A Common Core for Relational, Functional, and Imperative Calculi

Kristoffer Rose

IBM Thomas J. Watson Research Center, Yorktown Heights, New York

NeilFest, August 25-26, DIKU, København
Provenance

1 XQuery
- XML & XQuery & Virtualization
- Relational vs Functional
- Syntax & Semantics
- Types

2 Relational Algebra
- Syntax & Semantics
- Compilation
- Rewritings

3 Towards synthesis
- Join rewrite

4 Conclusions & Lessons
- Related work
Q
Is there a universal and practical way to view all data?

A
Extensible Markup Language (XML, W3C 1999)

1. XML is easy to learn (enough to use),
2. XML was designed to capture all aspects of information representation, and
3. XML was designed by information processing community*

* Except the relational database community.
Is there a universal and practical way to view all data?

Extensible Markup Language (XML, W3C 1999)

1. XML is easy to learn (enough to use),
2. XML was designed to capture all aspects of information representation, and
3. XML was designed by information processing community*

* Except the relational database community.
Q
Is there a universal and practical way to view all data?

A
Extensible Markup Language (XML, W3C 1999)

1. XML is easy to learn (enough to use),
2. XML was designed to capture all aspects of information representation, and
3. XML was designed by information processing community*

* Except the relational database community.
Can we query all data available through XML views? → we need *language* to express “all” queries...

XML Query Language* (XQuery, W3C 2007)

1. Fully based on XML (including XML Schema typing).
2. Data is considered constant.
3. Includes SQL-style relational queries.
4. Semantics is functional and formalized (W3C 2007).

* Now *with* the relational data base community.
Can we query all data available through XML views?  
→ we need language to express “all” queries...

XML Query Language* (XQuery, W3C 2007)

1. Fully based on XML (including XML Schema typing).
2. Data is considered constant.
3. Includes SQL-style relational queries.
4. Semantics is functional and formalized (W3C 2007).

* Now with the relational data base community.
Can we query all data available through XML views? 
→ we need *language* to express “all” queries...

---

**XML Query Language** (XQuery, W3C 2007)

1. Fully based on XML (including XML Schema typing).
2. Data is considered constant.
3. Includes SQL-style relational queries.
4. Semantics is functional and formalized (W3C 2007).

* Now *with* the relational data base community.
Project: effectively useable XML views of everything

Virtualize,
Let no one else’s work evade your eyes,
Remember why the good Lord made your eyes,
So don’t shade your eyes,
But virtualize, virtualize, virtualize...
Only be sure always to call it please ... search.

... with apologies to Tom Lehrer.

Virtual XML: XQuery + effective adaptive XML views

http://www.research.ibm.com/virtualxml/
Q

Virtualize,
Let no one else’s work evade your eyes,
Remember why the good Lord made your eyes,
So don’t shade your eyes,
But virtualize, virtualize, virtualize...
Only be sure always to call it please ... search.

... with apologies to Tom Lehrer.

A

Virtual XML: XQuery + effective adaptive XML views

http://www.research.ibm.com/virtualxml/

Project: **effectively useable** XML views of everything
Project: **effectively useable** XML views of everything

**Q**

*Virtualize,*
Let no one else’s work evade your eyes,
Remember why the good Lord made your eyes,
So don’t shade your eyes,
But virtualize, virtualize, virtualize...
Only be sure always to call it please ... *search.*

...with apologies to Tom Lehrer.

**A**

Virtual XML: XQuery + **effective** adaptive XML views

Join query

\[
\text{for } \$a \text{ in } \text{$input/descendant::q1} \\
\text{for } \$b \text{ in } \text{$input/descendant::q2} \\
\text{where } \$a = \$b \\
\text{return } \$a
\]

1. “For every element q1 (ranged over by variable $a$) consider every element q2 (ranged over by variable $b$) then for every pair test whether the two have the same value and if so return the q1-element.”

2. “For every pair of elements q1 (ranged over by variable $a$), and q2 (ranged over by $b$), where the values (of $a$ and $b$) are equal, return that element (from $a$).”
### Join query

```xml
for $a in $input/descendant::q1
for $b in $input/descendant::q2
where $a = $b
return $a
```

1. “For every element q1 (ranged over by variable $a) consider every element q2 (ranged over by variable $b) then for every pair test whether the two have the same value and if so return the q1-element.”

2. “For every pair of elements q1 (ranged over by variable $a), and q2 (ranged over by $b), where the values (of $a and $b) are equal, return that element (from $a).”
Join query

for $a$ in $\text{input/descendant::q1}$
for $b$ in $\text{input/descendant::q2}$
where $a = b$
return $a$

1. “For every element $q_1$ (ranged over by variable $a$) consider every element $q_2$ (ranged over by variable $b$) then for every pair test whether the two have the same value and if so return the $q_1$-element.”

2. “For every pair of elements $q_1$ (ranged over by variable $a$), and $q_2$ (ranged over by $b$), where the values (of $a$ and $b$) are equal, return that element (from $a$).”

Kristoffer Rose @ NeilFest
Does it matter?!?

No. . . . unless we want to observe operational behaviour . . .

- XML updates (XQuery Update, W3C ongoing).
- Imperative control (XQueryP, ongoing).
- Concurrency.
- Exceptions.
- etc.
Does it matter?!?

No . . . . unless we want to **observe operational behaviour** . . .

- XML updates (XQuery Update, W3C ongoing).
- Imperative control (XQueryP, ongoing).
- Concurrency.
- Exceptions.
- *etc.*
It’s... a monadic comprehension!

Cf. XQuery formal semantics “core normalization”

```
for $a in $input/descendant::q1 return
for $b in $input/descendant::q2 return
if ($a=$b) then $a else ()
```

- Forward value flow
- Variables directly iterate over collections
- Nesting by recursion
It's... a monadic comprehension!

Cf. XQuery formal semantics “core normalization”

\[
\text{for } a \text{ in } \text{input}/\text{descendant}::q1 \text{ return }
\text{for } b \text{ in } \text{input}/\text{descendant}::q2 \text{ return }
\text{if } (a=b) \text{ then } a \text{ else } ()
\]

- Forward value flow
- Variables directly iterate over collections
- Nesting by recursion
It’s... relational algebra!

Cf. traditional relational database translation

Map{#a}
  (Select{#a = #b}
   (MapConcat
    {Map{[b:ID]}
     (TreeJoin[descendant::q2]
      (#input))}
    (Map{[a:ID]}
     (TreeJoin[descendant::q1]
      (#input))))

- Backward “tuple flow”
- Fields refer to collection values
- Nesting by tuple concatenation
It’s... relational algebra!

Cf. traditional relational database translation

Map{#a}
  (Select{#a = #b}
   (MapConcat
     {Map{[b:ID]}
      (TreeJoin[descendant::q2]
       (#input))
     (Map{[a:ID]}
      (TreeJoin[descendant::q1]
       (#input))))
  )

- Backward “tuple flow”
- Fields refer to collection values
- Nesting by tuple concatenation
A: It’s... both!

**Combined approach**

```xml
for $t_5$ in
  for $t_4$ in
    for $t_2$ in
      for $t_0$ in [input:$input]
        return
          for $t_1$ in $t_0$.input/descendant::q1 return [a:$t_1]++ $t_0
          return
            for $t_3$ in $t_2$.input/descendant::q2 return [b:$t_3]++ $t_2
            return (if ($t_4.a = $t_4.b) then $t_4 else ( ))
        return $t_5.a
```

- Retains explicit tuples and
- is consistent with XQuery specification and database tradition!
A: It’s... both!

