

A Practical Theory of Language-Integrated Query

James Cheney, Sam Lindley, Philip Wadler
University of Edinburgh

FP Days

Cambridge, Thursday 24 October 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?

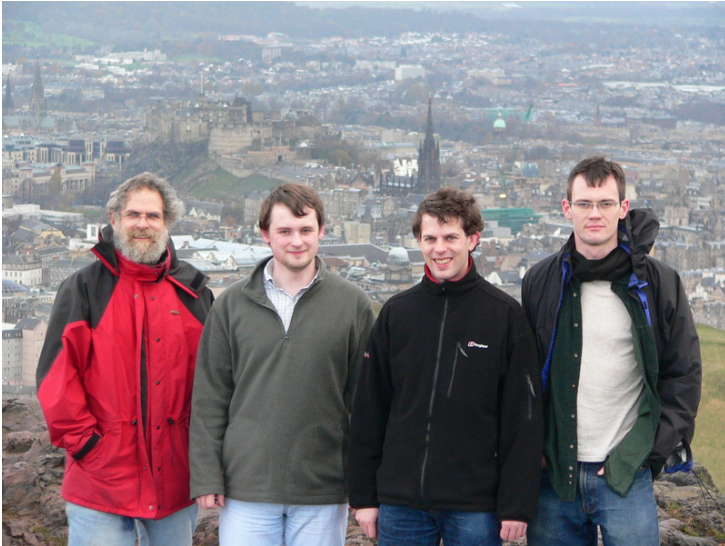
How does one integrate SQL
and a host language?

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

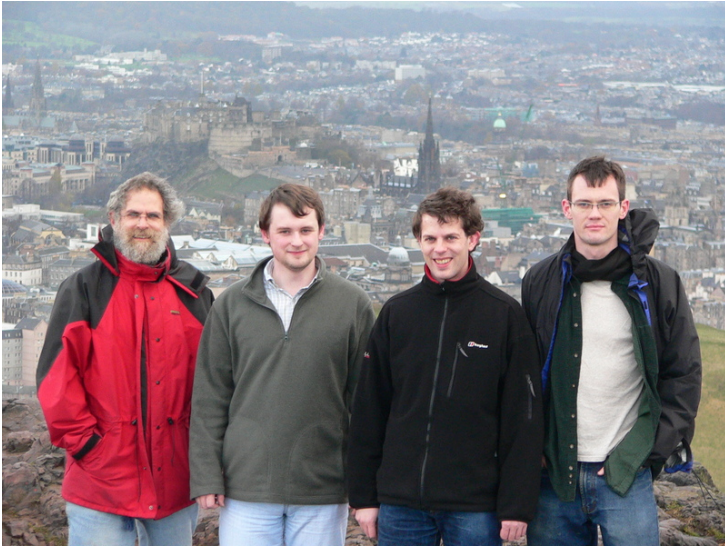
LINQ



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 (×)

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

$\text{Expr} \langle A \rightarrow B \rangle$ 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

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

{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)

```
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)

```
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 \rightarrow **bool**) \rightarrow Names > =

<@ **fun**(p) \rightarrow **for** w **in** (%db).people **do**

if p(w.age) **then**

yield {name : w.name} @>

run(<@ (%satisfies)(**fun**(x) \rightarrow $30 \leq x \ \&\& \ x < 40$) @>)

select w.name **as** name

from people **as** w

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

Datatype of predicates

type Predicate =

- | Above **of** int
- | Below **of** int
- | And **of** Predicate \times Predicate
- | Or **of** Predicate \times Predicate
- | Not **of** Predicate

let t_0 : 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

$P(t_0)$

$\rightsquigarrow \langle @ \text{ fun}(x) \rightarrow (\text{fun}(x) \rightarrow 30 \leq x)(x) \ \&\& \ (\text{fun}(x) \rightarrow x < 40)(x) \ @ \rangle$

