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 \textit{select-from-where} queries, with \textit{exists} and \textit{union}.

We equate \textit{bags} and \textit{lists}.

Future work to extend to \textit{group-by} and \textit{sort-by}.
Part I

A first example
A database

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Alex”</td>
<td>60</td>
</tr>
<tr>
<td>“Bert”</td>
<td>56</td>
</tr>
<tr>
<td>“Cora”</td>
<td>33</td>
</tr>
<tr>
<td>“Drew”</td>
<td>31</td>
</tr>
<tr>
<td>“Edna”</td>
<td>21</td>
</tr>
<tr>
<td>“Fred”</td>
<td>60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>her</th>
<th>him</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Alex”</td>
<td>“Bert”</td>
</tr>
<tr>
<td>“Cora”</td>
<td>“Drew”</td>
</tr>
<tr>
<td>“Edna”</td>
<td>“Fred”</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>“Alex”</td>
<td>4</td>
</tr>
<tr>
<td>“Cora”</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)

```plaintext
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)

```plaintext
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)

```plaintext
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

```haskell
  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

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

```hs
run(<@ (%satisfies)(fun(x) → 30 ≤ x && x < 40) @>)
[ {name = “Cora”}; {name = “Drew”} ]
```

```hs
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)

```plaintext
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))
let t₁ : Predicate = Not(Or(Below(30), Above(40)))
```
Dynamically generated queries (2)

\[
\text{let rec } P(t : \text{Predicate}) : \text{Expr< int } \rightarrow \text{ bool }> =
\]

match \( t \) with

| Above(a) \rightarrow @{ \text{fun}(x) \rightarrow (\%\text{lift}(a)) \leq x } @> |
| Below(a) \rightarrow @{ \text{fun}(x) \rightarrow x < (\%\text{lift}(a)) } @> |
| And(t, u) \rightarrow @{ \text{fun}(x) \rightarrow (\%P(t))(x) \&\& (\%P(u))(x) } @> |
| Or(t, u) \rightarrow @{ \text{fun}(x) \rightarrow (\%P(t))(x) \lor \lor (\%P(u))(x) } @> |
| Not(t) \rightarrow @{ \text{fun}(x) \rightarrow \text{not}(\%P(t))(x) } @>
\]
Dynamically generated queries (3)

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

\[ \text{run}(\langle @ (\%satisfies)(\%P(t_0)) \rangle >) \]
\[ \{ \text{name} = \text{“Cora”}; \ {\text{name} = \text{“Drew”}} \} \]

\[ \text{run}(\langle @ (\%satisfies)(\%P(t_1)) \rangle >) \]
\[ \{ \text{name} = \text{“Cora”}; \ {\text{name} = \text{“Drew”}} \} \]
Part III

Nesting
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

```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

```haskell
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

```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

```haskell
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

```haskell
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

```fsharp
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.

