XQuery on SQL Hosts

Torsten Grust

U Konstanz, Database Research Group

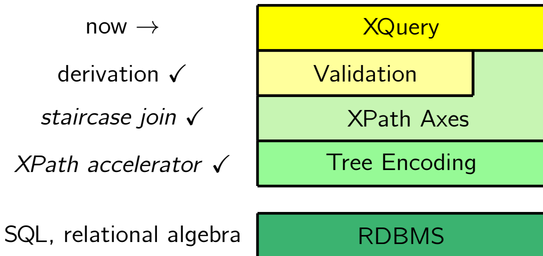
Torsten.Grust@uni-konstanz.de

S. Margherita di Pula—Sardinia, September 2004

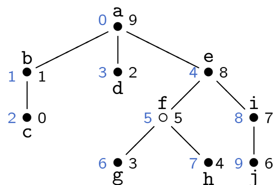
7th EDBT Summer School on "XML and Databases"

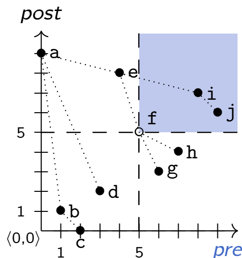
A Purely Relational XQuery Processing Stack

- A **fully relational** XQuery processor, developed bottom-up:



Node-based Relational Encodings of XQuery's Data Model





pre	post
0	9
1	1
2	0
3	2
4	8
5	5
6	3
7	4
8	7
9	6

- Staircase join (⌓) evaluates XPath axes on *pre*/*post* plane
- **Tuple $\widehat{=}$ node** (¬ tree-based)
- Any encoding reflecting **node identity**/**document order** suffices

Source Language: XQuery Core

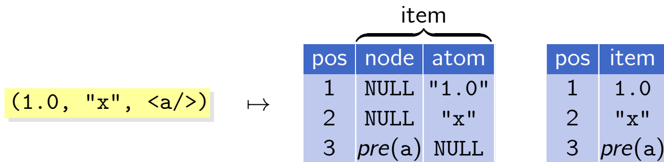
XQuery Core

- literals
- sequences (e_1, e_2)
- variables $(\$v)$
- `let ··· return`
- `for ··· where ··· return`
- `for ··· [at $v] ··· where ··· return`
- `if ··· then ··· else`
- `typeswitch ··· case ··· default`
- `element {···} {···}`
- `text {···}`
- XPath (e/α)
- function application
- document order $(e_1 << e_2)$
- node identity $(e_1 \texttt{ is } e_2)$
- arithmetics (`+`,`−`,`*`,`idiv`)
- `fn:doc()`
- `fn:root()`
- `fn:data()`
- `fn:distinct-doc-order()`
- `fn:count()`
- `fn:sum()`
- `fn:empty()`
- `fn:position()`
- `fn:last()`

- Expression may nest as defined by W3C XQuery Working Draft

Here: Focus on non-XPath-related XQuery Fragment

- Item sequences, **sequence order**



`(1.0, "x", <a/>)` ↦

pos	node (item)	atom (item)
1	NULL	"1.0"
2	NULL	"x"
3	*pre*(a)	NULL

pos	item
1	1.0
2	"x"
3	*pre*(a)

- Compiling **nested** `for···let···where···return` **blocks**
  - ▷ **Variable representation** and **scopes**
- Node construction
- XPath evaluation over persistent and transient nodes

Target Language: Flat Relational Algebra

Relational Algebra

π	column projection, renaming
σ	row selection
$\dot\cup$, $\setminus$	disjoint union, difference
δ	duplicate elimination
$\bowtie$	equi-join
$\times$	Cartesian product
ρ	row numbering
⫽	staircase join
ε, τ	element/text node construction
$\circledast$	arithmetic/comparison/Boolean operator $*$

- **No tree pattern matching** or similar operators involved here
- This algebra is efficiently implementable on (top of) SQL hosts

Iteration in XQuery Core

- XQuery Core has been designed around the `for` **iteration** primitive:

XQuery iteration

$$\texttt{for \$}v\texttt{ in (}x_1\texttt{,}x_2\texttt{, ... ,}x_n\texttt{) return } e$$
$$\equiv$$
$$(e[x_1/\$v], e[x_2/\$v], \ldots, e[x_n/\$v])$$

