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)
Links

Wadler, Yallop, Lindley, Cooper
(Edinburgh)

LINQ

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

Wadler, Yallop, Lindley, Cooper (Edinburgh)

LINQ

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

\(\lambda\) \hspace{2cm} \lambda
Scylla and Charybdis
Avoid Scylla and Charybdis

Each host query generates one SQL query

*Scylla*: failure to generate a query (×)

*Charybdis*: multiple queries, avalanche (av)
<table>
<thead>
<tr>
<th>Example</th>
<th>F# 2.0</th>
<th>F# 3.0</th>
<th>us</th>
</tr>
</thead>
<tbody>
<tr>
<td>differences</td>
<td>17.6</td>
<td>20.6</td>
<td>18.1</td>
</tr>
<tr>
<td>range</td>
<td>×</td>
<td>5.6</td>
<td>2.9</td>
</tr>
<tr>
<td>satisfies</td>
<td>2.6</td>
<td>×</td>
<td>2.9</td>
</tr>
<tr>
<td>satisfies</td>
<td>4.4</td>
<td>×</td>
<td>4.6</td>
</tr>
<tr>
<td>compose</td>
<td>×</td>
<td>×</td>
<td>4.0</td>
</tr>
<tr>
<td>$P(t_0)$</td>
<td>2.8</td>
<td>×</td>
<td>3.3</td>
</tr>
<tr>
<td>$P(t_1)$</td>
<td>2.7</td>
<td>×</td>
<td>3.0</td>
</tr>
<tr>
<td>expertise'</td>
<td>7.2</td>
<td>9.2</td>
<td>8.0</td>
</tr>
<tr>
<td>expertise</td>
<td>×</td>
<td>66.7\textsuperscript{av}</td>
<td>8.3</td>
</tr>
<tr>
<td>$\text{xp}_0$</td>
<td>×</td>
<td>8.3</td>
<td>7.9</td>
</tr>
<tr>
<td>$\text{xp}_1$</td>
<td>×</td>
<td>14.7</td>
<td>13.4</td>
</tr>
<tr>
<td>$\text{xp}_2$</td>
<td>×</td>
<td>17.9</td>
<td>20.7</td>
</tr>
<tr>
<td>$\text{xp}_3$</td>
<td>×</td>
<td>3744.9</td>
<td>3768.6</td>
</tr>
</tbody>
</table>

\textsuperscript{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
Expr< A → B > vs. Expr< A > → 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

<table>
<thead>
<tr>
<th>people</th>
<th>couples</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>her</td>
</tr>
<tr>
<td>age</td>
<td>him</td>
</tr>
<tr>
<td>“Alex”</td>
<td>“Alex”</td>
</tr>
<tr>
<td>60</td>
<td>“Bert”</td>
</tr>
<tr>
<td>“Bert”</td>
<td>“Cora”</td>
</tr>
<tr>
<td>56</td>
<td>“Drew”</td>
</tr>
<tr>
<td>“Cora”</td>
<td>“Edna”</td>
</tr>
<tr>
<td>33</td>
<td>“Fred”</td>
</tr>
<tr>
<td>“Drew”</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>“Edna”</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>“Fred”</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></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>&quot;Alex&quot;</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Cora&quot;</td>
<td>2</td>
</tr>
</tbody>
</table>
A database as data

{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)

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)

```javascript
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)

```haskell
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

\[
\text{let range : Expr< (int, int) → Names > =}
\]
\[
<@ \text{fun}(a, b) \rightarrow \text{for w in (%db).people do}
\]
\[
\quad \text{if } a \leq w.\text{age} \land w.\text{age} < b \text{ then yield } \{\text{name : w.name}\} @>
\]

\[
\text{run}(<@ (%range)(30, 40) @>)
\]

\[
\text{select w.name as name}
\]
\[
\text{from people as w}
\]
\[
\text{where } 30 \leq w.\text{age} \land w.\text{age} < 40
\]
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

```haskell

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

let t₀ : Predicate = And(Above(30), 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

\[ P(t_0) \]
\[ \leadsto \langle @ \hspace{1em} \text{fun}(x) \rightarrow (\text{fun}(x) \rightarrow 30 \leq x)(x) \hspace{1em} \&\& \hspace{1em} (\text{fun}(x) \rightarrow x < 40)(x) \rangle @> \]
\[ \leadsto \langle @ \hspace{1em} \text{fun}(x) \rightarrow 30 \leq x \hspace{1em} \&\& \hspace{1em} x < 40 \rangle @> \]

\[ \text{run}(\langle @ \hspace{1em} (\%\text{satisfies})(\%P(t_0)) \rangle @>) \]

\textbf{select \hspace{0.5em} w.name as \hspace{0.5em} name}
\textbf{from \hspace{0.5em} people \hspace{0.5em} as \hspace{0.5em} w}
\textbf{where \hspace{0.5em} 30 \leq w.age \hspace{1em} and \hspace{0.5em} w.age < 40}
Part III

Nested intermediate data
Flat data

