash \textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ N : B\ \textbf{list}}$$

Database

$$\frac{\Sigma(\mathsf{db}) = \{\overline{\ell : T}\}}{\Gamma; \Delta \vdash \textbf{database}(\mathsf{db}) : \{\overline{\ell : T}\}}$$

Quotation and anti-quotation

QUOTE

$$\frac{\Gamma; \cdot \vdash M : A}{\Gamma \vdash \texttt{<@}\ M\ \texttt{@>} : \textsf{Expr}< A >}$$

ANTIQUOTE

$$\frac{\Gamma \vdash M : \textsf{Expr}< A >}{\Gamma; \Delta \vdash (\%M) : A}$$

RUN

$$\frac{\Gamma \vdash M : \textsf{Expr}< T >}{\Gamma \vdash \textbf{run}(M) : T}$$

LIFT

$$\frac{\Gamma \vdash M : O}{\Gamma \vdash \textbf{lift}(M) : \textsf{Expr}< O >}$$

Normalisation: symbolic evaluation

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

Normalisation: *ad hoc* rewriting

$$\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ (M\ \texttt{@}\ N) \hookrightarrow (\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ M)\ \texttt{@}\ (\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ N)$$

$$\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ \texttt{[ ]} \hookrightarrow \texttt{[ ]}$$

$$\textbf{if}\ L\ \textbf{then}\ (M\ \texttt{@}\ N) \hookrightarrow (\textbf{if}\ L\ \textbf{then}\ M)\ \texttt{@}\ (\textbf{if}\ L\ \textbf{then}\ N)$$

$$\textbf{if}\ L\ \textbf{then}\ \texttt{[ ]} \hookrightarrow \texttt{[ ]}$$

$$\textbf{if}\ L\ \textbf{then}\ (\textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ N) \hookrightarrow \textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ (\textbf{if}\ L\ \textbf{then}\ N)$$

$$\textbf{if}\ L\ \textbf{then}\ (\textbf{if}\ M\ \textbf{then}\ N) \hookrightarrow \textbf{if}\ (L\ \ \&\&\ \ M)\ \textbf{then}\ N$$

Properties of reduction

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

- *preserve* typing,
- are *stongly 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)
*/
def dbfor[A,B](l: Exp[List[A]], f: Exp[A] => Exp[List[B]])
  = l match {
    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 $\rightsquigarrow$

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 $\leq$ w.age && w.age $<$ b **then**

yield $\{$name : w.name$\}$

Example (4): other rewriting $\rightsquigarrow$

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 $\leq$ w.age && w.age $<$ v.age **then**

yield $\{$name : w.name$\}$

Example (5): *ad hoc* reductions $\hookrightarrow$

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 $\leq$ w.age && w.age $<$ v.age **then**
yield $\{$name : w.name$\}$

Example (6): 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 $\leq$ w.age **and** w.age $<$ v.age

Part VII

Results

Example	F# 2.0	F# 3.0	us	(norm)
differences	17.6	20.6	18.1	0.5
range	×	5.6	2.9	0.3
satisfies	2.6	×	2.9	0.3
satisfies	4.4	×	4.6	0.3
compose	×	×	4.0	0.8
$P(t_0)$	2.8	×	3.3	0.3
$P(t_1)$	2.7	×	3.0	0.3
expertise′	7.2	9.2	8.0	0.6
expertise	×	66.7[av]	8.3	0.9
xp_0	×	8.3	7.9	1.9
xp_1	×	14.7	13.4	1.1
xp_2	×	17.9	20.7	2.2
xp_3	×	3744.9	3768.6	4.4

[av] marks query avalanche. All times in milliseconds.

Q#	F# 3.0	us	(norm)
Q1	2.0	2.4	0.3
Q2	1.5	1.7	0.2
Q5	1.7	2.1	0.3
Q6	1.7	2.1	0.3
Q7	1.5	1.8	0.2
Q8	2.3	2.4	0.2
Q9	2.3	2.7	0.3
Q10	1.4	1.7	0.2
Q11	1.4	1.7	0.2
Q12	4.4	4.9	0.4
Q13	2.5	2.9	0.4
Q14	2.5	2.9	0.3

Q#	F# 3.0	us	(norm)
Q15	3.5	4.0	0.5
Q16	3.5	4.0	0.5
Q17	6.2	6.7	0.4
Q18	1.5	1.8	0.2
Q19	1.5	1.8	0.2
Q20	1.5	1.8	0.2
Q21	1.6	1.9	0.3
Q22	1.6	1.9	0.3
Q23	1.6	1.9	0.3
Q24	1.8	2.0	0.3
Q25	1.4	1.6	0.2
Q27	1.8	2.1	0.2