- Representation of $(x_1, x_2, \ldots, x_n)$:

pos	item
1	x_1
2	x_2
⋮	⋮
n	x_n

- Derive $\$v$ as follows:

iter	pos	item
1	1	x_1
2	1	x_2
⋮	⋮	⋮
n	1	x_n

Loop Lifting

- Subexpressions are compiled in dependence of **iteration scope** s in which they appear—represented as unary relation $loop(s)$

XQuery iteration

s_0
```
for $v in (x1,x2,...,xn)
    return e
```
(s_1 is the scope of `return e`; s_0 is the scope of the whole `for` expression)

$loop(s_0)$

iter
1

$loop(s_1)$

iter
1
⋮
n

▷ Single item "a" in scope s_1:

$loop(s_1)$

iter	pos	item
1	1	"a"
⋮	⋮	⋮
n	1	"a"

▷ Sequence ("a","b") in scope s_1:

$loop(s_1)$

iter	pos	item
1	1	"a"
1	2	"b"
⋮	⋮	⋮
n	1	"a"
n	2	"b"

Nested Scopes

Nested `for` blocks

s_0 [`for` $\$v_0$ `in (10,20)`
  s_1 [`for` $\$v_1$ `in (100,200)`
    s_2 [`return` $\$v_0$ `+` $\$v_1$

$loop(s_0)$

iter
1

$loop(s_1)$

iter
1
2

$loop(s_2)$

iter
1
2
3
4

- Derive $\$v_0, \$v_1$ as before (uses row numbering operator ρ):

$\$v_0$ in s_1:

iter	pos	item
1	1	10
2	1	20

$\$v_1$ in s_2:

iter	pos	item
1	1	100
2	1	200
3	1	100
4	1	200

▷ Variable $\$v_0$ in scope s_2?

Relating Scopes

Nested `for` blocks

```
s0 [ for $v0 in (10,20)
s1   [ for $v1 in (100,200)
s2     [ return $v0 + $v1
```

- Relation *map* captures the semantics of nested iteration:

map:

inner	outer
1	1
2	1
3	2
4	2

▷ Representation of $\$v_0$ in s_2:

$$\pi_{iter:inner,pos,item}(\$v_0 \bowtie_{iter=outer} map) =$$

iter	pos	item
1	1	10
2	1	20
3	1	10
4	1	20

Evaluation in scope s_2

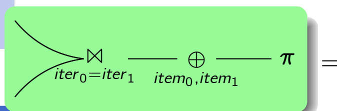
```
for $v0 in (10,20)
   for $v1 in (100,200)
      s2 [ return $v0 + $v1
```

$\$v_0$

$iter_0$	pos_0	$item_0$
1	1	10
2	1	20
3	1	10
4	1	20

$\$v_1$

$iter_1$	pos_1	$item_1$
1	1	100
2	1	200
3	1	100
4	1	200



=

iter	pos	item
1	1	110
2	1	210
3	1	120
4	1	220

Empty Sequences

- Encode `()` by **absence** of *iter* value in loop-lifted sequence *e*:

loop:

iter
1
2
3

e:

iter	pos	item
1	1	`"a"`
3	1	`"x"`
3	2	`"y"`

Compile `fn:empty(`*e*`)`

=

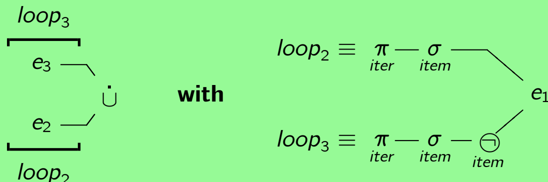
iter	pos	item
1	1	0
2	1	1
3	1	0

Example: Compiling Conditional Expressions

XQuery conditional expression

`if (`e_1`) then `e_2` else `e_3

Equivalent algebraic code



- $\ominus_c$ denotes the algebra's Boolean negation operator (column c)
- σ_c selects all tuples with column $c \neq 0$

Inference Rules

- The compiler is specified in terms of **inference rules**, collectively defining the $\Mapsto$ (*compiles to*) function
- Here is the (somewhat simplified) inference rule for the compilation of `if···then···else`:

Inference rule `if···then···else`

$$\frac{\begin{array}{c}\Gamma; loop \vdash e_1 \Mapsto q_1 \\ loop_2 \equiv \pi_{iter}(\sigma_{item}(q_1)) \qquad loop_3 \equiv \pi_{iter}(\sigma_{item}(\ominus_{item}(q_1))) \\ \Gamma; loop_2 \vdash e_2 \Mapsto q_2 \qquad \Gamma; loop_3 \vdash e_3 \Mapsto q_3\end{array}}{\Gamma; loop \vdash \texttt{if (}e_1\texttt{) then } e_2 \texttt{ else } e_3 \Mapsto q_2 \,\dot{\cup}\, q_3}$$

  - ▷ Γ denotes an environment mapping variables to their compiled equivalent

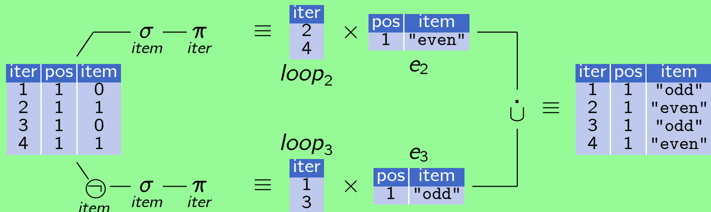
Example: Evaluation of a Conditional Expression

XQuery expression

```
for $x in 1 to 4
return if ($x mod 2 = 0) then "even" else "odd"
```

s_1: `return if ($x mod 2 = 0) then "even" else "odd"`, where e_1 = `$x mod 2 = 0`, e_2 = `"even"`, e_3 = `"odd"`

Evaluation

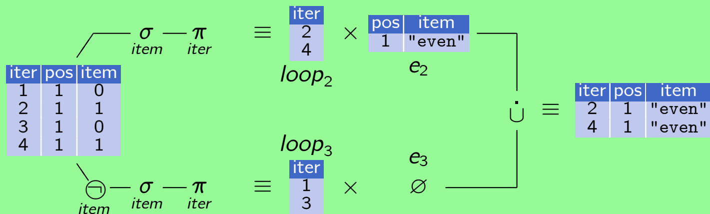


Compiling the `where` Clause

XQuery expression

```
⟦ for $x in 1 to 4          ⟧     ⟦ for $x in 1 to 4
⟦ where $x mod 2 = 0        ⟧  ≡  ⟦ return if $x mod 2 = 0
⟦ return "even"             ⟧     ⟦        then "even" else ()
```

Evaluation



XQuery Core Optimization

- Compiler generates code which expands the FLWOR **tuple space**

⇒ **loop-invariant code motion** becomes relevant

Q_1: Original Query

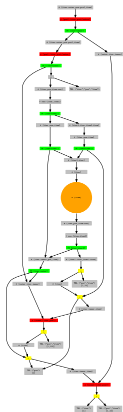
```
for $x in e1 , $y in e2
    where p($x) return e3($x, $y)
```

Q_2: Loop-invariant predicate moved

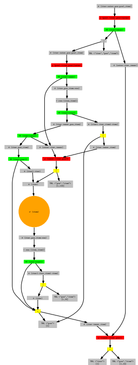
```
for $x in e1
    where p($x) return for $y in e2
                        return e3($x, $y)
```

Effect of Code Motion

Q_1:



Q_2:



Properties of Compiled Query Plans

- The target algebra as well as the compiled plans exhibit a number of nice properties:

Algebra/plan properties

π	**no need to eliminate duplicate tuples**
$\dot\cup$	all **unions are disjoint**
$\bowtie$	all joins are **equi-joins**
$\times$	one input is **singleton** (column attachment)
	plans are **DAGs** with significant sharing

- Simple, "*assembly style*" operators with simple semantics

XQuery on SQL Hosts (DB2 UDB V8.1)

- Plans translate into SQL query (nesting, but **no correlation**)
  - ▷ π translates into plain `SELECT` (no `DISTINCT`)
  - ▷ $\dot\cup$ translates into `UNION ALL`
  - ▷ ρ (row numbering) exactly mirrors SQL:1999 **OLAP** functionality:

pos	iter	item
1	1	2
2	1	6
3	1	7
1	2	3
2	2	5

≡

```
SELECT iter, item
       DENSE_RANK() OVER
       (PARTITION BY iter
        ORDER BY item) pos
  FROM (···)
```