Combined approach

```xml
for $t_5$ in
  for $t_4$ in
    for $t_2$ in
      for $t_0$ in [input:$input]
        return
          for $t_1$ in $t_0$.input/descendant::q1 return [a:$t_1] ++ $t_0
    return
      for $t_3$ in $t_2$.input/descendant::q2 return [b:$t_3] ++ $t_2
  return (if ($t_4.a = $t_4.b) then $t_4 else ())
return $t_5.a
```

- Retains explicit tuples and
- is consistent with XQuery specification and database tradition!
A: It’s… both (Peyton Jones and Wadler)!

Combined approach with field variables

```xquery
for [$b_5, a_5, input_5] in
  for [$b_4, a_4, input_4] in
    for [$a_2, input_2] in
      for [input_0] in [input]
        return
        for $a_1 in input_0/descendant::q1 return [$a_1, input_0]
      return
    return
  for $b_3 in input_2/descendant::q2 return [$b_3, a_2, input_2]
return (if ($a_4 = $b_4) then [$b_4, a_4, input_4] else ())
return $a_5
```
XQuery Syntax

\[ M ::= \text{declare function } f(\$x_1 \text{ as } t_1, \ldots, \$x_n \text{ as } t_n) \text{ as } \{E\}; M \]

\[ E ::= F \mid () \mid E_1,E_2 \mid \$x \mid \text{if (}E_1\text{) then }E_2\text{ else }E_3 \]
\[ \mid f(E_1, \ldots, E_n) \mid \text{element } q \{E\} \mid E/s \mid \ell \]
\[ \mid \text{insert node } \{E_1\} \text{ into } \{E_2\} \mid \text{delete node } \{E\} \]

\[ F ::= \text{for } \$x \text{ in } E F \mid \text{let } \$x := E F \]
\[ \mid \text{where } E F \mid \text{order by } E F \mid \text{return } E \]

\[ s ::= \text{child}::q \mid \text{descendant}::q \mid \cdots \]
Values: Literals ℓ
XML node ids referencing an XML store σ
Sequences ⃗v of the above

Types: Contains values $T[t]_\sigma$ (items and sequences)
Checking: $\Gamma \vdash E : xt$
Record subtyping: $[r_1; r_2] \leq: [r_1]$ if $t \leq: \{t\}$

Semantics: $\Sigma; \sigma \vdash E \Rightarrow v'; \sigma'$ iff $\Sigma; \sigma \vdash \mathcal{X}[F] \Rightarrow v'; \sigma'$
XQuery details

**Values:** *Literals* \( \ell \)
- XML node ids referencing an *XML store* \( \sigma \)
- Sequences \( \vec{v} \) of the above

**Types:** Contains values \( T[t]_\sigma \) (items and sequences)
- Checking: \( \Gamma \vdash E : xt \)
- Record subtyping: \( [r_1; r_2] \leq: [r_1] \) if \( t \leq: \{t\} \)

**Semantics:** \( \Sigma; \sigma \vdash E \Rightarrow v'; \sigma' \) iff \( \Sigma; \sigma \vdash X[F] \Rightarrow v'; \sigma' \)
Core XQuery supporting tuples

\[
m ::= \text{declare function } f(\$x_1 \text{ as } t_1, \ldots, \$x_n \text{ as } t_n) \text{ as } t \{ e \}; \ m \mid e
\]
\[
e ::= \text{for } \$x \text{ in } e_1 \text{ return } e_2 \mid \text{order } \$x \text{ in } e_1 \text{ by } e_2
\]
\[
\mid \text{let } \$x := e_1 \text{ return } e_2 \mid \text{if } (e_1) \text{ then } e_2 \text{ else } e_3
\]
\[
\mid \$x \mid \ell \mid () \mid e_1, e_2 \mid \text{element } q \{ e \} \mid \$x/s
\]
\[
\mid \text{insert nodes } \{ e_1 \} \text{ into } \{ e_2 \} \mid \text{delete nodes } \{ e \}
\]
\[
\mid f(e_1, \ldots, e_n) \mid [a : e] \mid \$x.a \mid e_1 \text{++ } e_2
\]

Values as XQuery plus tuples:

- \([a : e]\) constructs tuple mapping a to value of e
- \(e_1 \text{++ } e_2\) concatenates tuples, no conflicts
- \(\$x.a\) returns value of a field in \$x tuple
Core XQuery supporting tuples

\[ m ::= \text{declare function } f (x_1 \text{ as } t_1, \ldots, x_n \text{ as } t_n) \text{ as } t \{e\}; \ m \mid e \]

\[ e ::= \text{for } x \text{ in } e_1 \text{ return } e_2 \mid \text{order } x \text{ in } e_1 \text{ by } e_2 \]
\[ \mid \text{let } x := e_1 \text{ return } e_2 \mid \text{if } (e_1) \text{ then } e_2 \text{ else } e_3 \]
\[ \mid x \mid \ell \mid () \mid e_1, e_2 \mid \text{element } q \{e\} \mid x/s \]
\[ \mid \text{insert nodes } \{e_1\} \text{ into } \{e_2\} \mid \text{delete nodes } \{e\} \]
\[ \mid f (e_1, \ldots, e_n) \mid [a : e] \mid x.a \mid e_1 ++ e_2 \]

Values as XQuery plus tuples:

- \([a : e]\) constructs tuple mapping \(a\) to value of \(e\)
- \(e_1 ++ e_2\) concatenates tuples, no conflicts
- \(x.a\) returns value of \(a\) field in \(x\) tuple
Core XQuery supporting tuples

\[ m ::= \text{declare function } f(\$x_1 \text{ as } t_1, \ldots, \$x_n \text{ as } t_n) \text{ as } t \{ e \}; m \mid e \]

\[ e ::= \text{for } \$x \text{ in } e_1 \text{ return } e_2 \mid \text{order } \$x \text{ in } e_1 \text{ by } e_2 \]
\[ \mid \text{let } \$x ::= e_1 \text{ return } e_2 \mid \text{if } (e_1) \text{ then } e_2 \text{ else } e_3 \]
\[ \mid \$x \mid \ell \mid () \mid e_1, e_2 \mid \text{element } q \{ e \} \mid \$x/s \]
\[ \mid \text{insert } \text{nodes } \{ e_1 \} \text{ into } \{ e_2 \} \mid \text{delete } \text{nodes } \{ e \} \]
\[ \mid f(e_1, \ldots, e_n) \mid [a : e] \mid \$x.a \mid e_1++] e_2 \]

Values as XQuery plus *tuples*:

- \([a : e]\) constructs tuple mapping \(a\) to value of \(e\)
- \(e_1++] e_2\) concatenates tuples, **no conflicts**
- \(\$x.a\) returns value of \(a\) field in \(\$x\) tuple
Types: As XQuery plus tuple types \([a_1 : x_{t1}, \ldots, a_n : x_{tn}]\)
where \(e_1 \perps e_2\) with a common field is only typable
when the two fields have identical types (coming up)

Semantics: \(\Sigma; \sigma \vdash e \Rightarrow v'; \sigma'\)
Types: As XQuery plus tuple types \([a_1 : xt_1, \ldots, a_n : xt_n]\)
where \(e_1 \oplus e_2\) with a common field is only typable when the two fields have identical types (coming up)

Semantics: \(\Sigma; \sigma \vdash e \Rightarrow v'; \sigma'\)
Types: As XQuery plus tuple types $[a_1 : xt_1, \ldots, a_n : xt_n]$ where $e_1 + e_2$ with a common field is only typable when the two fields have identical types (coming up)