Q#	F# 3.0	us	(norm)
Q29	1.5	1.7	0.2
Q30	1.8	2.0	0.2
Q32	2.7	3.1	0.3
Q33	2.8	3.1	0.3
Q34	3.1	3.6	0.5
Q35	3.1	3.6	0.4
Q36	2.2	2.4	0.2
Q37	1.3	1.6	0.2
Q38	4.2	4.9	0.6
Q39	4.2	4.7	0.4
Q40	4.1	4.6	0.4
Q41	6.3	7.3	0.6

Q#	F# 3.0	us	(norm)
Q42	4.7	5.5	0.5
Q43	7.2	6.9	0.7
Q44	5.4	6.2	0.7
Q45	2.2	2.6	0.3
Q46	2.3	2.7	0.4
Q47	2.1	2.5	0.3
Q48	2.1	2.5	0.3
Q49	2.4	2.7	0.3
Q50	2.2	2.5	0.3
Q51	2.0	2.4	0.3
Q52	6.1	5.9	0.4
Q53	11.9	11.2	0.6

Q#	F# 3.0	us	(norm)
Q54	4.4	4.8	0.4
Q55	5.2	5.6	0.4
Q56	4.6	5.1	0.5
Q57	2.5	2.9	0.4
Q58	2.5	2.9	0.4
Q59	3.1	3.6	0.5
Q60	3.6	4.4	0.7

Q#	F# 3.0	us	(norm)
Q61	5.8	6.3	0.3
Q62	5.4	5.9	0.2
Q63	3.4	3.8	0.4
Q64	4.3	4.9	0.6
Q65	10.2	10.1	0.4
Q66	8.9	8.7	0.6
Q67	14.7	13.1	1.1

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.

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.


Deddy
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

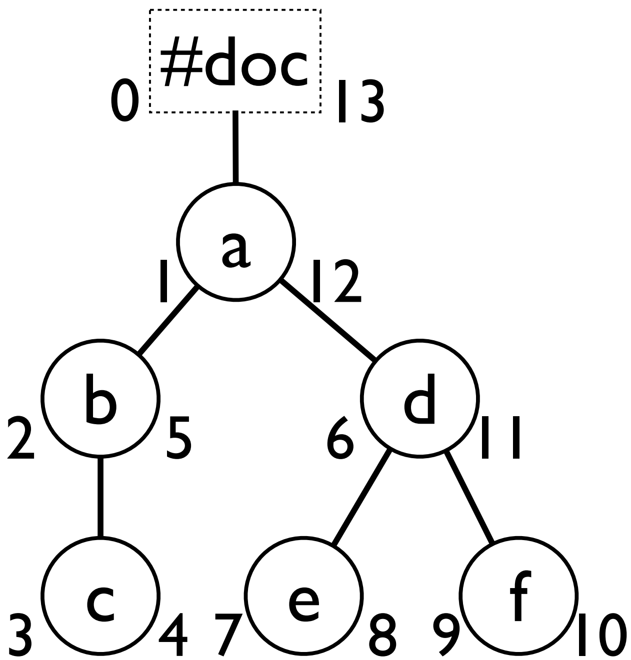
  ```
  query if 1 = 2 then yield 3
  ```

  leads to a run-time type error.

Appendix A7

From XPath to SQL

Representing XML



xml

id	parent	name	pre	post
0	-1	#doc	0	13
1	0	a	1	12
2	1	b	2	5
3	2	c	3	4
4	1	d	6	11
5	4	e	7	8
6	4	f	9	10

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 $\times$ 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) @>
```

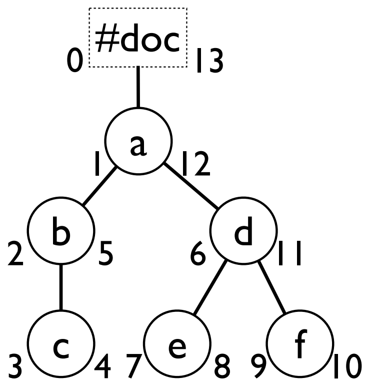
An evaluator for XPath: path

```
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

```
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



$/ * / *$

run(xpath(Seq(Axis(Child), Axis(Child))))

$[2;\ 4]$

$//*[$following-sibling::d$]$

run(xpath(Seq(Axis(Descendant),
Filter(Seq(Axis(FollowingSibling),
NameTest("d"))))))

$[2]$