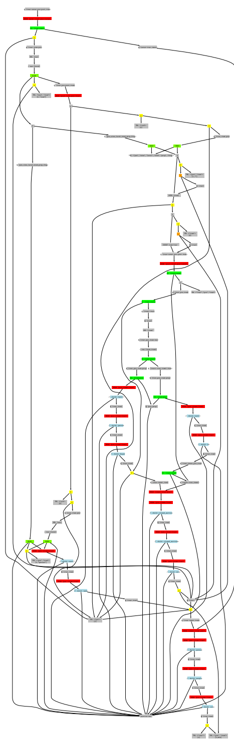
execution time [s]	1.1 MB	110 MB	1.1 GB
XMark Q1	< 0.01	< 0.01	< 0.01
XMark Q6	0.01	0.18	1.7
XMark Q7	0.01	0.52	5.3

XQuery on SQL Hosts

XMark Query Q8

```
for $p in fn:doc("auction.xml")/site/people/person
return let $a := for $t in fn:doc("auction.xml")/site/
                                     closed_auctions/closed_auction
                 return if fn:data($t/buyer/person/text()) =
                           fn:data($p/id/text())
                        then $t else ()
       return <item> { <person> { $p/name/text() } </person>,
                        text { fn:count($a) } } </item>
```

- Compiles into DAG of 120 algebraic operators, significant sharing



XQuery on SQL Hosts

XMark Query Q8

```
for $p in fn:doc("auction.xml")/site/people/person
return let $a := for $t in fn:doc("auction.xml")/site/
                                 closed_auctions/closed_auction
                 return if fn:data($t/buyer/person/text()) =
                           fn:data($p/id/text())
                        then $t else ()
       return <item> { <person> { $p/name/text() } </person>,
                      text { fn:count($a) } } </item>
```

- Compiles into DAG of 120 algebraic operators, significant sharing
  - ▷ Equivalent tree has ≈ 2000 nodes
- **NB:** No optimizations applied yet (neither XQuery nor algebraic)

Work in Flux

Optimizations

- Exploit **column properties**: *unique*, *constant*, *dense*
- **Order awareness** [ICDE 2004, SIGMOD 1996]
- Exploit **disjointness** of intermediate results
- "**Live node analysis**": evaluate ⌰ over minimal tree fragments

New implementation

- Main-memory DBMS kernel **MonetDB** (CWI, Amsterdam)
- Target language is MIL, algebra over binary tables
- Ordered data model (ρ may largely become obsolete)

Schema Validation and Type Annotation for Encoded Trees

Torsten Grust

U Konstanz, Database Research Group

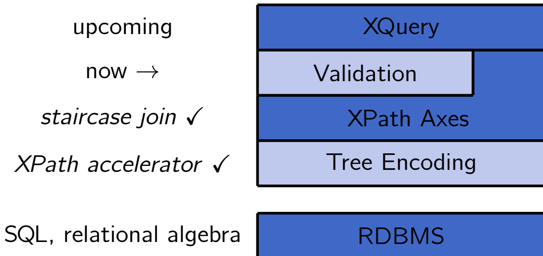
Torsten.Grust@uni-konstanz.de

S. Margherita di Pula—Sardinia, September 2004

7th EDBT Summer School on “XML and Databases”

A Purely Relational XQuery Processing Stack

- Need for efficient **XPath evaluation** drove development of stack:



Inspecting Type Annotations in XQuery

- XQuery provides several means to **inspect the type** of a specific node (or item, in general): **sequence type matching**

Sequence types

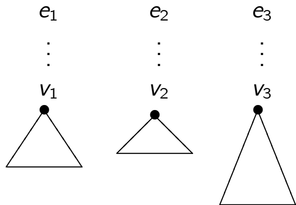
```
for $a in fn:doc("tagency.xml")//element(*,airline)
where some $f in $a/flights satisfies
      ($f/from = "Rome" and $f/to = "Cagliari")
return $a/name
```

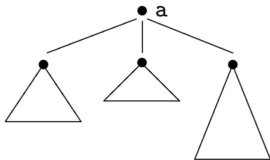
- Also via. . .
  - ▷ `typeswitch···case···default`
  - ▷ `instance of`