Semantics: $\Sigma; \sigma \vdash e \Rightarrow v'; \sigma'$
“Normalization” of FLWOR to tuples

\[ \mathcal{X}[E] = \text{for } \$\text{dot} \text{ in } \text{input} : \$\text{input} \text{ return } \mathcal{X}[E]_{\$\text{dot}} \]  

- \( \mathcal{X}[E]_{\$t} \) evaluates as \( E \) provided \( \varrho \) maps variables to fields in \( \$t \)
- \( \mathcal{X}^*[F]_{\$e} \) accumulates \( F \)-clauses iterating over \( (\varrho-) \)tuples from \( e \)

Our example can be assembled as \( \mathcal{X}[F_1] = e_4 \) as follows:

<table>
<thead>
<tr>
<th>( i )</th>
<th>( F_i )</th>
<th>( e_i = \mathcal{X}^*[F_i]e_{(i-1)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>\text{input} : $\text{input} ]</td>
</tr>
<tr>
<td>1</td>
<td>( F_2 )</td>
<td>\text{for } $t_0 \text{ in } e_0 \text{ return for } $t_1 \text{ in } \ldots \text{ return } [a : $t_1] ++ $t_0</td>
</tr>
<tr>
<td>2</td>
<td>( F_3 )</td>
<td>\text{for } $t_2 \text{ in } e_1 \text{ return for } $t_3 \text{ in } \ldots \text{ return } [b : $t_3] ++ $t_2</td>
</tr>
<tr>
<td>3</td>
<td>( F_4 )</td>
<td>\text{where } a = b \text{ for } $t_4 \text{ in } e_2 \text{ return if } (a = b) \text{ then } $t_4 \text{ else } ()</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>\text{for } $t_5 \text{ in } e_3 \text{ return } $t_5 . a</td>
</tr>
</tbody>
</table>

Kristoffer Rose @ NeilFest Core Relational + Functional + Imperative
The subtyping relation $\leq$ is the transitive homomorphic closure of

$[r_1; r_2] \leq: [r_1]$ \hspace{1cm} $t \leq: \{t\}$

Warning!
Not sound on core+tuples!

- $|a:1|++|a:2|$
- let $x := (\text{for } w \text{ in } (1,2) \text{ return } |a:w|)$
  return for $y_1$ in $x$ return for $y_2$ in $x$ return $(y_1++y_2)$
Subtyping

The *subtyping* relation $\leq$ is the transitive homomorphic closure of

$$[r_1; r_2] \leq: [r_1]$$

$$t \leq: \{t\}$$

**Warning!**

Not sound on core+tuple!

1. $[a:1] + [a:2]$
2. let $x := (\text{for } w \text{ in } (1,2) \text{ return } [a:w])$
   return for $y_1 \text{ in } x$ return for $y_2 \text{ in } x$ return $(y_1 + y_2)$
The *subtyping* relation $\leq$: is the transitive homomorphic closure of

$$[r_1; r_2] \leq: [r_1] \quad t \leq: \{t\}$$

---

**Warning!**

Not sound on core+tuples!

1. $[a:1] ++ [a:2]$

2. let $x := (for \ w in (1,2) return [a:w])$
   return for $y_1$ in $x$ return for $y_2$ in $x$ return ($y_1 ++ y_2$)
Observations:

- XQuery has *stack discipline*
- \( \mathcal{X}[E] \) *never mixes sequences and tuples*
- Implies core has *unique field value property*

**Theorem**

*For every XQuery E with typing E : xt*

1. \( \mathcal{X}[E] \) is linear
2. \( \mathcal{X}[E] : xt \)
3. \( \Sigma, \sigma \vdash \mathcal{X}[E] \Rightarrow \forall \prime m \sigma' \) implies \( xt' \in T[t]_{\sigma'} \)

This may be trivial with field variables?
Observations:
- XQuery has *stack discipline*
- \( \mathcal{X}[E] \) never mixes sequences and tuples
- Implies core has *unique field value property*

Theorem

For every XQuery \( E \) with typing \( E : xt \)

1. \( \mathcal{X}[E] \) is linear
2. \( \mathcal{X}[E] : xt \)
3. \( \Sigma, \sigma \vdash \mathcal{X}[E] \Rightarrow \forall' m\sigma' \text{ implies } xt' \in T[t]_{\sigma'} \)

This may be trivial with field variables?
Linearity and type preservation

Observations:
- XQuery has *stack discipline*
- \( \mathcal{X}[E] \) never mixes sequences and tuples
- Implies core has *unique field value property*

Theorem

*For every XQuery* \( E \) *with typing* \( E : xt \)

1. \( \mathcal{X}[E] \) is linear
2. \( \mathcal{X}[E] : xt \)
3. \( \Sigma, \sigma \vdash \mathcal{X}[E] \Rightarrow \forall' m \sigma' \) implies \( xt' \in T[t]_{\sigma'} \)

This may be trivial with field variables?
1 XQuery
   - XML & XQuery & Virtualization
   - Relational vs Functional
   - Syntax & Semantics
   - Types

2 Relational Algebra
   - Syntax & Semantics
   - Compilation
   - Rewritings

3 Towards synthesis
   - Join rewrite

4 Conclusions & Lessons
   - Related work
What is it?

Traditional relational algebra:

**Syntax:** \( p ::= \text{ID} | \text{Empty}() | \text{Sequence}(p_1, p_2) | \text{Scalar}[v]() \\
| \text{Select}\{p_1\}(p_2) | \text{OrderBy}\{p_1\}(p_2) \\
| \text{Map}\{p_1\}(p_2) | \text{MapConcat}\{p_1\}(p_2) \\
| \text{Conditional}(p_1, p_2, p_3) | \text{Call}[f](p_1, \ldots, p_n) \\
| [a:p] | \#a \\
| \text{Element}[q](p) | \text{TreeJoin}[s](p) \\
| \text{InsertInto}(p_1, p_2) | \text{Delete}(p) \\

**Values:** As core w/tuples

**Types:** As core w/tuples

**Checking:** \( p : t_1 \rightarrow t_2 \)
What is it?

Traditional relational algebra with XML “tree” operator:

**Syntax:**

\[ p ::= \text{ID} | \text{Empty}() | \text{Sequence}(p_1, p_2) | \text{Scalar}[v]() \\
| \text{Select}\{p_1\}(p_2) | \text{OrderBy}\{p_1\}(p_2) \\
| \text{Map}\{p_1\}(p_2) | \text{MapConcat}\{p_1\}(p_2) \\
| \text{Conditional}(p_1, p_2, p_3) | \text{Call}[f](p_1, \ldots, p_n) \\
| [a:p] | \#a \\
| \text{Element}[q](p) | \text{TreeJoin}[s](p) \\
| \text{InsertInto}(p_1, p_2) | \text{Delete}(p) \]

**Values:** As core w/tuples

**Types:** As core w/tuples

**Checking:** \[ p : t_1 \rightarrow t_2 \]

---

Kristoffer Rose @ NeilFest Core Relational+Functional+Imperative
What is it?

Traditional relational algebra with XML “tree” operator and effects:

**Syntax:**\thashline
\( p ::= \text{ID} \mid \text{Empty}() \mid \text{Sequence}(p_1, p_2) \mid \text{Scalar}[v]() \)
\( \mid \text{Select}\{p_1\}(p_2) \mid \text{OrderBy}\{p_1\}(p_2) \)
\( \mid \text{Map}\{p_1\}(p_2) \mid \text{MapConcat}\{p_1\}(p_2) \)
\( \mid \text{Conditional}(p_1, p_2, p_3) \mid \text{Call}[f](p_1, \ldots, p_n) \)
\( \mid [a:p] \mid \#a \)
\( \mid \text{Element}[q](p) \mid \text{TreeJoin}[s](p) \)
\( \mid \text{InsertInto}(p_1, p_2) \mid \text{Delete}(p) \)

**Values:** As core w/tuples

**Types:** As core w/tuples

**Checking:** \( p : t_1 \rightarrow t_2 \)
Normalization of relational algebra to \texttt{core+tuples}

\[
\begin{align*}
\mathcal{A}[\text{id}]_{st} &= \mathit{t} \\
\mathcal{A}[\text{Sequence}(p_1, p_2)]_{st} &= (\mathcal{A}[p_1]_{st}, \mathcal{A}[p_2]_{st}) \\
\mathcal{A}[\text{Empty()}]_{st} &= () \\
\mathcal{A}[\text{Scalar}[\ell]() ]_{st} &= \ell \\
\mathcal{A}[\text{Element}[q](p)]_{st} &= \text{element } q \{ \mathcal{A}[p]_{st} \} \\
\mathcal{A}[\text{Select}\{p_1\}(p_2)]_{st} &= \text{for } \mathit{t_1} \text{ in } \mathcal{A}[p_2]_{st} \text{ return if } (\mathcal{A}[p_1]_{t_1}) \text{ then } \mathit{t_1} \text{ else } () \\
\mathcal{A}[\text{TreeJoin}[s](p)]_{st} &= \text{for } \mathit{t_1} \text{ in } \mathcal{A}[p]_{st} \text{ return } \mathit{t_1}/s \\
\mathcal{A}[\text{Map}\{p_1\}(p_2)]_{st} &= \text{for } \mathit{t_1} \text{ in } \mathcal{A}[p_2]_{st} \text{ return } \mathcal{A}[p_1]_{\mathit{t_1}} \\
\mathcal{A}[\text{MapConcat}\{p_1\}(p_2)]_{st} &= \text{for } \mathit{t_1} \text{ in } \mathcal{A}[p_2]_{st} \text{ return } \text{for } \mathit{t_2} \text{ in } \mathcal{A}[p_1]_{\mathit{t_1}} \text{ return } \mathit{t_1} + + \mathit{t_2} \\
\mathcal{A}[\text{OrderBy}\{p_1\}(p_2)]_{st} &= \text{order } \mathit{t_1} \text{ in } \mathcal{A}[p_2]_{st} \text{ by } \mathcal{A}[p_1]_{\mathit{t_1}} \\
\mathcal{A}[\text{Conditional}(p_1, p_2, p_3)]_{st} &= \text{if } (\mathcal{A}[p_1]_{st}) \text{ then } \mathcal{A}[p_2]_{st} \text{ else } \mathcal{A}[p_3]_{st} \\
\mathcal{A}[\text{Call}[f](p_1, \ldots, p_n)]_{st} &= f(\mathcal{A}[p_1]_{st}, \ldots, \mathcal{A}[p_n]_{st}) \\
\mathcal{A}[[a : p]]_{st} &= [a : \mathcal{A}[p]_{st}] \\
\mathcal{A}[\#a]_{st} &= \mathit{t}.a
\end{align*}
\]
How does it fit?

Equivalences

XQuery \rightarrow \text{Compilation (traditional)} \rightarrow \text{Algebra}

\downarrow \downarrow

\text{Normalization (definition)} \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow

\text{Semantics (definition)}

\text{Core+tuples}

Results:
- Semantic equivalence
- Type translation (with constrained soundness)

Benefit: PL for DB! Specifically:
- Type-based Algebraic rewritings
- Functional rewritings possible after join optimizations
- Functional approach to analyses of side effects can be used
How does it fit?

### Equivalences

- **XQuery** → **Compilation (traditional)** → **Algebra**
- **Core+tuples**
  - **Normalization (definition)**
  - **Semantics (definition)**

### Results:

- Semantic equivalence
- Type translation (with constrained soundness)

### Benefit:

- PL for DB!

**Specifically:**

- Type-based Algebraic rewritings
- Functional rewritings possible after join optimizations
- Functional approach to analyses of side effects can be used
How does it fit?

Equivalences

```
XQuery  Compilation (traditional)  Algebra
   \______________\                  \______________\  \\
  Normalization (definition)       Semantics (definition)
   ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑∪
How does it fit?

Results:
- Semantic equivalence
- Type translation (with constrained soundness)

Benefit: PL for DB! Specifically:
- Type-based Algebraic rewritings
- Functional rewritings possible after join optimizations
- Functional approach to analyses of side effects can be used
Traditional compilation to flow form is $C[E]$ defined as follows on FLWOR:

\[
\begin{align*}
  i & \quad F_i & \quad p_i = C^*\left[F_i\right]: \ldots \\
 1 & \text{for } a \in \ldots F_2 & \quad \text{Map}\{[a:\text{ID}]\}(\ldots) \\
 2 & \text{for } b \in \ldots F_3 & \quad \text{MapConcat}\{\text{Map}\{[b:\text{ID}]\}(\ldots)\}(p_1) \\
 3 & \text{where } a=b F_4 & \quad \text{Select}\{\#a=\#b\}(p_2) \\
 4 & \text{return } a & \quad \text{Map}\{\#a\}(p_3)
\end{align*}
\]
Equivalence

Observations:
- Compilation $C$ from XQuery follows the exact pattern of $X$
- $A$ trivially preserves linearity and types

Theorem

For any environment $\Sigma$, stores $\sigma, \sigma'$, XQuery expression $E$, variable $\$t$, field name assignment $\varrho$ defined for all free variables in $E$, and value $v'$,

$$\Sigma, \sigma \vdash A[C[E]^{\varrho}]_{\$t} \Rightarrow v, \sigma' \iff \Sigma, \sigma \vdash X[E]^{\varrho}_{\$t} \Rightarrow v, \sigma'$$
Join rewrite

Cf. optimized join example

Map{#a}
  (Join{#a = #b}
    (Map{[b:ID]}
      (TreeJoin[descendant::*:q2]
       (#input))),
    (Map{[a:ID]}
      (TreeJoin[descendant::*:q1]
       (#input))))

Example

Kristoffer Rose @ NeilFest  Core Relational+Functional+Imperative
Extended algebra

Syntax:  \( p ::= \text{BaseOperators} | \text{Product}(p_1, p_2) \\
| \text{Join}\{p_1\}(p_2, p_3) | \text{LOuterJoin}[q]\{p_1\}(p_2, p_3) \\
| \text{OMap}[q](p_1) | \text{OMapConcat}[q]\{p_1\}(p_2) \\
| \text{GroupBy}[q_{\text{Agg}}, q_{\text{Index}}, q_{\text{N}}]\{p_1\}\{p_2\}(p_3) \\
| \text{MapIndex}[q](p_1) | p_1 \circ p_2 \)

Values:  As core w/tuples

Types:   As core w/tuples

Checking:  \( p : t_1 \rightarrow t_2 \)

Semantics:  \ldots
Rewritings

\[
p_1 \equiv \text{ID : } [] \rightarrow []
\]

\[
p_1 : [r_1] \rightarrow \{[r_1; r_3]\} \quad p_2 : [r_0; r_1] \rightarrow [r_1]
\]

\[
\text{MapConcat}\{p_1\}(p_2) \equiv p_1 : [r_0; r_1] \rightarrow \{[r_1; r_3]\}
\]

\[
p_1 : [r_1] \rightarrow \{[r_1; r_2]\} \quad p_2 : [r_1] \rightarrow \{[r_3]\}
\]

\[
\text{MapConcat}\{p_2\}(p_1) \equiv \text{Product}(p_1, p_2) : [r_1] \rightarrow \{[r_1; r_2; r_3]\}
\]

\[
p_2 : t \rightarrow \{[r_2]\} \quad p_3 : t \rightarrow \{[r_3]\} \quad p_1 : [r_2; r_3] \rightarrow \text{Item}
\]

\[
\text{Select}\{p_1\}(\text{Product}(p_2, p_3)) \equiv \text{Join}\{p_1\}(p_2, p_3) : t \rightarrow \{[r_2; r_3]\}
\]

\[
p_1 : t \rightarrow \{[r_1]\} \quad p_2 : [r_1] \rightarrow \{[r_2]\}
\]

\[
\text{MapConcat}\{\text{OMap}[n](p_2)\}(p_1) \equiv \text{OMapConcat}[n]\{p_2\}(p_1) : t \rightarrow \{[r_1; r_2; n]\}
\]

\[
p_3 : [r_0; r_1] \rightarrow \{[r_0; r_3]\} \quad p_2 : [r_0] \rightarrow \{[r_2]\} \quad p_1 : [r_0; r_2; r_3] \rightarrow \{\text{Item}\}
\]

\[
p_1 \text{ and } p_2 \text{ commute}
\]

\[
\text{OMapConcat}[n]\{\text{Join}\{p_1\}(\text{ID}, p_2)\}(p_3) \equiv \text{LOuterJoin}[n]\{p_1\}(p_3, p_2) : [r_0; r_1] \rightarrow \{[r_0; r_2; r_3; n]\}
\]

\[
p_1 : \{[r_0]\} \rightarrow \{\text{Item}\} \quad p_2 : [r_1] \rightarrow r_0 \quad p_3 : t \rightarrow \{[r_1]\}
\]

\[
[x : p_1 \circ (\text{Map}[p_2](p_3))] \Rightarrow \text{GroupBy}[x, \emptyset, n]\{p_1\}\{p_2\}(\text{OMap}[n](p_3)) : t \rightarrow [x : \{\text{Item}\}]
\]

\[
p_4 : t \rightarrow \{[r_0]\} \quad p_3 : [r_0] \rightarrow [r_3; i : \{\text{Item}\}; n : \{\text{Item}\}] \quad p_2 : [r_3; i : \{\text{Item}\}] \rightarrow [r_2] \quad p_1 : \{[r_2]\} \rightarrow \{\text{Item}\}
\]

\[
\text{MapConcat}\{\text{GroupBy}[x, \emptyset, n]\{p_1\}\{p_2\}(p_3)\}(p_4)
\]

\[
\Rightarrow \text{GroupBy}[x, i, n]\{p_1\}\{p_2\}(\text{MapConcat}[p_3](\text{MapIndex}[i](p_4))) : t \rightarrow \{[r_0; x : \{\text{Item}\}; i : \{\text{Item}\}]\}
\]
1 XQuery
   • XML & XQuery & Virtualization
   • Relational vs Functional
   • Syntax & Semantics
   • Types
2 Relational Algebra
   • Syntax & Semantics
   • Compilation
   • Rewritings
3 Towards synthesis
   • Join rewrite
4 Conclusions & Lessons
   • Related work
Algebraic join rewrite

\[
p_1 : [r_1] \rightarrow \{[r_1; r_2]\} \quad p_2 : [r_1] \rightarrow \{[r_3]\}
\]

\[
\text{MapConcat}\{p_2\}(p_1) \equiv \text{Product}(p_1, p_2)
\]

\[
p_2 : t \rightarrow \{[r_2]\} \quad p_3 : t \rightarrow \{[r_3]\} \quad p_1 : [r_2; r_3] \rightarrow \text{Item}
\]

\[
\text{Select}\{p_1\}(\text{Product}(p_2, p_3)) \equiv \text{Join}\{p_1\}(p_2, p_3)
\]
Core join rewrite?

Join pattern...

for $t_4$ in
  for $t_2$ in
    for $t_0$ in $e_0()$
      return
        for $t_1$ in $e_1(t_0)$ return $[f_1:t_1] + + t_0$
      return
        for $t_3$ in $e_3(t_0)$ return $[f_3:t_3] + + t_2$
    return
  join
if $(e_4(t_4))$ then $t_4$ else ()
Core join rewrite?

Join pattern...

for $t_4$ in
  for $t_2$ in
    for $t_0$ in e_0()
      return
      for $t_1$ in e_1(t_0) return [f_1:$t_1] ++ $t_0
      return nonestig
      for $t_3$ in e_3(t_0) return [f_3:$t_3] ++ $t_2
      return join
      if (e_4(t_4)) then $t_4$ else ()
Core join rewrite?

Join pattern...

```
for $t_4$ in
  for $t_2$ in
    for $t_0$ in e_0()
      return
        for $t_1$ in e_1(t_0) return [f_1:$t_1] ++ $t_0
        return nonesting
        for $t_3$ in e_3(t_0) return [f_3:$t_3] ++ $t_2
      return join
    if (e_4(t_4)) then t_4 else ()
```
1. XQuery
   - XML & XQuery & Virtualization
   - Relational vs Functional
   - Syntax & Semantics
   - Types
2. Relational Algebra
   - Syntax & Semantics
   - Compilation
   - Rewritings
3. Towards synthesis
   - Join rewrite
4. Conclusions & Lessons
   - Related work
Conclusions

Achieved:

- New functional and relational semantics for XQuery
- Operationally faithful so it allows control extensions
- Type system that propagates properly “between the worlds” (and cute new variation of subtyping & records/tuples)
- Improved framework for functional and relational optimizations, especially type-base algebraic rewriting

Future work:

- Developing a properly combined functional and relational compiler
- Expressing combined rewrites with higher order rewriting
Conclusions

Achieved:

- New functional and relational semantics for XQuery
- Operationally faithful so it allows control extensions
- Type system that propagates properly “between the worlds” (and cute new variation of subtyping & records/tuples)
- Improved framework for functional and relational optimizations, especially type-base algebraic rewriting

Future work:

- Developing a properly combined functional and relational compiler
- Expressing combined rewrites with higher order rewriting
Reflections

- Reexplain what others have done.
- Make others reexplain and improve what you’ve done.
- Impose yourself on any research field.
- Talk research to everybody.
- Everything is data!
- Build stuff.
- Be surprised over where you can sell your research...
- Have fun.
Reflections

- Reexplain what others have done.
- Make others reexplain and improve what you’ve done.
  - Impose yourself on any research field.
  - Talk research to everybody.
  - Everything is data!
  - Build stuff.
  - Be surprised over where you can sell your research...
- Have fun.
Reflections

- Reexplain what others have done.
- Make others reexplain and improve what you’ve done.
- Impose yourself on any research field.
  - Talk research to everybody.
  - Everything is data!
  - Build stuff.
  - Be surprised over where you can sell your research...
- Have fun.
Reflections

- Reexplain what others have done.
- Make others reexplain and improve what you’ve done.
- Impose yourself on any research field.
- Talk research to everybody.
- Everything is data!
- Build stuff.
- Be surprised over where you can sell your research...
- Have fun.
Reflections

- Reexplain what others have done.
- Make others reexplain and improve what you’ve done.
- Impose yourself on any research field.
- Talk research to everybody.
- Everything is data!
- Build stuff.
- Be surprised over where you can sell your research...
- Have fun.
Reflections

- Reexplain what others have done.
- Make others reexplain and improve what you’ve done.
- Impose yourself on any research field.
- Talk research to everybody.
- Everything is data!
- Build stuff.
  - Be surprised over where you can sell your research...
- Have fun.
Reflections

- Reexplain what others have done.
- Make others reexplain and improve what you’ve done.
- Impose yourself on any research field.
- Talk research to everybody.
- Everything is data!
- Build stuff.
- Be surprised over where you can sell your research...
- Have fun.
Reflections

- Reexplain what others have done.
- Make others reexplain and improve what you’ve done.
- Impose yourself on any research field.
- Talk research to everybody.
- Everything is data!
- Build stuff.
- Be surprised over where you can sell your research...
- Have fun.
Related work

- Using monadic comprehensions for database languages [1, 5], especially Kleisli [6, 2] which does not, however, express join optimizations functionally.
- Mainstream work on database management system optimizers to this day rely on tuple-based algebras [3, 4].
- Work on reconciliation of record subtyping and record concatenation.
- Recent: Peyton Jones and Wadler on encoding GroupBy with comprehensions and field variables.
Thank you!
5 Backup – Core+tuples
   * Typing of core+tuples
   * Core+tuples semantics

6 Backup – XQuery
   * XQuery type rules
   * Normalization of XQuery to core+tuples

7 Backup – Algebra
   * Algebra type rules
   * Compiling XQuery into relational algebra
   * Equivalence details

8 Backup – Algebraic rewrites
   * Extended algebra type rules
Type Semantics

The *semantics* of a type in a store, $\mathcal{T}[t]\sigma$, is defined as follows:

- $\mathcal{T}[\text{Item}]\sigma$ contains all node ids bound in $\sigma$ and all literal values;
- $\mathcal{T}[\{t\}]\sigma$ is the set of all finite sequences of elements of $\mathcal{T}[t]\sigma$; a single element of $\mathcal{T}[t]\sigma$ belongs to this set, and is equivalent to a singleton sequence;
- $\mathcal{T}[[a_1 : t_1; \ldots; a_n : t_n]]\sigma$ is the set of all functions that, for $i$ in $1 \ldots n$, map $a_i$ to an element of $\mathcal{T}[t_i]\sigma$; an element of $\mathcal{T}[[a_1 : t_1; \ldots; a_n : t_n]]\sigma$ may also be defined on any field name that is not specified in the type;
- $\mathcal{T}[(t_1, \ldots, t_n) \rightarrow t]\sigma$ is the set of all functions that, applied to $n$ arguments in $\mathcal{T}[t_1]\sigma \ldots \mathcal{T}[t_n]\sigma$, return a value in $\mathcal{T}[t]\sigma$. 
Typing of core+tuples

\[\Gamma(x) = t \quad \Gamma \vdash e_1 : t_1 \quad (\Gamma, x : t) \vdash e_2 : t_2\]

\[\Gamma \vdash x : t \quad \Gamma \vdash e_1 : t_1 \quad \Gamma \vdash \text{let } x := e_1 \text{ return } e_2 : t_2\]

\[\Gamma \vdash \ell : \text{Item}\]

\[\Gamma \vdash \ell : \{t\}\]

\[\Gamma \vdash e_1 : \{t\} \quad \Gamma \vdash e_2 : \{t\}\]

\[\Gamma \vdash \text{for } x \text{ in } e_1 \text{ return } e_2 : \{t_2\}\]

\[\Gamma \vdash e_1 : \{t_1\} \quad \Gamma \vdash e_2 : \{t_2\} \quad \Gamma \vdash e_3 : \{t_2\}\]