```fsharp
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

```plaintext
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**

\[
\Gamma, x : A \vdash x : A
\]

**FUN**

\[
\Gamma, x : A \vdash N : B \\
\Gamma \vdash \text{fun}(x) \to N : A \to B
\]

**APP**

\[
\Gamma \vdash L : A \to B \quad \Gamma \vdash M : A \\
\Gamma \vdash L \, M : B
\]

**SINGLETON**

\[
\Gamma \vdash M : A \\
\Gamma \vdash \text{yield} \, M : A \, \text{list}
\]

**FOR**

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

**Rec**

\[
\Gamma, f : A \to B, x : A \vdash N : B \\
\Gamma \vdash \text{rec} \, f(x) \to N : A \to B
\]
Quoted terms

\[ \text{VARQ} \]
\[ \frac{\Gamma; \Delta, x : A \vdash x : A}{\Gamma; \Delta, x : A \vdash x : A} \]

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

\[ \text{APPQ} \]
\[ \frac{\Gamma; \Delta \vdash L : A \rightarrow B, \Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash L \, M : B} \]

\[ \text{SINGLETONQ} \]
\[ \frac{\Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash \text{yield} \, M : A \, \text{list}} \]

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

\[ \text{DATABASE} \]
\[ \Sigma(db) = \{ \ell : T \} \]
\[ \frac{}{\Gamma; \Delta \vdash \text{database}(db) : \{ \ell : T \}} \]
Quotation and anti-quotation

**QUOTE**

$$\Gamma; \cdot \vdash M : A$$

$$\Gamma \vdash <@ M @> : \text{Expr}< A >$$

**ANTIQUOTE**

$$\Gamma \vdash M : \text{Expr}< A >$$

$$\Gamma; \Delta \vdash (% M) : A$$

**RUN**

$$\Gamma \vdash M : \text{Expr}< T >$$

$$\Gamma \vdash \text{run}(M) : T$$

**LIFT**

$$\Gamma \vdash M : O$$

$$\Gamma \vdash \text{lift}(M) : \text{Expr}< O >$$
Normalisation: symbolic evaluation

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

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

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

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

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

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

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

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

\[\text{if false then } M \iff [\ ]\]
Normalisation: \textit{ad hoc} rewriting

\[
\begin{align*}
\text{for } x \text{ in } L \text{ do } (M @ N) & \leftrightarrow (\text{for } x \text{ in } L \text{ do } M) @ (\text{for } x \text{ in } L \text{ do } N) \\
\text{for } x \text{ in } L \text{ do } [ ] & \leftrightarrow [ ] \\
\text{if } L \text{ then } (M @ N) & \leftrightarrow (\text{if } L \text{ then } M) @ (\text{if } L \text{ then } N) \\
\text{if } L \text{ then } [ ] & \leftrightarrow [ ] \\
\text{if } L \text{ then } (\text{for } x \text{ in } M \text{ do } N) & \leftrightarrow \text{for } x \text{ in } M \text{ do } (\text{if } L \text{ then } N) \\
\text{if } L \text{ then } (\text{if } M \text{ then } N) & \leftrightarrow \text{if } (L \text{ && } M) \text{ then } N \\
\text{yield } x & \leftrightarrow \text{yield } \{\ell = x.\ell\} \\
\text{database(db).}\ell & \leftrightarrow \text{for } x \text{ in } \text{database(db).}\ell \text{ do } \text{yield } x
\end{align*}
\]
Properties of reduction

On well-typed terms, the relations $\rightsquigarrow$ and $\rightarrow$

- *preserve* typing,
- are *strongly 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

\[
(fun(s, t) \rightarrow \\
\quad \text{for } a \text{ in } (fun(s) \rightarrow \\
\quad \quad \text{for } u \text{ in database("People").people } \text{do} \\
\quad \quad \quad \text{if } u.\text{name} = s \text{ then yield } u.\text{age}) (s) \text{ do} \\
\quad \text{for } b \text{ in } (fun(s) \rightarrow \\
\quad \quad \text{for } u \text{ in database("People").people } \text{do} \\
\quad \quad \quad \text{if } u.\text{name} = s \text{ then yield } u.\text{age}) (t) \text{ do} \\
(fun(a, b) \rightarrow \\
\quad \text{for } w \text{ in database("People").people } \text{do} \\
\quad \quad \text{if } a \leq w.\text{age} \&\& w.\text{age} < b \text{ then} \\
\quad \quad \quad \text{yield } \{ \text{name : w.name} \} (a, b))
\]

(“Edna”, “Bert”)
Example (3): beta reduction

\[
\begin{align*}
\text{for } a \text{ in (for } u \text{ in database("People").people do} \\
& \quad \text{if } u.\text{name} = \text{"Edna" then yield } u.\text{age) do} \\
\text{for } b \text{ in (for } u \text{ in database("People").people do} \\
& \quad \text{if } u.\text{name} = \text{"Bert" then yield } u.\text{age) do} \\
\text{for } w \text{ in database("People").people do} \\
& \quad \text{if } a \leq w.\text{age} \quad \& \quad w.\text{age} < b \quad \text{then} \\
\text{yield } \{\text{name : } w.\text{name}\}
\end{align*}
\]
Example (4): other rewriting

```plaintext
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 ≤ w.age && w.age < v.age then
            yield {name : w.name}
```
Example (5): *ad hoc* reductions

```plaintext
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 ≤ w.age && w.age < v.age then
    yield {name : w.name}
```
Example (6): SQL

```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
<table>
<thead>
<tr>
<th>Example</th>
<th>F# 2.0</th>
<th>F# 3.0</th>
<th>ILINQ</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\text{av}</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>

All times in milliseconds. \text{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)
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 (e.g. 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

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 × Path
- Axis of Axis
- NameTest of string
- Filter of Path
An evaluator for XPath: axis

```plaintext
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

```plaintext
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

\[
\text{let } \text{xpath}(p : \text{Path}) : \text{Expr} \langle \text{Node list} \rangle = \\
<@ \text{for } root \text{ in } (\%db).\text{xml do} \text{for } s \text{ in } (\%db).\text{xml do} \text{if } root.\text{parent} = -1 \&\& (\%\text{path}(p))(root, s) \text{ then yield } s @> 
\]
Examples

```java
/* */
run(xpath(Seq(Axis(Child), Axis(Child))))
[2; 4]

// *[following-sibling::d]
run(xpath(Seq(Axis(Descendant),
    Filter(Seq(Axis(FollowingSibling),
        NameTest("d"))))))
[2]
```