$\rightsquigarrow \langle @ \text{ fun}(x) \rightarrow 30 \leq x \ \&\& \ x < 40 \ @ \rangle$

`run(<@ (%satisfies)(%P(t0)) @>)`

`select w.name as name`

`from people as w`

`where 30 ≤ w.age and w.age < 40`

Part III

Nested intermediate data

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

```
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

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

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

$(\text{Expr} \langle A \rightarrow B \rangle)$

vs.

functions of quotations

$(\text{Expr} \langle A \rangle \rightarrow \text{Expr} \langle B \rangle)$

Part VI

T-LINQ: the theory

Host language

FUN

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

APP

$$\frac{\Gamma \vdash L : A \rightarrow B \quad \Gamma \vdash M : A}{\Gamma \vdash L M : B}$$

SINGLETON

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

FOR

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

QUOTE

$$\frac{\Gamma; \cdot \vdash M : A}{\Gamma \vdash \langle @ M @ \rangle : \mathbf{Expr} \langle A \rangle}$$

RUN

$$\frac{\Gamma \vdash M : \mathbf{Expr} \langle T \rangle}{\Gamma \vdash \mathbf{run}(M) : T}$$

REC

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

Quoted language

FUNQ

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

APPQ

$$\frac{\Gamma; \Delta \vdash L : A \rightarrow B \quad \Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash L M : B}$$

SINGLETONQ

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

FORQ

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

ANTIQUOTE

$$\frac{\Gamma \vdash M : \mathbf{Expr} \langle A \rangle}{\Gamma; \Delta \vdash (\%M) : A}$$

LIFT

$$\frac{\Gamma \vdash M : O}{\Gamma \vdash \mathbf{lift}(M) : \mathbf{Expr} \langle O \rangle}$$

DATABASE

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

Normalisation: symbolic evaluation

$(\mathbf{fun}(x) \rightarrow N) M \rightsquigarrow N[x := M]$

$\{\overline{\ell = M}\}.\ell_i \rightsquigarrow M_i$

$\mathbf{for } x \mathbf{ in (yield } M) \mathbf{ do } N \rightsquigarrow N[x := M]$

$\mathbf{for } y \mathbf{ in (for } x \mathbf{ in } L \mathbf{ do } M) \mathbf{ do } N \rightsquigarrow \mathbf{for } x \mathbf{ in } L \mathbf{ do (for } y \mathbf{ in } M \mathbf{ do } N)$

$\mathbf{for } x \mathbf{ in (if } L \mathbf{ then } M) \mathbf{ do } N \rightsquigarrow \mathbf{if } L \mathbf{ then (for } x \mathbf{ in } M \mathbf{ do } N)$

$\mathbf{for } x \mathbf{ in } [] \mathbf{ do } N \rightsquigarrow []$

$\mathbf{for } x \mathbf{ in } (L @ M) \mathbf{ do } N \rightsquigarrow (\mathbf{for } x \mathbf{ in } L \mathbf{ do } N) @ (\mathbf{for } x \mathbf{ in } M \mathbf{ do } N)$

$\mathbf{if true then } M \rightsquigarrow M$

$\mathbf{if false then } M \rightsquigarrow []$

Normalisation: *ad hoc* rewriting

for x **in** L **do** $(M @ N) \hookrightarrow (\text{for } x \text{ in } L \text{ do } M) @ (\text{for } x \text{ in } L \text{ do } N)$

for x **in** L **do** $[] \hookrightarrow []$

if L **then** $(M @ N) \hookrightarrow (\text{if } L \text{ then } M) @ (\text{if } L \text{ then } N)$

if L **then** $[] \hookrightarrow []$

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

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

Theorem (*Scylla and Charybdis*) If

$$\vdash L : A$$

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

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

where M and N are in normal form with respect to \rightsquigarrow 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

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

$\text{Expr} \langle A \rightarrow B \rangle$ 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)

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!)

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.