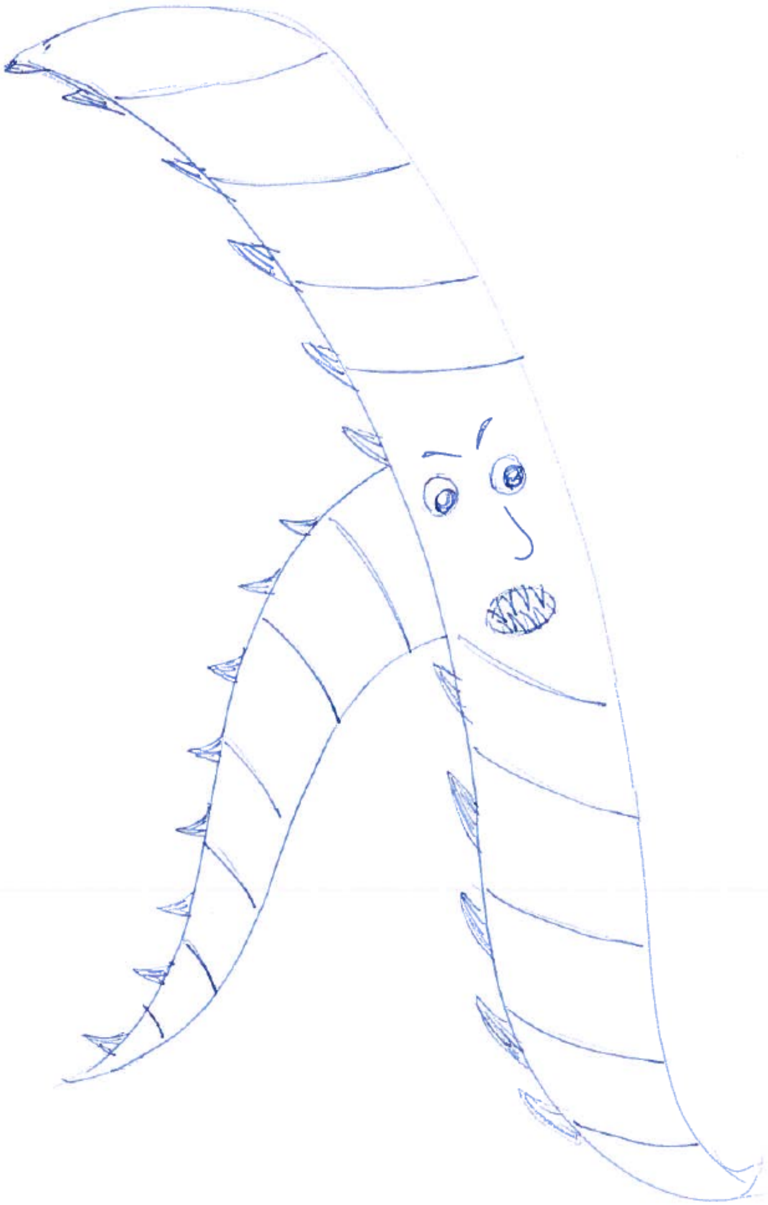
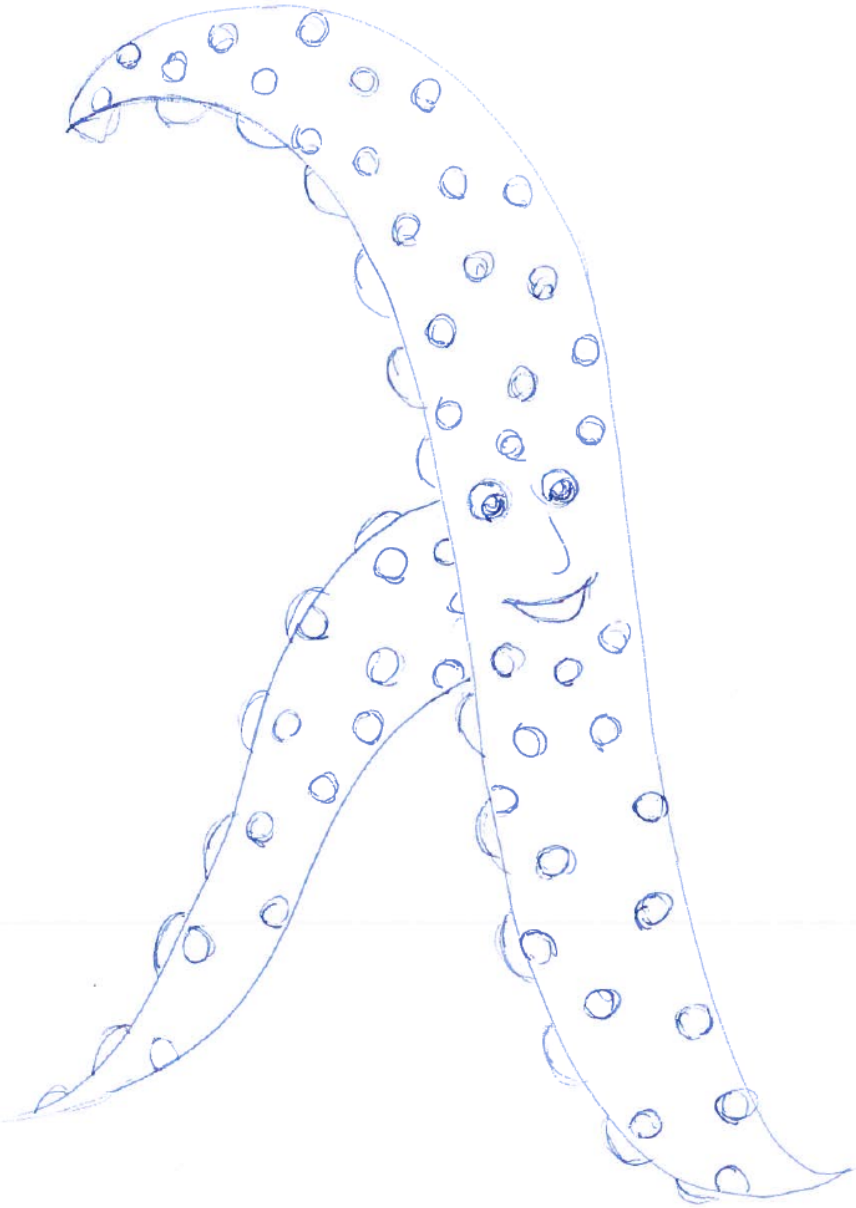
The Essence of
Language Integrated Query