Validation in XQuery

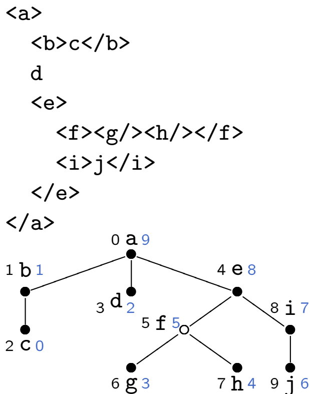
- **Explicitly:** `validate` e

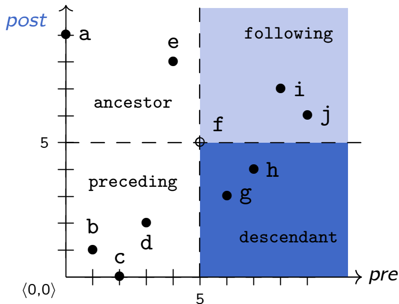
  Implicitly: `element a {` e_1`,` e_2`,` e_3 `}`



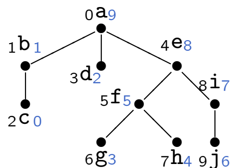


An XPath-aware Tree Encoding





Encoding Validated Trees



pre	*post*
0	9
1	1
2	0
3	2
4	8
5	5
6	3
7	4
8	7
9	6

pre	*kind*
0	*elem*
1	*elem*
2	*text*
3	*text*
4	*elem*
5	*elem*
6	*elem*
7	*elem*
8	*elem*
9	*text*

pre	*name*
0	a
1	b
2	□
3	□
4	e
5	f
6	g
7	h
8	i
9	□

pre	*type*
0	t1
1	xs:string
2	□
3	□
4	t2
5	t3
6	xs:string
7	xs:string
8	xs:string
9	□

Efficient Validation of Encoded Trees

- Scan tables forward (in sync), populate *pre type* table:

	pre	*post*		*pre*	*kind*		*pre*	*name*		*pre*	*type*
↓	0	9	↓	0	*elem*	↓	0	`a`	↓	0	`t1`
↓	1	1	↓	1	*elem*	↓	1	`b`	↓	1	`xs:string`
↓	2	0	↓	2	*text*	↓	2	□	↓	2	□
↓	3	2	↓	3	*text*	↓	3	□	↓	3	□
↓	4	8	↓	4	*elem*	↓	4	`e`	↓	4	`t2`
↓	5	5	↓	5	*elem*	↓	5	`f`	↓	5	`t3`
↓	6	3	↓	6	*elem*	↓	6	`g`	↓	6	`xs:string`
↓	7	4	↓	7	*elem*	↓	7	`h`	↓	7	`xs:string`
↓	8	7	↓	8	*elem*	↓	8	`i`	↓	8	`xs:string`
↓	9	6	↓	9	*text*	↓	9	□	↓	9	□

- Encounter **sequence of encoded nodes** in document order

⇒ XML Schema type ≡ regular expression over node sequences

Regular Expressions over Node Sequences

$e ::= \varepsilon$	empty
$\mid\ \varnothing$	none
$\mid$ elem t	element node (tag t)
$\mid$ attr t	attribute node
$\mid$ text	text node
$\mid\ e \cdot e$	sequence
$\mid\ e \mid e$	choice
$\mid\ e^{m,n}$	repetition
$\mid\ e \mathbin{\&} \cdots \mathbin{\&} e$	interleave
$\mid$ elem $t\,\{e\}$	element + content
$\mid$ attr $t\,\{e\}$	attribute + content
$\mid\ id$	type name

Example: Representation of XML Schema (Type Defn's)

```
<xs:element name="a" type="t1"/>
<xs:complexType name="t1" mixed="true">
 <xs:sequence>
  <xs:element name="b" type="xs:string"/>
  <xs:element name="e" type="t2"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="t2">
 <xs:sequence>
  <xs:element name="f" type="t3" maxOccurs="unbounded"/>
  <xs:element name="i" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
```

type		*type definition*
txt	$\rightarrow$	$\varepsilon \mid \text{text}$
xs:string	$\rightarrow$	txt
t1	$\rightarrow$	$\mathtt{txt} \cdot \text{elem b}\,\{\mathtt{xs{:}string}\} \cdot \mathtt{txt} \cdot \text{elem e}\,\{\mathtt{t2}\} \cdot \mathtt{txt}$
t2	$\rightarrow$	$(\text{elem f}\,\{\mathtt{t3}\})^{1,*} \cdot \text{elem i}\,\{\mathtt{xs{:}string}\}$