```javascript
{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

\[
\text{let } \text{expertise}' : \text{Expr< string } \rightarrow \{ \text{dpt : string}\} \text{ list } = \\
<@ \text{fun}(u) \rightarrow \text{for } d \text{ in } (%\text{org}).\text{departments} \text{ do}
\text{if not(exists(}
\text{for } e \text{ in } (%\text{org}).\text{employees} \text{ do}
\text{if } d.\text{dpt} = e.\text{dpt} \text{ && not(exists(}
\text{for } t \text{ in } (%\text{org}).\text{tasks} \text{ do}
\text{if } e.\text{emp} = t.\text{emp} \text{ && t.tsk} = u \text{ then yield } \{\} )
)) \text{ then yield } \{\} )
)) \text{ then yield } \{ \text{dpt} = d.\text{dpt} \} @>
\]

\[
\text{run}(<@ (\text{%expertise}')(\text{"abstract"}) @>)
[ \{\text{dpt} = \text{"Quality"}\}; \{\text{dpt} = \text{"Research"}\} ]
\]
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

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

\[
\text{let expertise : Expr< string → \{dpt : string\} list > =}
\]

\[
<@ \text{fun}(u) \rightarrow \text{for } d \text{ in } (\%\text{nestedOrg})
\]

\[
\text{if } (\%\text{all})(d.employees,}
\]

\[
\text{fun}(e) \rightarrow (\%\text{contains})(e.tasks, u) \text{ then}
\]

\[
\text{yield } \{dpt = d.dpt\} @>
\]

\[
\text{run}(<@ (\%\text{expertise)}(“abstract”) @>)
\]

\[
[\{dpt = “Quality”\}; \{dpt = “Research”\}]\]
Part IV

Compiling XPath to SQL
Part V

Closed quotation vs. open quotation
Dynamically generated queries, revisited

\[
\textbf{let rec } P(t : \text{Predicate}) : \text{Expr< int } \rightarrow \text{ bool } > = \\
\text{match } t \text{ with } \\
| \text{Above}(a) \rightarrow \text{fun}(x) \rightarrow (\%\text{lift}(a)) \leq x @> \\
| \text{Below}(a) \rightarrow \text{fun}(x) \rightarrow x < (\%\text{lift}(a)) @> \\
| \text{And}(t, u) \rightarrow \text{fun}(x) \rightarrow (\%P(t))(x) \&\& (\%P(u))(x) @>
\]

\textbf{VS.}

\[
\textbf{let rec } P(t : \text{Predicate})(x : \text{Expr< int } ) : \text{Expr< bool } > = \\
\text{match } t \text{ with } \\
| \text{Above}(a) \rightarrow \text{fun}(x) \rightarrow (\%\text{lift}(a)) \leq (\%x) @> \\
| \text{Below}(a) \rightarrow \text{fun}(x) \rightarrow (\%x) < (\%\text{lift}(a)) @> \\
| \text{And}(t, u) \rightarrow \text{fun}(x) \rightarrow (\%P(t)(x)) \&\& (\%P(u)(x)) @>
\]
Abstracting over a predicate, revisited

```latex
\begin{align*}
\textbf{let} \text{ satisfies} & : \text{Expr}<\text{int} \to \text{bool}> \to \text{Names} > = \\
<@ \textbf{fun}(p) \to \textbf{for} w \textbf{ in } (@db).\text{people } \textbf{do} \\
& \textbf{ if } p(w.\text{age}) \textbf{ then} \\
& \textbf{yield} \{\text{name} : w.\text{name}\} @> \\
\end{align*}
```

VS.

```latex
\begin{align*}
\textbf{let} \text{ satisfies}(p : \text{Expr}<\text{int}> \to \text{Expr}<\text{bool}> ) : \text{Expr}<\text{Names}> = \\
<@ \textbf{for} w \textbf{ in } (@db).\text{people } \textbf{do} \\
& \textbf{ if } (p(\texttt{<@ w.\text{age} @>})) \textbf{ then} \\
& \textbf{yield} \{\text{name} : w.\text{name}\} @>
\end{align*}
```
closed quotations

vs.

