# The Essence of Language Integrated Query

James Cheney, Sam Lindley, Philip Wadler
University of Edinburgh

Midlands Graduate School

Leicester, 9 April 2013

# Database programming languages

Kleisli

Buneman, Libkin, Suciu, Tannen, Wong (Penn)

Ferry

Grust, Mayr, Rittinger, Schreiber (Tübingen)

Links

Cooper, Lindley, Wadler, Yallop (Edinburgh)

SML#

Ohori, Ueno (Tohoku)

Ur/Web

Chlipala (Harvard/MIT)

LINQ for C#, VB, F#

Helsbjorg, Meijer, Syme (Microsoft Redmond & Cambridge)

*Our goals:*

Abstraction over values (first-order)

Abstraction over predicates (higher-order)

Composition of queries

Dynamic generation of queries

Type-safety

*Goldilocks:*

Exactly one query per *run*

Not too few (failure)

Not too many (avalanche)

*Our restrictions:*

We consider only *select-from-where* queries,
with *exists* and *union*.

We equate *bags* and *lists*.

Future work to extend to *group-by* and *sort-by*.

# 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

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

# 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)

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

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} ]

Execute **run** as follows:

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

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. consistent use of **database** (all same)

# 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)

$$
\begin{aligned}
&\textbf{type}\ \mathsf{Predicate} = \\
&\quad |\ \mathsf{Above}\ \textbf{of int} \\
&\quad |\ \mathsf{Below}\ \textbf{of int} \\
&\quad |\ \mathsf{And}\ \textbf{of}\ \mathsf{Predicate} \times \mathsf{Predicate} \\
&\quad |\ \mathsf{Or}\ \textbf{of}\ \mathsf{Predicate} \times \mathsf{Predicate} \\
&\quad |\ \mathsf{Not}\ \textbf{of}\ \mathsf{Predicate} \\
&\textbf{let}\ \mathsf{t}_0 : \mathsf{Predicate} = \mathsf{And}(\mathsf{Above}(30), \mathsf{Below}(40)) \\
&\textbf{let}\ \mathsf{t}_1 : \mathsf{Predicate} = \mathsf{Not}(\mathsf{Or}(\mathsf{Below}(30), \mathsf{Above}(40)))
\end{aligned}
$$

# 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)

$$P(t_0)$$

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

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

$$\mathbf{run}(\texttt{<@}\ (\%\text{satisfies})(\%P(t_0))\ \texttt{@>})$$

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

$$\mathbf{run}(\texttt{<@}\ (\%\text{satisfies})(\%P(t_1))\ \texttt{@>})$$

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

# 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)

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

# 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

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

# 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

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

AppQ

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

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

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

Antiquote

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

Run

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

Lift

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

# Normalisation: symbolic evaluation

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

# Normalisation: *ad hoc* rewriting

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

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

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

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

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

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

$$\textbf{yield } x \hookrightarrow \textbf{yield } \{\overline{\ell = x.\ell}\}$$

$$\textbf{database}(\mathrm{db}).\ell \hookrightarrow \textbf{for } x \textbf{ in } \textbf{database}(\mathrm{db}).\ell \textbf{ do yield } x$$

# 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.

# 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 | ILINQ | *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 |

All times in milliseconds. $^{av}$ marks query avalanche.

*Our goals:*

Abstraction over values (first-order)

Abstraction over predicates (higher-order)

Composition of queries

Dynamic generation of queries

Type-safety

*Goldilocks:*

Exactly one query per *run*

Not too few (failure)

Not too many (avalanche)


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

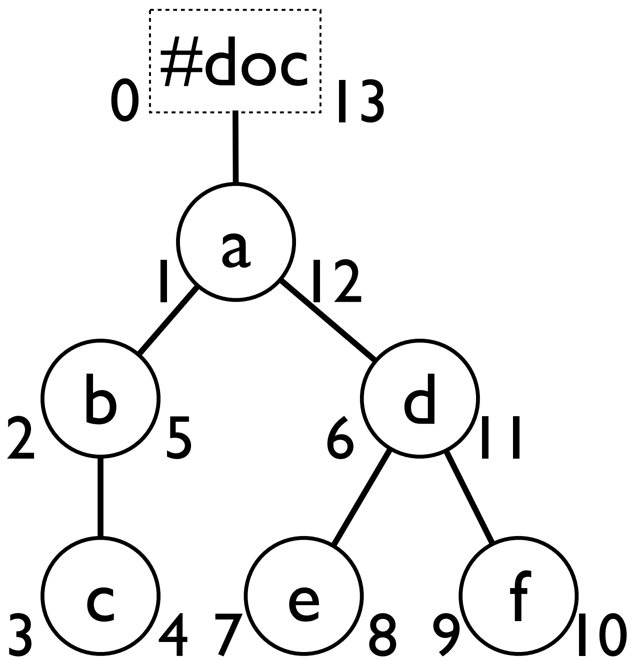
```
query if 1 = 2 then yield 3
```

leads to a run-time type error.

Appendix A7

# From XPath to SQL

# Representing XML




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) @>
```

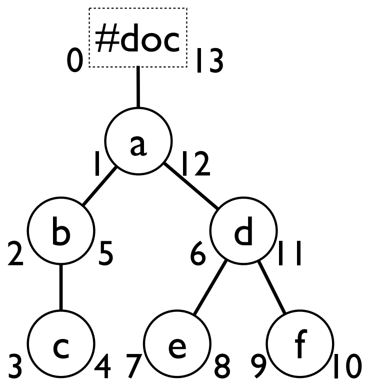
# An evaluator for XPath: path

```
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"))))))

$[2]$