\[\Gamma \vdash \text{order } x \text{ in } e_1 \text{ by } e_2 : \{t_1\}\]

\[\Gamma \vdash e : \{\text{Item}\}\]

\[\Gamma \vdash \ell : \text{Item}\]

\[\Gamma \vdash \text{element } q \{e\} : \text{Item}\]

\[\Gamma \vdash e : x t\]

\[\Gamma \vdash \ell : \{a : x\}\]

\[\Gamma \vdash e : [a : x]\]

\[\Gamma \vdash \ell : [a : x]\]

\[\Gamma \vdash f(e_1, \ldots, e_n) : x t\]

\[\Gamma \vdash f(e_1, \ldots, e_n) : x t\]

\[\Gamma \vdash \text{if } (e_1) \text{ then } e_2 \text{ else } e_3 : t_2\]

\[\Gamma \vdash \text{order } x \text{ in } e_1 \text{ by } e_2 : \{t_1\}\]

\[\Gamma \vdash e_1 : \{t_1\} \quad \Gamma \vdash e_2 : \{t_2\}\]

\[\Gamma \vdash \text{order } x \text{ in } e_1 \text{ by } e_2 : \{t_1\}\]

\[\Gamma \vdash f(e_1, \ldots, e_n) : x t\]

\[\Gamma \vdash \text{if } (e_1) \text{ then } e_2 \text{ else } e_3 : t_2\]

\[\Gamma \vdash \text{order } x \text{ in } e_1 \text{ by } e_2 : \{t_1\}\]

\[\Gamma \vdash f(e_1, \ldots, e_n) : x t\]

\[\Gamma \vdash \text{if } (e_1) \text{ then } e_2 \text{ else } e_3 : t_2\]

\[\Gamma \vdash \text{order } x \text{ in } e_1 \text{ by } e_2 : \{t_1\}\]

\[\Gamma \vdash f(e_1, \ldots, e_n) : x t\]

\[\Gamma \vdash \text{if } (e_1) \text{ then } e_2 \text{ else } e_3 : t_2\]
Core+tuples semantics (1 of 2)

\[
\begin{align*}
\Sigma; \sigma \vdash e_1 \Rightarrow (v_1, \ldots, v_n); \sigma_1 & \quad \forall i \in 1..n: (\Sigma, \$x \triangleright v_i); \sigma_i \vdash e_2 \Rightarrow v'_i; \sigma_{i+1} \\
\Sigma; \sigma \vdash \text{for } \$x \text{ in } e_1 \text{ return } e_2 \Rightarrow (v'_1, \ldots, v'_n); \sigma_{n+1} \\
\Sigma; \sigma \vdash e_1 \Rightarrow v_1; \sigma_1 & \quad (\Sigma, \$x \triangleright v_1); \sigma_1 \vdash e_2 \Rightarrow v_2; \sigma_2 \\
\Sigma; \sigma \vdash \text{let } \$x := e_1 \text{ return } e_2 \Rightarrow v_2; \sigma_2 \\
\Sigma; \sigma \vdash e_1 \Rightarrow (v_1, \ldots, v_n); \sigma_1 & \quad \forall i \in 1..n: (\Sigma, \$x \triangleright v_i); \sigma_i \vdash e_2 \Rightarrow k_i; \sigma_{i+1} \quad \text{sort}(k_1, \ldots, k_n) = (m_1, \ldots, m_n) \\
\Sigma; \sigma \vdash \text{order } \$x \text{ in } e_1 \text{ by } e_2 \Rightarrow (v_{m_1}, \ldots, v_{m_n}); \sigma_{n+1} \\
\Sigma; \sigma \vdash e_1 \Rightarrow v_1; \sigma_1 & \quad v_1 \not= () \quad \Sigma; \sigma \vdash e_2 \Rightarrow v_2; \sigma_2 \\
\Sigma; \sigma \vdash \text{if } (e_1) \text{ then } e_2 \text{ else } e_3 \Rightarrow v_2; \sigma_2 \\
\Sigma; \sigma \vdash e_1 \Rightarrow (); \sigma_1 & \quad \Sigma; \sigma \vdash e_3 \Rightarrow v_3; \sigma_3 \\
\Sigma; \sigma \vdash \text{if } (e_1) \text{ then } e_2 \text{ else } e_3 \Rightarrow v_3; \sigma_3 \\
\Sigma; \sigma \vdash e_1 \Rightarrow \$x; \sigma \\
\Sigma; \sigma \vdash \$x \Rightarrow v; \sigma \\
\Sigma; \sigma \vdash \ell \Rightarrow \ell; \sigma \\
\Sigma; \sigma \vdash () \Rightarrow (); \sigma \\
\Sigma; \sigma \vdash e_1, e_2 \Rightarrow (v_1, v_2); \sigma_2 \\
\forall i \in 1..n: \Sigma; \sigma_i \vdash e_i \Rightarrow xv_i; \sigma_{i+1} \quad \Sigma(f)(xv_1, \ldots, xv_n) = xv \\
\Sigma; \sigma_1 \vdash f(e_1, \ldots, e_n) \Rightarrow xv; \sigma_{n+1}
\end{align*}
\]
Core+tuples semantics (2 of 2)

\[
\Sigma; \sigma \vdash e \Rightarrow xv; \sigma' \quad (\sigma'', id) = \text{newelem}(\sigma', q, xv)
\]
\[
\Sigma; \sigma \vdash \text{element } q\{e\} \Rightarrow id; \sigma''
\]
\[
\Sigma; \sigma \vdash $x/s \Rightarrow (xv', xv_2); \sigma_2
\]
\[
\forall i \in 1..n: \Sigma; \sigma_i \vdash e_i \Rightarrow xv_i; \sigma_{i+1}
\]
\[
\Sigma; \sigma_1 \vdash [a_1 : e_1; \ldots; a_n : e_n] \Rightarrow [a_1 : xv_1; \ldots; a_n : xv_n]; \sigma_{n+1}
\]
\[
\Sigma; \sigma \vdash e_1 \Rightarrow [a_{11} : xv_{11}; \ldots; a_{1n} : xv_{1n}]; \sigma_1
\]
\[
\Sigma; \sigma_1 \vdash e_2 \Rightarrow [a_{21} : xv_{21}; \ldots; a_{2m} : xv_{2m}]; \sigma_2
\]
\[
\forall i \in 1..n, j \in 1..m: a_{1j} = a_{2j} \Rightarrow xv_{1i} = xv_{2j}
\]
\[
\Sigma; \sigma \vdash e_1 \oplus e_2 \Rightarrow [a_{11} : xv_{11}; \ldots; a_{1n} : xv_{1n}; a_{21} : xv_{21}; \ldots; a_{2m} : xv_{2m}]; \sigma_2
\]
XQuery type rules

\[
\begin{align*}
\Gamma \vdash E : x_t & \quad \text{xt} \leq x_t \\
\Gamma \vdash E : \{x_t\} & \\
\Gamma(\$x) = x_t & \\
\Gamma \vdash \$x : x_t & \\
\Gamma \vdash E : \{\text{Item}\} & \\
\Gamma \vdash E_1 : \{\text{Item}\} & \quad \Gamma \vdash E_2 : x_t & \quad \Gamma \vdash E_3 : x_t \\
\Gamma \vdash \text{if}(E_1) \text{ then } E_2 \text{ else } E_3 : \{x_t\} & \\
\Gamma \vdash E_1 : \{\text{Item}\} & \quad \Gamma \vdash E_2 : \{\text{Item}\} & \quad \Gamma \vdash E_3 : x_t \\
\Gamma \vdash f(E_1, \ldots, E_n) : x_t & \\
\Gamma \vdash E : \{\text{Item}\} & \quad \Gamma \vdash E_1 : \{\text{Item}\} & \quad \Gamma \vdash E_2 : x_t \\
\Gamma \vdash E \text{ / } s : \{\text{Item}\} & \\
\Gamma \vdash \text{element} q\{E\} : \text{Item} & \\
\Gamma \vdash \ell : \text{Item} & \\
\Gamma \vdash () : \{\text{Item}\} & \\
\end{align*}
\]
Normalization of XQuery to core+tuples (1 of 2)

\[
X[F]_t^o = X^*[F]_t^o \\
X[()]_t^o = () \\
X[E_1, E_2]_t^o = X[E_1]_t^o \cdot X[E_2]_t^o \\
X[$x$]_t^o = $t$.a \quad \text{where} \ a = \varrho($x$) \\
X[if (E_1) \text{ then } E_2 \text{ else } E_3]_t^o = \text{if } (X[E_1]_t^o) \text{ then } X[E_2]_t^o \text{ else } X[E_3]_t^o \\
X[f(E_1, \ldots, E_n)]_t^o = f(X[E_1]_t^o, \ldots, X[E_n]_t^o) \\
X[element q \{E\}]_t^o = \text{element } q \{X[E]_t^o\} \\
X[E/s]_t^o = \text{for } $dot$ in (X[E]_t^o) \text{ return } $dot/s \\
X[\ell]_t^o = \ell
\]
Normalization of XQuery to core+tuples (FLWOR, 2 of 2)

\[ \mathcal{X}^* \left[ \text{for } \$x \text{ in } E \ F \right]^\varrho_e = \mathcal{X}^* \left[ \mathcal{F} \right]^\varrho_1_{e_1} \quad \text{where} \]
\[ \varrho_1 = (\varrho, \$x \rightarrow a), \text{ a fresh} \]
\[ e_1 = \text{for } \$t_1 \text{ in } e \text{ return} \]
\[ \quad \text{for } \$t_2 \text{ in } (\mathcal{X}[E]^\varrho_{\$t_1}) \text{ return } \$t_1 \triangleright \mathbb{[} a : \$t_2 \mathbb{]} \]

\[ \mathcal{X}^* \left[ \text{order by } E \ F \right]^\varrho_e = \mathcal{X}^* \left[ \mathcal{F} \right]^\varrho_e \quad \text{where} \]