Example: Representation of XML Schema (Declarations)

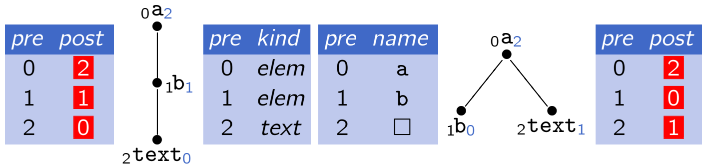
```
<xs:element name="a" type="t1"/>
<xs:complexType name="t1" mixed="true">
 <xs:sequence>
  <xs:element name="b" type="xs:string"/>
  <xs:element name="e" type="t2"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="t2">
 <xs:sequence>
  <xs:element name="f" type="t3" maxOccurs="unbounded"/>
  <xs:element name="i" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
```

validation context	*schema declaration*
	elem a {t1}
a	elem b {xs:string}
a	elem e {t2}
a/e	elem f {t3}
a/e	elem i {xs:string}

Flat Node Sequences vs. Element Content

- Table scan encounters v/`descendant::node()` right after v itself
- This suggests the equivalence

$$\text{elem } a \,\{\text{elem } b\, \{\text{text}\}\} \quad \equiv \quad \text{elem } a \cdot \text{elem } b \cdot \text{text}$$



pre	post
0	2
1	1
2	0

pre	kind
0	elem
1	elem
2	text

pre	name
0	a
1	b
2	□

pre	post
0	2
1	0
2	1

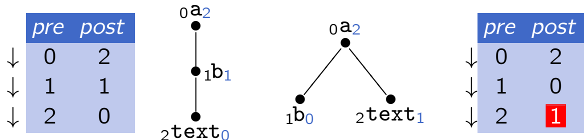
Pre/Post Guards

- If reg.exp. (elem t) has been matched with node v, then

$$\text{elem}\ t\,\{e\} \quad\equiv\quad \text{elem}\ t \cdot {}_{pre(v)}^{\infty}[\![\, e \,]\!]_{-\infty}^{post(v)}$$

- Tree t encoded as node sequence $v_0, v_1, \dots, v_n$:

$$\begin{aligned} {}_{a}^{b}[\![e]\!]_{c}^{d}\ \textbf{matches}\ t \quad :\Leftrightarrow \quad & e\ \textbf{matches}\ t \ \wedge \\ & \forall_{i=0}^{n}:\ a < pre(v_i) < b \ \wedge\ c < post(v_i) < d \end{aligned}$$



$$\text{elem b}\,\{\text{text}\} \quad\equiv\quad \text{elem b} \cdot {}_{1}^{\infty}[\![\text{text}]\!]_{-\infty}^{1}$$

Deriving Validity

- **Brzozowski's derivative** ∂ of a regular expression e:

> ∂ : node $\times$ reg.exp. $\rightarrow$ reg.exp.
>
> $\partial_v(e)$ is the residual regular expression remaining after node v has been matched against e.

▷ Tree t encoded as node sequence $v_0, v_1, v_2, \ldots, v_n$

$$\begin{array}{llll} e \textbf{ matches } t & \Leftrightarrow & \partial_{v_0}(e) & \textbf{matches } v_1, v_2, \ldots, v_n \\ & \Leftrightarrow & \partial_{v_1}(\partial_{v_0}(e)) & \textbf{matches } v_2, \ldots, v_n \\ & \vdots & & \\ & \Leftrightarrow & \partial_{v_n}(\cdots(\partial_{v_1}(\partial_{v_0}(e)))\cdots) & \textbf{matches } \varepsilon \quad \checkmark \end{array}$$

- No FSA construction involved

Derivation (1)

- Let v denote an encoded node. $\partial_v(e)$ is defined as follows:

Derivation

$$
\begin{aligned}
\partial_v(\varepsilon) &= \varnothing \\
\partial_v(\varnothing) &= \varnothing \\
\partial_v(\text{elem } t) &= \begin{cases} \varepsilon & \text{if } kind(v) = elem \wedge name(v) = t \\ \varnothing & \text{otherwise} \end{cases} \\
\partial_v(\text{attr } t) &= \begin{cases} \varepsilon & \text{if } kind(v) = attr \wedge name(v) = t \\ \varnothing & \text{otherwise} \end{cases} \\
\partial_v(\text{text}) &= \begin{cases} \varepsilon & \text{if } kind(v) = text \\ \varnothing & \text{otherwise} \end{cases} \\
\partial_v(id) &= \begin{cases} \partial_v(e) & \text{if } id \rightarrow e \\ \varnothing & \text{otherwise} \end{cases}
\end{aligned}
$$

Derivation (2)

Derivation

$$\partial_v(e_1 \cdot e_2) = \begin{cases} (\partial_v(e_1) \cdot {}_{pre(v)}[\![e_2]\!]_{post(v)}) \mid \partial_v(e_2) & \text{if } \nu(e) \\ \partial_v(e_1) \cdot {}_{pre(v)}[\![e_2]\!]_{post(v)} & \text{otherwise} \end{cases}$$

$$\partial_v(e_1 \mid e_2) = \partial_v(e_1) \mid \partial_v(e_2)$$

$$\partial_v(e^{m,n}) = \begin{cases} \partial_v(e) & \text{if } n = 1 \\ \partial_v(e) \cdot {}_{pre(v)}[\![e^{max(0,m-1),n-1}]\!]_{post(v)} & \text{if } n > 1 \end{cases}$$

$$\partial_v(\text{elem } t \, \{id\}) = \begin{cases} [type(v) := id] \ {}_{pre(v)}[\![e]\!]^{post(v)} & \text{if } name(v) = t \\ & \wedge \ id \rightarrow e \\ \varnothing & \text{otherwise} \end{cases}$$

$$\partial_v({}^{q}_{p}[\![e]\!]^{s}_{r}) = \begin{cases} {}^{q}_{p}[\![\partial_v(e)]\!]^{s}_{r} & \text{if } p < pre(v) < q \wedge r < post(v) < s \\ \varnothing & \text{otherwise} \end{cases}$$

A “Nullable” Test

- **Nullable test**:

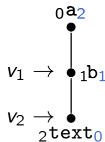
$$\nu(e) \quad \Leftrightarrow \quad \varepsilon \in L(e)$$

Nullable test

$$
\begin{aligned}
\nu(\varepsilon) &= \mathit{true} \\
\nu(\varnothing) &= \mathit{false} \\
\nu(\mathrm{elem}\ t) &= \mathit{false} \\
\nu(\mathrm{text}) &= \mathit{false} \\
\nu(e_1 \cdot e_2) &= \nu(e_1) \wedge \nu(e_2) \\
\nu(e_1 \mid e_2) &= \nu(e_1) \vee \nu(e_2) \\
\nu(e^{m,n}) &= m = 0 \vee \nu(e) \\
\nu(\mathrm{elem}\ t\, \{e\}) &= \mathit{false} \\
\nu({}^{q}_{p}[\![e]\!]^{s}_{r}) &= \nu(e) \\
\nu(\mathit{id}) &= \nu(e) \quad \text{with } \mathit{id} \rightarrow e
\end{aligned}
$$

Validation and Type Annotation

- Schema: type definition $t \to \text{text}$, element declaration elem b $\{t\}$
- Current state of table scan:



	pre	post
↓	0	2
↓	1	1
↓	2	0

	pre	kind
↓	0	*elem*
↓	1	*elem*
↓	2	*text*

	pre	name
↓	0	a
↓	1	b
↓	2	□

	pre	type
↓	⋮	⋮
↓	1	*t*
↓	2	□

① $\partial_{v_1}(\text{elem b}\,\{t\}) = {}^{\infty}_{1}[\![\, \text{text} \,]\!]^{1}_{-\infty}$

② $\partial_{v_2}({}^{\infty}_{1}[\![\, \text{text} \,]\!]^{1}_{-\infty}) = {}^{\infty}_{1}[\![\, \partial_{v_2}(\text{text}) \,]\!]^{1}_{-\infty}$