James Cheney, Sam Lindley, Philip Wadler
University of Edinburgh

DDFP, Rome, 9am Tuesday 22 January 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

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

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)

```
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}]
```

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 ageFromName : Expr< string → int list > =
  <@ fun(s) → for u in (%db).people do
              if u.name = s then
              yield u.age @>
let rangeFromNames : Expr< (string, string) → Names > =
  <@ fun(s, t) → for a in (%ageFromName)(s) do
                 for b in (%ageFromName)(t) do
                 (%range)(a, b) @>
```

```
run(<@ (%nameRange)(“Edna”, “Bert”) @>)
[ {name = “Cora”}; {name = “Drew”}; {name = “Edna”} ]
```

Dynamically generated queries (1)

type Predicate =
  | Above **of int**
  | Below **of int**
  | And **of** Predicate × Predicate
  | Or **of** Predicate × Predicate
  | 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)

$\mathsf{P}(\mathsf{t}_0)$

`<@` **fun**(x) → (**fun**(x) → 30 ≤ x)(x) && (**fun**(x) → x < 40)(x) `@>`

`<@` **fun**(x) → 30 ≤ x && x < 40 `@>`

run(`<@` (%satisfies)(%P(t_0)) `@>`)

[{name = "Cora"}; {name = "Drew"}]

run(`<@` (%satisfies)(%P(t_1)) `@>`)

[{name = "Cora"}; {name = "Drew"}]

Part III

Nesting

Flat data

$$
\begin{array}{l}
\{\text{departments} = \\
\qquad [\,\{\text{dpt} = \text{“Product”}\}; \\
\qquad\;\; \{\text{dpt} = \text{“Quality”}\}; \\
\qquad\;\; \{\text{dpt} = \text{“Research”}\}; \\
\qquad\;\; \{\text{dpt} = \text{“Sales”}\}\,]\,; \\
\;\text{employees} = \\
\qquad [\,\{\text{dpt} = \text{“Product”}; \text{emp} = \text{“Alex”}\}; \\
\qquad\;\; \{\text{dpt} = \text{“Product”}; \text{emp} = \text{“Bert”}\}; \\
\qquad\;\; \{\text{dpt} = \text{“Research”}; \text{emp} = \text{“Cora”}\}; \\
\qquad\;\; \{\text{dpt} = \text{“Research”}; \text{emp} = \text{“Drew”}\}; \\
\qquad\;\; \{\text{dpt} = \text{“Research”}; \text{emp} = \text{“Edna”}\}; \\
\qquad\;\; \{\text{dpt} = \text{“Sales”}; \text{emp} = \text{“Fred”}\}\,]\,;
\end{array}
$$

Flat data (continued)

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

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

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

vs.

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

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

APPQ

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

SINGLETONQ

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

FORQ

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

DATABASE

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

Quotation and anti-quotation

Quote

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

Antiquote

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

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

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

Normalisation: *ad hoc* rewriting

$$\begin{aligned}
\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ (M \mathbin{\texttt{@}} N) &\hookrightarrow (\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ M) \mathbin{\texttt{@}} (\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ N) \\
\textbf{for}\ x\ \textbf{in}\ L\ \textbf{do}\ \texttt{[ ]} &\hookrightarrow \texttt{[ ]} \\
\textbf{if}\ L\ \textbf{then}\ (M \mathbin{\texttt{@}} N) &\hookrightarrow (\textbf{if}\ L\ \textbf{then}\ M) \mathbin{\texttt{@}} (\textbf{if}\ L\ \textbf{then}\ N) \\
\textbf{if}\ L\ \textbf{then}\ \texttt{[ ]} &\hookrightarrow \texttt{[ ]} \\
\textbf{if}\ L\ \textbf{then}\ (\textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ N) &\hookrightarrow \textbf{for}\ x\ \textbf{in}\ M\ \textbf{do}\ (\textbf{if}\ L\ \textbf{then}\ N) \\
\textbf{if}\ L\ \textbf{then}\ (\textbf{if}\ M\ \textbf{then}\ N) &\hookrightarrow \textbf{if}\ (L\ \ \&\&\ \ M)\ \textbf{then}\ N \\
\textbf{yield}\ x &\hookrightarrow \textbf{yield}\ \{\overline{\ell = x.\ell}\} \\
\textbf{database}(\mathsf{db}).\ell &\hookrightarrow \textbf{for}\ x\ \textbf{in}\ \textbf{database}(\mathsf{db}).\ell\ \textbf{do}\ \textbf{yield}\ x
\end{aligned}$$

Properties of reduction

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

- *preserve* typing,
- are *stongly normalising*, and
- are *confluent*.

Example (1): query

