A Practical Theory of Language-Integrated Query

James Cheney, Sam Lindley, Philip Wadler
University of Edinburgh

Yow!, MPLW, FP with the Stars, SAPLING
Melbourne, Brisbane, Sydney
December 2013

What is the difference between
theory and practice?

In theory there
is no difference.
But in practice
there is.

How does one integrate SQL
and a host language?

How does one integrate SQL
and a host language?

How does one integrate a Domain-Specific Language
and a host language?

Domain-Specific Language (DSL)
Domain-Specific Embedded Language (DSEL)

A functional language is a

Domain-Specific Language

for defining

Domain-Specific Languages

Links

Wadler, Yallop, Lindley, Cooper

(Edinburgh)

LINQ

Meijer (C#,VB), Syme (F#)

(Microsoft)

λ

Links

Wadler, Yallop, Lindley, Cooper

(Edinburgh)

λ

LINQ

Meijer (C#,VB), Syme (F#)

(Microsoft)

Links

Wadler, Yallop, Lindley, Cooper

(Edinburgh)

λ

LINQ

Meijer (C#,VB), Syme (F#)

(Microsoft)

λ

Scylla and Charybdis

Avoid Scylla and Charybdis

Each host query generates one SQL query

Scylla: failure to generate a query ($\times$)

Charybdis: multiple queries, avalanche (^{av})

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

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

Series of examples

Join queries

Abstraction over values (first-order)

Abstraction over predicates (higher-order)

Dynamic generation of queries

Nested intermediate data

Compiling XPath to SQL

Closed quotation vs. open quotation

$\mathsf{Expr}< A \to B >$ *vs.* $\mathsf{Expr}< A > \to \mathsf{Expr}< B >$

T-LINQ: the theory

Scylla and Charybdis Theorem

P-LINQ: the practice

Measured times comparable

Normalisation a small fraction of time

Part I

Join queries

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

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

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)

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

$$\text{differences}'$$

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

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

```
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) @>)
```

```
select w.name as name
from people as w
where 30 ≤ w.age and w.age < 40
```

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) @>)
```

```
select w.name as name
from people as w
where 30 ≤ w.age and w.age < 40
```

Datatype of predicates

$$\textbf{type}\ \text{Predicate} =$$
$$\quad |\ \text{Above}\ \textbf{of int}$$
$$\quad |\ \text{Below}\ \textbf{of int}$$
$$\quad |\ \text{And}\ \textbf{of}\ \text{Predicate} \times \text{Predicate}$$
$$\quad |\ \text{Or}\ \textbf{of}\ \text{Predicate} \times \text{Predicate}$$
$$\quad |\ \text{Not}\ \textbf{of}\ \text{Predicate}$$

$$\textbf{let}\ \text{t}_0 : \text{Predicate} = \text{And}(\text{Above}(30), \text{Below}(40))$$

Dynamically generated queries

```
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)) @>
```

Generating the query

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

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

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

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

select w.name **as** name

from people **as** w

where $30 \leq$ w.age **and** w.age < 40

Part III

Nested intermediate data

Flat data

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

Flat data (continued)

$$\text{tasks} =$$

$$[\{\text{emp} = \text{“Alex”}; \text{tsk} = \text{“build”}\};$$
$$\{\text{emp} = \text{“Bert”}; \text{tsk} = \text{“build”}\};$$
$$\{\text{emp} = \text{“Cora”}; \text{tsk} = \text{“abstract”}\};$$
$$\{\text{emp} = \text{“Cora”}; \text{tsk} = \text{“build”}\};$$
$$\{\text{emp} = \text{“Cora”}; \text{tsk} = \text{“design”}\};$$
$$\{\text{emp} = \text{“Drew”}; \text{tsk} = \text{“abstract”}\};$$
$$\{\text{emp} = \text{“Drew”}; \text{tsk} = \text{“design”}\};$$
$$\{\text{emp} = \text{“Edna”}; \text{tsk} = \text{“abstract”}\};$$
$$\{\text{emp} = \text{“Edna”}; \text{tsk} = \text{“call”}\};$$
$$\{\text{emp} = \text{“Edna”}; \text{tsk} = \text{“design”}\};$$
$$\{\text{emp} = \text{“Fred”}; \text{tsk} = \text{“call”}\}]\}$$

Importing the database

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

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

$$
\begin{aligned}
&[\{\mathsf{dpt} = \text{“Product”};\ \mathsf{employees} = \\
&\quad [\{\mathsf{emp} = \text{“Alex”};\ \mathsf{tasks} = [\text{“build”}]\} \\
&\quad \{\mathsf{emp} = \text{“Bert”};\ \mathsf{tasks} = [\text{“build”}]\}]\}; \\
&\{\mathsf{dpt} = \text{“Quality”}; \mathsf{employees} = [\,]\}; \\
&\{\mathsf{dpt} = \text{“Research”};\ \mathsf{employees} = \\
&\quad [\{\mathsf{emp} = \text{“Cora”};\ \mathsf{tasks} = [\text{“abstract”}; \text{“build”}; \text{“design”}]\}; \\
&\quad \{\mathsf{emp} = \text{“Drew”};\ \mathsf{tasks} = [\text{“abstract”}; \text{“design”}]\}; \\
&\quad \{\mathsf{emp} = \text{“Edna”};\ \mathsf{tasks} = [\text{“abstract”}; \text{“call”}; \text{“design”}]\}]\}; \\
&\{\mathsf{dpt} = \text{“Sales”};\ \mathsf{employees} = \\
&\quad [\{\mathsf{emp} = \text{“Fred”};\ \mathsf{tasks} = [\text{“call”}]\}]\}]
\end{aligned}
$$

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

Compiling XPath to SQL

Part V

Closed quotation vs. open quotation

Dynamically generated queries, revisited

```
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) @>
```

vs.

```
let rec P(t : Predicate)(x : Expr< int >) : Expr< bool > =
  match t with
  | Above(a)→ <@ (%lift(a)) ≤ (%x) @>
  | Below(a)→ <@ (%x) < (%lift(a)) @>
  | And(t, u) → <@ (%P(t)(x))  &&  (%P(u)(x)) @>
```

Abstracting over a predicate, revisited

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

vs.

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

closed quotations

vs.

open quotations

quotations of functions

$$(\mathsf{Expr}\!<\mathsf{A} \longrightarrow \mathsf{B}>)$$

vs.

functions of quotations

$$(\mathsf{Expr}\!<\mathsf{A}> \longrightarrow \mathsf{Expr}\!<\mathsf{B}>)$$

Part VI

T-LINQ: the theory

Host language

FUN

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

APP

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

SINGLETON

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

FOR

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

QUOTE

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

RUN

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

REC

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

Quoted language

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

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

Antiquote

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

Lift

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

Database

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

Normalisation: symbolic evaluation

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

Normalisation: *ad hoc* rewriting

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

Theorem *(Scylla and Charybdis)* If

$$\vdash L : A$$

and A is a table type (list of record of scalars) then

$$L \leadsto^* M \quad \text{and} \quad M \hookrightarrow^* N,$$

where M and N are in normal form with respect to $\leadsto$ and $\hookrightarrow$, and N is isomorphic to an SQL query.

Part VII

P-LINQ: the practice

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

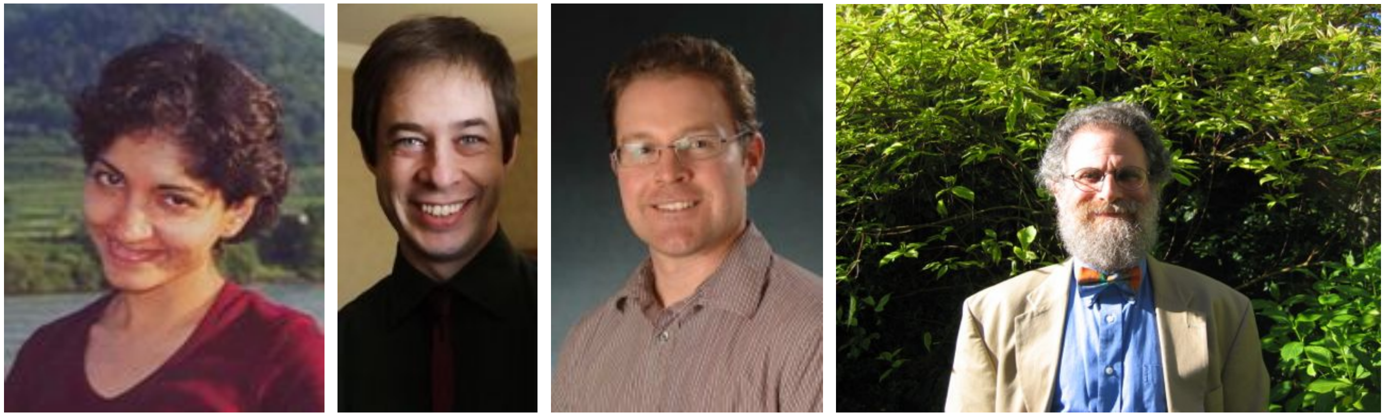
Q#	F# 3.0	us	(norm)
Q61	5.8	6.3	0.3
Q62	5.4	5.9	0.2
Q63	3.4	3.8	0.4
Q64	4.3	4.9	0.6
Q65	10.2	10.1	0.4
Q66	8.9	8.7	0.6
Q67	14.7	13.1	1.1

All times in milliseconds.

Part VIII

What else are we up to?

Blame: Integrating static and dynamic typing



Ahmed, Findler, Siek, Wadler

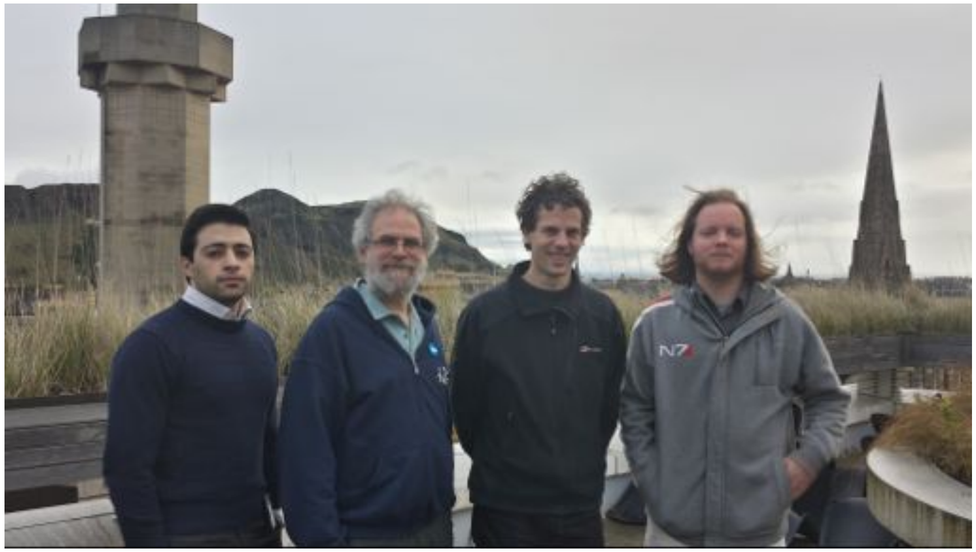
- Well-typed programs can't be blamed, ESOP 2009.
- Threesomes, with and without blame, POPL 2010.
- Blame for all, POPL 2011.
- A plague on both your houses: Allocating blame symmetrically and precisely 2013, to appear.

Links: Web programming without tiers

Wadler, Yallop, Lindley, Cooper

- Links: Web programming without tiers, FMCO 2006.
- The essence of form abstraction, ASPLAS 2008.
  F# (WebSharper), Haskell (Tupil, Digestive Functors, Happstack, Yesod), Common Lisp, JavaScript, Racket, Scala.
- Idioms are Oblivious, Arrows are Meticulous, Monads are Promiscuous MSFP 2008.
- The arrow calculus, JFP 2010.

ABCD: A Basis for Concurrency and Distribution



Najd, Wadler, Lindley, Morris

- From Session Types to Data Types: A Basis for Concurrency and Distribution, EPSRC 2013–2018.
- Co-PIs: Simon Gay, Glasgow, and Nobuko Yoshida, Imperial. Collaborators: Amazon, Cognizant, OOI, Red Hat, VMWare.
- Propositions as Sessions, ICFP 2012, JFP 2014.
- A practical theory of language-integrated query, ICFP 2013.

Part IX

Conclusion

Series of examples

Join queries

Abstraction over values (first-order)

Abstraction over predicates (higher-order)

Dynamic generation of queries

Nested intermediate data

Compiling XPath to SQL

Closed quotation vs. open quotation

$\mathsf{Expr}< A \to B >$ *vs.* $\mathsf{Expr}< A > \to \mathsf{Expr}< B >$

T-LINQ: the theory

Scylla and Charybdis Theorem

P-LINQ: the practice

Measured times comparable

Normalisation a small fraction of time

Good DSLs copy, great DSLs steal

Nikola (Mainland and Morrisett 2010)

Feldspar (Axelsson et al. 2010; Axelsson and Svenningsson 2012)

Host	DSEL
$a + b$	$a + b$
$a < b$	a .<. b
if a **then** b **else** c	a ? (b, c)

DSEL's steal the host's *type system*.

We steal the host's *type system* and *syntax*, and we provide *normalisation*.

Theory and Practice

T-LINQ:

doesn't cover sorting, grouping, aggregation

(work for tomorrow)

P-LINQ:

covers all of LINQ

(put it to work today!)

http://fsprojects.github.io/FSharp.Linq.Experimental.ComposableQuery/

What is the difference between
theory and practice?

In theory there
is no difference.
But in practice
there is.

What is the difference between theory and practice?

In theory there is a difference. But in practice there isn't.