\[ e_1 = \text{order } \$t_1 \text{ in } e \text{ by } (\mathcal{X}[E]^\varrho_{\$t_1}) \]

\[ \mathcal{X}^* \left[ \text{let } \$x := E \ F \right]^\varrho_e = \mathcal{X}^* \left[ \mathcal{F} \right]^\varrho_1_{e_1} \quad \text{where} \]
\[ \varrho_1 = (\varrho, \$x \rightarrow a), \text{ a fresh} \]
\[ e_1 = \text{for } \$t_1 \text{ in } e \text{ return } \$t_1 \triangleright \mathbb{[} a : \mathcal{X}[E]^\varrho_{\$t_1} \mathbb{]} \]

\[ \mathcal{X}^* \left[ \text{where } E \ F \right]^\varrho_e = \mathcal{X}^* \left[ \mathcal{F} \right]^\varrho_e \quad \text{where} \]
\[ e_1 = \text{for } \$t_1 \text{ in } e \text{ return if } (\mathcal{X}[E]^\varrho_{\$t_1}) \text{ then } \$t_1 \text{ else } () \]

\[ \mathcal{X}^* \left[ \text{return } E \right]^\varrho_e = \text{for } \$t_1 \text{ in } e \text{ return } (\mathcal{X}[E]^\varrho_{\$t_1}) \]
Algebra type rules

1. ID: \( t \rightarrow t \)
2. \( \ell : t \rightarrow \text{Item} \)
3. \( [a : p] : t \rightarrow [a : xt] \)
4. \( \#a : [a : xt] \rightarrow xt \)

- **Select**:
  \[ p_1 : [r] \rightarrow \{ \text{Item} \} \]
  \[ p_2 : t \rightarrow \{ [r] \} \]

- **OrderBy**:
  \[ p_1 : [r] \rightarrow \{ \text{Item} \} \]
  \[ p_2 : t \rightarrow \{ [r] \} \]

- **MapConcat**:
  \[ p_1 : [r_1] \rightarrow \{ \text{Item} \} \]
  \[ p_2 : t \rightarrow \{ r_1 : r_2 \} \]

- **Empty**:
  \[ p_1 : t \rightarrow \{ \text{Item} \} \]
  \[ p_2 : t \rightarrow \{ \text{Item} \} \]
  \[ p : t \rightarrow \{ \text{Item} \} \]

- **Sequence**:
  \[ \text{Sequence}(p_1, p_2) : t \rightarrow \{ \text{Item} \} \]

- **TreeJoin**:
  \[ \Gamma(f) = (t_1', \ldots, t_n') \rightarrow t \]
  \[ p_1 : t_1 \rightarrow t_1' \]
  \[ \ldots \]
  \[ p_n : t_n \rightarrow t_n' \]

- **Conditional**:
  \[ \text{Conditional}(p_1, p_2, p_3) : t \rightarrow u \]

- **Call**:
  \[ \text{Call}(f)(p_1, \ldots, p_n) : (t_1, \ldots, t_n) \rightarrow t' \]

- **Map**:
  \[ p : t \rightarrow u \]
  \[ p : u \leq u^+ \]

- **Back to algebra**

Kristoffer Rose @ NeilFest

Core Relational + Functional + Imperative
Compiling XQuery into relational algebra (1 of 2)

\[ C[F]^\circ = C^* [F]^\circ_{ID} \quad \text{where } C^* \text{ is defined on the following page} \]

\[ C[()]^\circ = \text{Empty()} \]

\[ C[E_1, E_2]^\circ = \text{Sequence}( C[E_1]^\circ, C[E_2]^\circ ) \]

\[ C[$x]^\circ = \#a \quad \text{with } a = \varrho($x) \]

\[ C[\text{if } (E_1) \text{ then } E_2 \text{ else } E_3]^\circ = \text{Conditional}( C[E_1]^\circ, C[E_2]^\circ, C[E_3]^\circ ) \]

\[ C[f(E_1, \ldots, E_n)]^\circ = \text{Call}[f]( C[E_1]^\circ, \ldots, C[E_n]^\circ ) \]

\[ C[\text{element } q \{E\}]^\circ = \text{Element}[q]( \{C[E]^\circ\} ) \]

\[ C[E/s]^\circ = \text{TreeJoin}[s]( C[E]^\circ ) \]

\[ C[\ell]^\circ_{st} = \text{Scalar}[\ell]() \]
Compiling XQuery into relational algebra (2 of 2)

\[ C^*[\text{for } x \text{ in } E \ F]_p^o = C^*[\text{F}]_{p_1}^o \text{ where } \]
\[ p_1 = \text{MapConcat}\{\text{Map}\{a : \text{ID}\}(C[E]^o)\}(p) \]

\[ C^*[\text{order by } E \ F]_p^o = C^*[\text{F}]_{p_1}^o \text{ where } \]
\[ p_1 = \text{OrderBy}\{C[E]^o\}(p) \]

\[ C^*[\text{let } x := E \ F]_p^o = C^*[\text{F}]_{p_1}^o \text{ where } \]
\[ p_1 = \text{MapConcat}\{a : C[E]^o\}(p) \]

\[ C^*[\text{where } E \ F]_p^o = C^*[\text{F}]_{p_1}^o \text{ where } \]
\[ p_1 = \text{Select}\{C[E]^o\}(p) \]

\[ C^*[\text{return } E]_p^o = \text{Map}\{C[E]^o\}(p) \]

where \( C \) is defined on the previous page
Equivalence details

Define \( \tilde{X} \) as \( X \) except
\[
\tilde{X}[\text{for } x \text{ in } E F]_e = \tilde{X}[F]_{e_1}
\]
where
\[
\rho_1 = (\rho, x \mapsto a) \quad \text{with } a \text{ a fresh field name}
\]
\[
e_1 = \text{for } t_1 \text{ in } e \text{ return for } t_2 \text{ in } (\text{for } t_3 \text{ in } (\tilde{X}[E]_{x_1}) \text{ return } [a:t_3]) \text{ return } t_1 + t_2
\]
\[
\tilde{X}[\text{let } x := E F]_e = \tilde{X}[F]_{e_1}
\]
where
\[
\rho_1 = (\rho, x \mapsto a) \quad \text{with } a \text{ a fresh field name}
\]
\[
e_1 = \text{for } t_1 \text{ in } e \text{ return for } t_2 \text{ in } [a:\tilde{X}[E]_{x_1}] \text{ return } t_1 + t_2
\]

Lemma

For all \( \Sigma, \sigma, E, t, v', \sigma' \) we have that

1. \( \Sigma, \sigma \vdash \tilde{X}[E]_t \Rightarrow v', \sigma' \iff \Sigma, \sigma \vdash X[E]_t \Rightarrow v', \sigma' \)
2. \( \forall E : \tilde{X}[E] = A[C[E]] \)
Extended algebra type rules

\[
p_1 : t \rightarrow [r_1] \quad p_2 : t \rightarrow [r_2] \\
\]

\[
p_1 \oplus p_2 : t \rightarrow [r_1; r_2] \\
\]

\[
p : t \rightarrow \{[r]\} \\
\]

\[
\text{OMap}[n](p_1) : t \rightarrow \{[r; n : \{\text{Item}\}]\} \\
\]

\[
p_1 : t \rightarrow \{[r]\} \quad p_2 : t' \rightarrow t'' \\
\]

\[
p_1 \circ p_2 : t \rightarrow t'' \\
\]

\[
p_1 : t' \rightarrow t' \quad p_2 : t' \rightarrow t'' \\
\]

\[
p_3 : t \rightarrow \{[r; i : t; n : \{\text{Item}\}]\} \\
\]

\[
\text{GroupBy}[a, i, n]\{p_1\}\{p_2\}(p_3) : \\
\quad t \rightarrow \{[a : V; i : t]\} \\
\]

\[
p_1 : t \rightarrow \{[r_1]\} \quad p_2 : t \rightarrow \{[r_2]\} \\
\]

\[
\text{Product}(p_1, p_2) : t \rightarrow \{[r_2; r_1]\} \\
\]

\[
\text{OMapConcat}[n]\{p_1\}(p_2) : t \rightarrow \{[r_1; r_2; n : \{\text{Item}\}]\} \\
\]

\[
p_2 : t \rightarrow \{[r_2]\} \quad p_1 : [r_2] \rightarrow \{[r_1]\} \\
\]

\[
\text{Join}\{p_1\}(p_2, p_3) : t \rightarrow \{[r_2; r_3]\} \\
\]

\[
p_2 : t \rightarrow \{[r_2]\} \quad p_3 : t \rightarrow \{[r_3]\} \quad p_1 : [r_2; r_3] \rightarrow \{\text{Item}\} \\
\]

\[
\text{LOuterJoin}[n]\{p_1\}(p_2, p_3) : t \rightarrow \{[r_2; r_3; n : \{\text{Item}\}]\} \\
\]
References

Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong.
Comprehension syntax.

Leonidas Fegaras.
Query unnesting in object-oriented databases.

Goetz Graefe.
Query evaluation techniques for large databases.

Guido Moerkotte.
Building query compilers, draft manuscript.

Val Tannen, Peter Buneman, and Limsoon Wong.
Naturally embedded query languages.

Limsoon Wong.
Kleisli, a functional query system.
The End