```
run(<@ (%nameRange)("Edna", "Bert") @>)
```

Example (2): after splicing

```
(fun(s, t) $\to$
   for a in (fun(s) $\to$
              for u in database(“People”).people do
              if u.name $=$ s then yield u.age)(s) do
   for b in (fun(s) $\to$
              for u in database(“People”).people do
              if u.name $=$ s then yield u.age)(t) do
   (fun(a, b) $\to$
      for w in database(“People”).people do
      if a $\leq$ w.age  &&  w.age $<$ b then
      yield {name : w.name})(a, b))
(“Edna”, “Bert”)
```

Example (3): beta reduction $\leadsto$

```
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 ≤ 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 ≤ w.age and w.age < v.age
```

Part VII

Results

Example	F# 2.0	F# 3.0	Our system
differences	✔	✔	✔
range	✖	✔	✔
satisfies	✔	✖	✔
satisfies	✔	✖	✔
rangeFromNames	✖	✖	✔
P(t_0)	✔	✖	✔
P(t_1)	✔	✖	✔
expertise$'$	✔	✔	✔
expertise	✖	avalanche	✔
xp$_0$	✖	✔	✔
xp$_1$	✖	✔	✔
xp$_2$	✖	✖	✔
xp$_3$	✖	✖	✔

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)

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

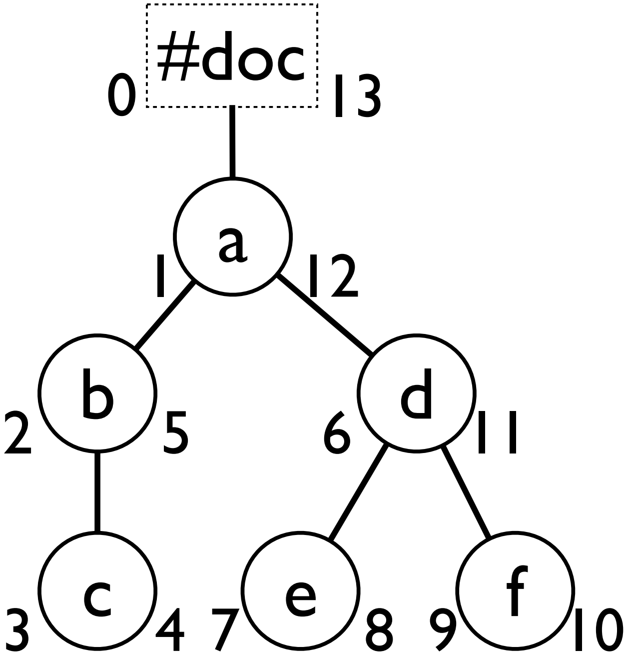
  ```
  query if 1 = 2 then yield 3
  ```

  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 =

$\{\mathsf{id} : \textbf{int}, \mathsf{parent} : \textbf{int}, \mathsf{name} : \textbf{string}, \mathsf{pre} : \textbf{int}, \mathsf{post} : \textbf{int}\}$

Abstract syntax of XPath

type Axis =
| Self
| Child
| Descendant
| DescendantOrSelf
| Following
| FollowingSibling
| Rev **of** Axis

type Path =
| Seq **of** Path × 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) @>
```

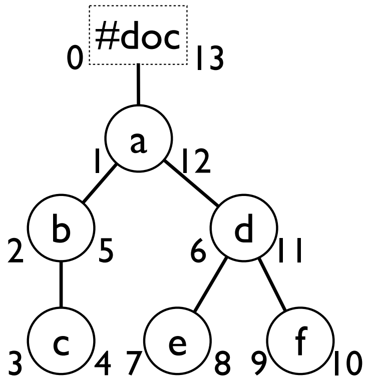
An evaluator for XPath: path

```
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”))))))

$[2]$