one open quotations

quotations of functions

(Expr< A → B >)

vs.

functions of quotations

(Expr< A > → Expr< B >)
Part VI

T-LINQ: the theory
Host language

\[
\begin{align*}
\text{FUN} & \quad \Gamma, x : A \vdash N : B \\
& \quad \Gamma \vdash \text{fun}(x) \rightarrow N : A \rightarrow B \\
\text{APP} & \quad \Gamma \vdash L : A \rightarrow B \\
& \quad \Gamma \vdash M : A \\
& \quad \Gamma \vdash LM : B \\
\text{SINGLETON} & \quad \Gamma \vdash M : A \\
& \quad \Gamma \vdash \text{yield} M : A \text{ list} \\
\text{FOR} & \quad \Gamma \vdash M : A \text{ list} \\
& \quad \Gamma, x : A \vdash N : B \text{ list} \\
& \quad \Gamma \vdash \text{for x in } M \text{ do } N : B \text{ list} \\
\text{QUOTE} & \quad \Gamma ; \cdot \vdash M : A \\
& \quad \Gamma \vdash \langle @ M @ \rangle : \text{Expr}< A > \\
\text{RUN} & \quad \Gamma \vdash M : \text{Expr}< T > \\
& \quad \Gamma \vdash \text{run}(M) : T \\
\text{REC} & \quad \Gamma, f : A \rightarrow B, x : A \vdash N : B \\
& \quad \Gamma \vdash \text{rec } f(x) \rightarrow N : A \rightarrow B
\end{align*}
\]
Quoted language

**FunQ**
\[\Gamma; \Delta, x : A \vdash N : B \quad \Gamma; \Delta \vdash \text{fun}(x) \rightarrow N : A \rightarrow B\]

**AppQ**
\[\Gamma; \Delta \vdash L : A \rightarrow B \quad \Gamma; \Delta \vdash M : A \quad \Gamma; \Delta \vdash L \ M : B\]

**SingletonQ**
\[\Gamma; \Delta \vdash M : A \quad \Gamma; \Delta \vdash \text{yield} \ M : A \ \text{list}\]

**ForQ**
\[\Gamma; \Delta \vdash M : A \ \text{list} \quad \Gamma; \Delta, x : A \vdash N : B \ \text{list} \quad \Gamma; \Delta \vdash \text{for} \ x \ \text{in} \ M \ \text{do} \ N : B \ \text{list}\]

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

**Lift**
\[\Gamma \vdash M : O \quad \Gamma \vdash \text{lift}(M) : \text{Expr}<O>\]

**Database**
\[\Sigma(db) = \{\ell : T\} \quad \Gamma; \Delta \vdash \text{database}(db) : \{\ell : T\}\]
Normalisation: symbolic evaluation

\[(\text{fun}(x) \rightarrow N) \ M \leadsto N[x := M]\]

\[\{\ell = M\}.\ell_i \leadsto M_i\]

\[\text{for } x \text{ in } (\text{yield } M) \text{ do } N \leadsto N[x := M]\]

\[\text{for } y \text{ in } (\text{for } x \text{ in } L \text{ do } M) \text{ do } N \leadsto \text{for } x \text{ in } L \text{ do } (\text{for } y \text{ in } M \text{ do } N)\]

\[\text{for } x \text{ in } (\text{if } L \text{ then } M) \text{ do } N \leadsto \text{if } L \text{ then } (\text{for } x \text{ in } M \text{ do } N)\]

\[\text{for } x \text{ in } [ ] \text{ do } N \leadsto [ ]\]

\[\text{for } x \text{ in } (L @ M) \text{ do } N \leadsto (\text{for } x \text{ in } L \text{ do } N) @ (\text{for } x \text{ in } M \text{ do } N)\]

\[\text{if true then } M \leadsto M\]

\[\text{if false then } M \leadsto [ ]\]
Normalisation: \textit{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 \text{ && } M) \text{ then } N
\end{align*}
Theorem \textit{(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
<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>
<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>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>

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

  - 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.


- Propositions as Sessions, ICFP 2012, JFP 2014.

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
\[ \text{Expr} \langle A \rightarrow B \rangle \text{ vs. } \text{Expr} \langle A \rangle \rightarrow \text{Expr} \langle B \rangle \]

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)

<table>
<thead>
<tr>
<th>Host</th>
<th>DSEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>a + b</td>
<td>a + b</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>a .&lt;. b</td>
</tr>
<tr>
<td>if a then b else c</td>
<td>a ? (b, c)</td>
</tr>
</tbody>
</table>

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!)

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.