$= {}^{\infty}_{1}[\![\, \varepsilon \,]\!]^{1}_{-\infty}$ ✓

XML Schema: Unique Particle Attribution

- Reg. exps. derived from XML Schema types: *1-unambiguous*

$\Rightarrow$ Matching of nodes and type annotation is **unambiguous**

$\Rightarrow$ **Immediate** deterministic choice:

$$\partial_v(e_1 \mid e_2) \quad = \quad \underbrace{\partial_v(e_1)}_{= \varnothing} \mid \underbrace{\partial_v(e_2)}_{= \varnothing}$$

- $\partial_v(e)$ preserves **1**-unambiguity, if e is in **star normal form**
  - ▷ A. Brüggemann-Klein, *Regular Expressions into Finite Automata*, Theoretical Computer Science, 120(2), 1993

`xs:all` Groups

- In XML Schema, `xs:attribute` and `xs:all` give rise to &-groups:

`xs:all` and &-groups

```
<xs:all>
  <xs:element name="a" type="xs:string" minOccurs="0"/>
  <xs:element name="b" type="xs:string"/>
  <xs:element name="c" type="xs:string"/>
</xs:all>
```

$$\equiv$$

$$\text{elem a } \{\texttt{xs:string}\}^{0,1} \;\&\; \text{elem b } \{\texttt{xs:string}\} \;\&\; \text{elem c } \{\texttt{xs:string}\}$$

- Naive expansion (size $O(n!)$, highly non-deterministic):

$$t_1 \,\&\, t_2 \,\&\, t_3 \equiv t_1\ t_2\ t_3 \mid t_1\ t_3\ t_2 \mid t_2\ t_1\ t_3 \mid t_2\ t_3\ t_1 \mid t_3\ t_1\ t_2 \mid t_3\ t_2\ t_1$$

Expanding &-Groups Lazily

- Let $m_i \in \{0, 1\}$ as required by XML Schema:

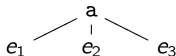
$$e_1{}^{m_1,1} \mathbin{\&} \cdots \mathbin{\&} e_n{}^{m_n,1} \equiv \Big|_{i=1}^{n} e_i \cdot \Bigg(\mathop{\&}_{\substack{j=1 \\ j \neq i}}^{n} e_j{}^{m_j,1} \Bigg)^{(\max_{\substack{j=1 \\ j \neq i}}^{n} m_j),1}$$

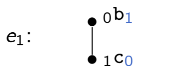
▷ All e_i different $\Rightarrow \partial$ yields **at most one** non-$\varnothing$ branch

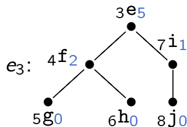
$$\begin{aligned} e_1 \mathbin{\&} e_2{}^{0,1} \mathbin{\&} e_3{}^{0,1} \;=\;& \; e_1 \cdot (e_2{}^{0,1} \mathbin{\&} e_3{}^{0,1})^{0,1} \\ & \mid e_2 \cdot (e_1 \mathbin{\&} e_3{}^{0,1}) \\ & \mid e_3 \cdot (e_1 \mathbin{\&} e_2{}^{0,1}) \\ =\;& \; e_1 \cdot (e_2 \cdot e_3{}^{0,1} \mid e_3 \cdot e_2{}^{0,1})^{0,1} \\ & \mid e_2 \cdot (e_1 \cdot e_3{}^{0,1} \mid e_3 \cdot e_1) \\ & \mid e_3 \cdot (e_1 \cdot e_2{}^{0,1} \mid e_2 \cdot e_1) \end{aligned}$$

Efficient Node Construction and Validation Mode `preserve`

`element a {` e_1 `,` e_2 `,` e_3 `}`

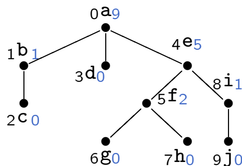






pre	size	type
	1	t_b
	0	t_c
	0	t_d
	5	t_e
	2	t_f
	0	t_g
	0	t_h
	1	t_i
	0	t_j

pre	size	type
0	9	t_a
1	1	t_b
2	0	t_c
3	0	t_d
4	5	t_e
5	2	t_f
6	0	t_g
7	0	t_h
8	1	t_i
9	0	t_j



High-Throughput Validation

- Early bail-out:

$$\partial_v(\varnothing) = \varnothing$$

- *pre*/*size* encoding: skip partially validated subtrees

- Validation against XMark schema: