XML and Databases

Lecture 9
Properties of XPath

Sebastian Maneth
NICTA and UNSW

CSE@UNSW -- Semester 1, 2009

Outline

1. XPath Equivalence
2. No Looking Back: How to Remove Backward Axes
3. Containment Test for XPath Expressions

A Note on Equality Test in XPath

Useful Functions (on Node Sets)

Careful with equality ("=")

XPath 2.0 has clearer comparison operators!

```
<a>
  <b>
    <d>red</d>
    <d>green</d>
    <d>blue</d>
  </b>
  <c>
    <d>yellow</d>
    <d>orange</d>
    <d>green</d>
  </c>
</a>
```

XPath 1.0
Equality ("=") is based on *string value* of a node!

`//a[b/d = c/d]` selects a-node!!!

there is a node in the node set for `b/d`
with same string value as a node in node set `c/d`

A Note on Equality Test

p1, p2 XPath (1.0) Expressions

(p1 == p2) is true if there exists a node selected by p1
that is *identical* to a node selected by p2

XPath 2.0
XQuery 1.0

```
<a>
 <b>
  <d>red</d>
  <d>green</d>
  <d>blue</d>
 </b>
 <c>
  <d>yellow</d>
  <d>orange</d>
  <d>green</d>
 </c>
</a>
```

`//a[b/d == c/d]` selects what?

A Note on Equality Test

p1, p2 XPath (1.0) Expressions

(p1 == p2) is true if there exists a node selected by p1
that is *identical* to a node selected by p2

XPath 2.0
XQuery 1.0

```
<a>
 <b>
  <d>red</d>
  <d>green</d>
  <d>blue</d>
 </b>
 <c>
  <d>yellow</d>
  <d>orange</d>
  <d>green</d>
 </c>
</a>
```

false (on *any* document)

`//a[b/d == c/d]` selects what?

```
//*[child::node()[1]
 == child::node()[position=last()]]
```

A Note on Equality Test

p1, p2 XPath (1.0) Expressions

(p1 == p2) is true if there exists a node selected by p1 that is *identical* to a node selected by p2

XPath 2.0
XQuery 1.0

XPath 1.0 simulation of (node) equality test (==)

Instead of (p1 == p2) write:

(count(p1 | p2) < count(p1) + count(p2)) ☺

1. XPath Equivalence

p1, p2 XPath (1.0) Expressions

(p1 ≡ p2) p1 "*is equivalent to*" p2
is true if,
*for any document **D**, and any context node **N** of **D**,*

p1 evaluated on ***D*** with context ***N*** gives the same result as
p2 evaluated on ***D*** with context ***N***.

Examples

```
/a//*/b         ≡  /a/*//b
//a/b/c/../..   ≡  //a[.b/c/]
//a[b | c]      ≡  //a/*[self::b | self::c]/..
```

NOT equivalent: `child::*/parent::*` ≢ `self::*`

→ show a counter example!

1. XPath Equivalence

EBNF for XPaths that we want to consider now:

```
         path ::= path | path |  / path | path / path | path [ qualif ]  | axis :: nodetest | ⊥ .
       qualif ::= qualif and qualif | qualif or qualif | ( qualif ) |
                  path = path | path == path | path .
         axis ::= reverse_axis | forward_axis .
 reverse_axis ::= parent | ancestor | ancestor-or-self |
                  preceding | preceding-sibling .
 forward_axis ::= self | child | descendant | descendant-or-self |
                  following | following-sibling .
     nodetest ::= tagname | * | text() | node() .
```

An XPath starting with "/" (root node) is called *absolute*,
otherwise it is called *relative*.

1. XPath Equivalence

p1, p2 XPaths
p arbitrary XPath
q arbitrary qualifier

Rel→Abs If p1 ≡ p2, then /p1 ≡ /p2.

Adjunct If p1 ≡ p2 and p is a relative, then p1/p ≡ p2/p.
If p1 ≡ p2 and p1,p2 relative, then p/p1 ≡ p/p2.
If p1 ≡ p2, then p1[q] ≡ p2[q] and p[p1] ≡ p[p2].

Qualifier Flattening p[p1/p2] ≡ p[p1[p2]]

ancestor-or-self::n ≡ ancestor::n | self::n
descendant-or-self::n ≡ descendant::n | self::n

p[p1 = /p2] ≡ p[p1[self::node() = /p2]]
p[p1 == /p2] ≡ p[p1[self::node() == /p2]]

1. XPath Equivalence

Lemma 3.2. *Let m and n be node tests, i.e. m and n are tag names or one of the xPath constructs* `*`, `node()`, *or* `text()`.

- *Let a be one of the axes* `parent`, `ancestor`, `preceding`, `preceding-sibling`, `self`, `following`, *or* `following-sibling`. *Then the following holds:*

$$/a::n \equiv \begin{cases} / & \text{if } a = \texttt{self} \text{ and } n = \texttt{node()} \\ \perp & \text{otherwise} \end{cases}$$

- *Let a be the* `preceding` *or* `ancestor` *axis. Then the following equivalences hold:*

$$\texttt{/child::}m/a\texttt{::}n \equiv \begin{cases} \texttt{/self::node()[child::}m\texttt{]} & \text{if } a = \texttt{ancestor} \text{ and } n = \texttt{node()} \\ \perp & \text{otherwise} \end{cases}$$

$$\texttt{/child::}m[a\texttt{::}n] \equiv \begin{cases} \texttt{/child::}m & \text{if } a = \texttt{ancestor} \text{ and } n = \texttt{node()} \\ \perp & \text{otherwise} \end{cases}$$

2. No Looking Back

Dual	backward	forward
	parent ancestor ancestor-or-self preceding preceding-sibling	child descendant descendant-or-self following following-sibling

Thus: dual(parent) = child
dual(following) = preceding
etc.

Rewrite rule #1 (p,s: relative paths, ax: reverse axis)

```
p[ax::m/s]   →
       p[/descendant::m[s]/dual(ax)::node() == self::node()]
```

Rewrite rule #1 (p,s: relative paths, ax: reverse axis)

```
p[ax::m/s]  ➔
        p[/descendant::m[s]/dual(ax)::node() == self::node()]
```

any "m[s]-node" in the tree — but, via dual axis, must reach context node

E.g. ax = ancestor

```
p[ancestor::m]  ➔
      p[/descendant::m/descendant::node()==self::node()]
```

"any m-node from which the context node can be reached via descendant, must be an ancestor of the context node."

Rewrite rule #1 (p,s: relative paths, ax: reverse axis)

```
p[ax::m/s]  ➔
        p[/descendant::m[s]/dual(ax)::node() == self::node()]
```

any "m[s]-node" in the tree — but, via dual axis, must reach context node

E.g. ax = preceding-sibling

```
p[preceding-sibling::m]  ➔
      p[/descendant::m/following-sibling::node()==self::node()]
```

"any m-node from which the context node can be reached via following-sibling, must be a preceding-sibling of the context node."

Rewrite rule #1 (p,s: relative paths, ax: reverse axis)

```
p[ax::m/s]  ➔
        p[/descendant::m[s]/dual(ax)::node() == self::node()]
```

any "m[s]-node" in the tree — but, via dual axis, must reach context node

E.g. ax=preceding-sibling

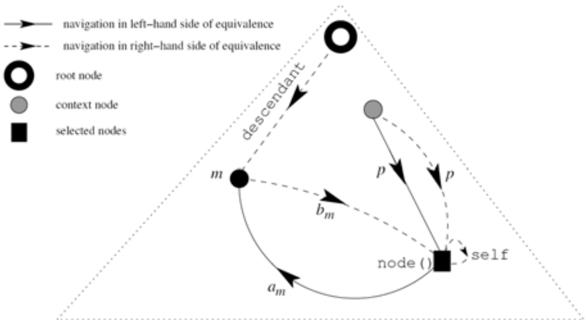
```
p[preceding-sibling::m]  ➔
      p[/descendant::m/following-sibling::node()==self::node()]
```

"any m-node from which the context node can be reached via following-sibling, must be a preceding-sibling of the context node."

Similar for parent and preceding. (ancestor-or-self not really needed. **Why?**)

Rewrite rule #1 (p,s: relative paths, ax: reverse axis)

```
p[ax::m/s]  ➔
        p[/descendant::m[s]/dual(ax)::node() == self::node()]
```



Rewrite rule #1 (p,s: relative paths, ax: reverse axis)

```
p[ax::m/s]  ➔
        p[/descendant::m[s]/dual(ax)::node() == self::node()]
```

Removes first reverse axis inside a filter (qualifier).

Use *qualifier flattening* to replace *any* reverse axis from inside a filter.

Qualifier Flattening p[p1/p2] ≡ p[p1[p2]]

Similar rules for **absolute paths**:

```
/p/fAx::n/ax::m  ➔ /descendant::m[dual(ax)::n == /p/fAx::n]
/fAx::n/ax::m    ➔ /descendant::m[dual(ax)::n == /fAx::n]
```

Rewrite rules #2 and #2a

E.g.

```
/descendant::price/preceding::name
```

is rewritten via Rule #2a into:

```
/descendant::name[following::price==/descendant::price]
```

Similar rules for **absolute paths**:

```
/p/fAx::n/ax::m  ➔ /descendant::m[dual(ax)::n == /p/fAx::n]
/fAx::n/ax::m    ➔ /descendant::m[dual(ax)::n == /fAx::n]
```

Rewrite rules #2 and #2a

E.g.

`/descendant::price/preceding::name`

is rewritten via Rule #2a into:

`/descendant::name[following::price==/descendant::price]`

Not needed, in this example.

Of course, the "join" can be removed in this example:

`/descendant::name[following::price]`

Similar rules for **absolute paths**:

```
/p/fAx::n/ax::m  → /descendant::m[dual(ax)::n == /p/fAx::n]
/fAx::n/ax::m    → /descendant::m[dual(ax)::n == /fAx::n]
```

Rewrite rules #2 and #2a

E.g.

`/descendant::journal[child::title]/descendant::price/preceding::name`

becomes

```
/descendant::name[following::price==
        /descendant::journal[child::title]/descendant::price]
```

Can you avoide the join, also for this example??

Similar rules for **absolute paths**:

```
/p/fAx::n/ax::m  → /descendant::m[dual(ax)::n == /p/fAx::n]
/fAx::n/ax::m    → /descendant::m[dual(ax)::n == /fAx::n]
```

Rewrite rules #2 and #2a


```
path ::= path | path | / path | path / path | path [ qualif ] | axis :: nodetest | ⊥ .
qualif ::= qualif and qualif | qualif or qualif | ( qualif ) |
           path = path | path == path | path .
axis ::= reverse_axis | forward_axis .
reverse_axis ::= parent | ancestor | ancestor-or-self |
                 preceding | preceding-sibling .
forward_axis ::= self | child | descendant | descendant-or-self |
                 following | following-sibling .
nodetest ::= tagname | * | text() | node() .
```

```
(1)   p[ax::m/s]     →
              p[/descendant::m[s]/dual(ax)::node() == self::node()]
(2)   /p/fAx::n/ax::m  → /descendant::m[dual(ax)::n == /p/fAx::n]
(2a)  /fAx::n/ax::m    → /descendant::m[dual(ax)::n == /fAx::n]
```

Rules (1),(2),(2a) suffice to remove ALL backward axes from above queries!
Why?
→ Size Increase?
→ How many joins?

2. No Looking Back

Dual	backward	forward	
	parent	child	
	ancestor	descendant	
	~~ancestor-or-self~~	~~descendant-or-self~~	not needed
	preceding	following	
	preceding-sibling	following-sibling	

Joins (==) are expensive! (typically quadratic wrt data)

To obtain queries with fewer joins
consider the **forward-axis** left of the reverse-axis to be removed!

New rules will be of the form

p/**forw**/**back** → p_new

p/**forw**[**back**] → p_new

2. No Looking Back

Interaction of `back=parent` with forward axes:

descendant::n/parent::m ≡ descendant-or-self::m[child::n] (3)

2. No Looking Back

Interaction of `back=parent` with forward axes:

descendant::n/parent::m ≡ descendant-or-self::m[child::n] (3)

child::n/parent::m ≡ self::m[child::n] (4)

2. No Looking Back

Interaction of back=parent with forward axes:

$$\texttt{descendant::}n\texttt{/parent::}m \equiv \texttt{descendant-or-self::}m\texttt{[child::}n\texttt{]} \tag{3}$$

$$\texttt{child::}n\texttt{/parent::}m \equiv \texttt{self::}m\texttt{[child::}n\texttt{]} \tag{4}$$

$$p\texttt{/self::}n\texttt{/parent::}m \equiv p\texttt{[self::}n\texttt{]/parent::}m \tag{5}$$

2. No Looking Back

Interaction of back=parent with forward axes:

$$\texttt{descendant::}n\texttt{/parent::}m \equiv \texttt{descendant-or-self::}m\texttt{[child::}n\texttt{]} \tag{3}$$

$$\texttt{child::}n\texttt{/parent::}m \equiv \texttt{self::}m\texttt{[child::}n\texttt{]} \tag{4}$$

$$p\texttt{/self::}n\texttt{/parent::}m \equiv p\texttt{[self::}n\texttt{]/parent::}m \tag{5}$$

$$p\texttt{/following-sibling::}n\texttt{/parent::}m \equiv p\texttt{[following-sibling::}n\texttt{]/parent::}m \tag{6}$$

2. No Looking Back

Interaction of back=parent with forward axes:

$$\texttt{descendant::}n\texttt{/parent::}m \equiv \texttt{descendant-or-self::}m\texttt{[child::}n\texttt{]} \tag{3}$$

$$\texttt{child::}n\texttt{/parent::}m \equiv \texttt{self::}m\texttt{[child::}n\texttt{]} \tag{4}$$

$$p\texttt{/self::}n\texttt{/parent::}m \equiv p\texttt{[self::}n\texttt{]/parent::}m \tag{5}$$

$$p\texttt{/following-sibling::}n\texttt{/parent::}m \equiv p\texttt{[following-sibling::}n\texttt{]/parent::}m \tag{6}$$

$$\begin{aligned} p\texttt{/following::}n\texttt{/parent::}m \equiv{} & p\texttt{/following::}m\texttt{[child::}n\texttt{]} \\ & \mid p\texttt{/ancestor-or-self::*[following-sibling::}n\texttt{]} \\ & \texttt{/parent::}m \end{aligned} \tag{7}$$

2. No Looking Back

Interaction of back=parent with forward axes:

$$\texttt{descendant::}n\texttt{/parent::}m \equiv \texttt{descendant-or-self::}m\texttt{[child::}n\texttt{]} \tag{3}$$

$$\texttt{child::}n\texttt{/parent::}m \equiv \texttt{self::}m\texttt{[child::}n\texttt{]} \tag{4}$$

$$p\texttt{/self::}n\texttt{/parent::}m \equiv p\texttt{[self::}n\texttt{]/parent::}m \tag{5}$$

$$p\texttt{/following-sibling::}n\texttt{/parent::}m \equiv p\texttt{[following-sibling::}n\texttt{]/parent::}m \tag{6}$$

$$\begin{aligned} p\texttt{/following::}n\texttt{/parent::}m \equiv{} & p\texttt{/following::}m\texttt{[child::}n\texttt{]} \\ & \mid p\texttt{/ancestor-or-self::*[following-sibling::}n\texttt{]} \\ & \texttt{/parent::}m \end{aligned} \tag{7}$$

$$\texttt{descendant::}n\ \texttt{[parent::}m\texttt{]} \equiv \texttt{descendant-or-self::}m\texttt{/child::}n \tag{8}$$

$$\texttt{child::}n\texttt{[parent::}m\texttt{]} \equiv \texttt{self::}m\texttt{/child::}n \tag{9}$$

$$p\texttt{/self::}n\texttt{[parent::}m\texttt{]} \equiv p\texttt{[parent::}m\texttt{]/self::}n \tag{10}$$

$$p\texttt{/following-sibling::}n\texttt{[parent::}m\texttt{]} \equiv p\texttt{[parent::}m\texttt{]/following-sibling::}n \tag{11}$$

$$\begin{aligned} p\texttt{/following::}n\texttt{[parent::}m\texttt{]} \equiv{} & p\texttt{/following::}m\texttt{/child::}n \\ & \mid p\texttt{/ancestor-or-self::*[parent::}m\texttt{]} \\ & \texttt{/following-sibling::}n \end{aligned} \tag{12}$$

2. No Looking Back

Interaction of back=ancestor with forward axes:

$$\begin{aligned} p\texttt{/descendant::}n\texttt{/ancestor::}m \equiv{} & p\texttt{[descendant::}n\texttt{]/ancestor::}m \\ & \mid p\texttt{/descendant-or-self::}m\texttt{[descendant::}n\texttt{]} \end{aligned} \tag{13}$$

2. No Looking Back

Interaction of back=ancestor with forward axes:

$$\begin{aligned} p\texttt{/descendant::}n\texttt{/ancestor::}m \equiv{} & p\texttt{[descendant::}n\texttt{]/ancestor::}m \\ & \mid p\texttt{/descendant-or-self::}m\texttt{[descendant::}n\texttt{]} \end{aligned} \tag{13}$$

$$\texttt{/descendant::}n\texttt{/ancestor::}m \equiv \texttt{/descendant-or-self::}m\texttt{[descendant::}n\texttt{]} \tag{13a}$$

2. No Looking Back

Interaction of `back=ancestor` with forward axes:

```
 p/descendant::n/ancestor::m ≡ p[descendant::n]/ancestor::m                (13)
                             | p/descendant-or-self::m[descendant::n]
  /descendant::n/ancestor::m ≡ /descendant-or-self::m[descendant::n]       (13a)
      p/child::n/ancestor::m ≡ p[child::n]/ancestor-or-self::m             (14)
```


2. No Looking Back

Interaction of `back=ancestor` with forward axes:

```
 p/descendant::n/ancestor::m ≡ p[descendant::n]/ancestor::m                (13)
                             | p/descendant-or-self::m[descendant::n]
  /descendant::n/ancestor::m ≡ /descendant-or-self::m[descendant::n]       (13a)
      p/child::n/ancestor::m ≡ p[child::n]/ancestor-or-self::m             (14)
       p/self::n/ancestor::m ≡ p[self::n]/ancestor::m                      (15)
```


2. No Looking Back

Interaction of `back=ancestor` with forward axes:

```
          p/descendant::n/ancestor::m ≡ p[descendant::n]/ancestor::m                (13)
                                      | p/descendant-or-self::m[descendant::n]
           /descendant::n/ancestor::m ≡ /descendant-or-self::m[descendant::n]       (13a)
               p/child::n/ancestor::m ≡ p[child::n]/ancestor-or-self::m             (14)
                p/self::n/ancestor::m ≡ p[self::n]/ancestor::m                      (15)
   p/following-sibling::n/ancestor::m ≡ p[following-sibling::n]/ancestor::m         (16)
```


2. No Looking Back

Interaction of `back=ancestor` with forward axes:

```
          p/descendant::n/ancestor::m ≡ p[descendant::n]/ancestor::m                (13)
                                      | p/descendant-or-self::m[descendant::n]
           /descendant::n/ancestor::m ≡ /descendant-or-self::m[descendant::n]       (13a)
               p/child::n/ancestor::m ≡ p[child::n]/ancestor-or-self::m             (14)
                p/self::n/ancestor::m ≡ p[self::n]/ancestor::m                      (15)
   p/following-sibling::n/ancestor::m ≡ p[following-sibling::n]/ancestor::m         (16)
           p/following::n/ancestor::m ≡ p/following::m[descendant::n]               (17)
                                      | p/ancestor-or-self::*
                                        [following-sibling::*/descendant-or-self::n]
                                        /ancestor::m
```

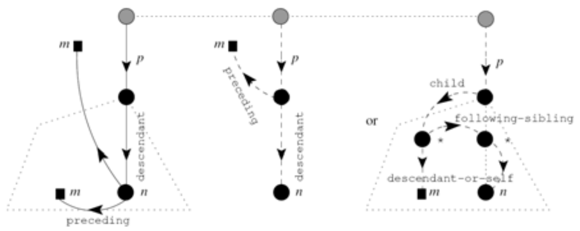
Similar rules for `ancestor` in a filters.

2. No Looking Back

Interaction of `back=preceding` with forward axes:

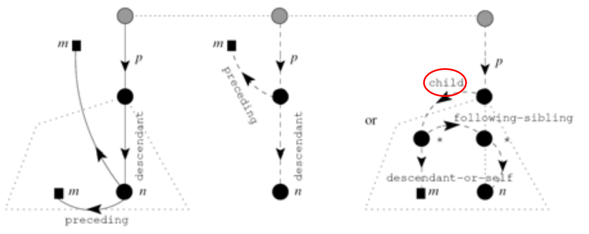
```
          p/descendant::n/preceding::m ≡ p[descendant::n]/preceding::m               (33)
                                       | p/child::*
                                         [following-sibling::*/descendant-or-self::n]
                                         /descendant-or-self::m
           /descendant::n/preceding::m ≡ /descendant::m[following::n]                (33a)
               p/child::n/preceding::m ≡ p[child::n]/preceding::m                    (34)
                                       | p/child::*[following-sibling::n]
                                         /descendant-or-self::m
                p/self::n/preceding::m ≡ p[self::n]/preceding::m                     (35)
   p/following-sibling::n/preceding::m ≡ p[following-sibling::n]/preceding::m       (36)
                                       | p/following-sibling::*[following-sibling::n]
                                         /descendant-or-self::m
                                       | p[following-sibling::n]/descendant-or-self::m
           p/following::n/preceding::m ≡ p[following::n]/preceding::m                (37)
                                       | p/following::m[following::n]
                                       | p[following::n]/descendant-or-self::m
```


Rule 33




```
p/descendant::n/preceding::m ≡ p[descendant::n]/preceding::m
                 | p/child::*[following-sibling::*/descendant-or-self::n]/descendant-or-self::m
```

Rule 33




```
p/descendant::n/preceding::m ≡ p[descendant::n]/preceding::m
      | p/child::*[following-sibling::*/descendant-or-self::n]/descendant-or-self::m
```

Wrong, I think!
Should be descendant instead!

2. No Looking Back

```
/descendant::price/preceding::name
```

is rewritten via Rule #2a into:

```
/descendant::name[following::price==/descendant::price]
```

Now, let us use Rule (33a)

```
/descendant::n/preceding::m  ➔  /descendant::m[following::n]
```

We obtain

```
/descendant::name[following::price]
```



```
/descendant::journal[child::title]/descendant::price/preceding::name
```

becomes

```
/descendant::name[following::price==
                   /descendant::journal[child::title]/descendant::price]
```

Rule (33a)

```
/descendant::n/preceding::m  ➔  /descendant::m[following::n]
```

doesn't work because descendant is absolute here.

Rule (33):

```
p/descendant::n/preceding::m  ➔  p[descendant::n]/preceding::m
                   | p/child::*[following-sibling::*/descendant-or-self::n]
                     /descendant-or-self::m
```

We obtain

```
p[descendant::price]/preceding::name
 | p/child::*[following-sibling::*/descendant-or-self::price]
              /descendant-or-self::name
```



```
/descendant::journal[child::title]/descendant::price/preceding::name
```

becomes

```
/descendant::name[following::price==
                   /descendant::journal[child::title]/descendant::price]
```

Rule (33a)

```
/descendant::n/preceding::m  ➔  /descendant::m[following::n]
```

doesn't work because descendant is absolute here.

Rule (33):

```
p/descendant::n/preceding::m  ➔  p[descendant::n]/preceding::m
                   | p/child::*[following-sibling::*/descendant-or-self::n]
                     /descendant-or-self::m
```

→ Rule (33a) with n = journal[child::title][descendant::price]

```
p[descendant::price]/preceding::name
 | p/child::*[following-sibling::*/descendant-or-self::price]
              /descendant-or-self::name
```



```
/descendant::journal[child::title]/descendant::price/preceding::name
```

becomes

```
/descendant::name[following::price==
                   /descendant::journal[child::title]/descendant::price]
```

Rule (33a)

```
/descendant::n/preceding::m  ➔  /descendant::m[following::n]
```

doesn't work because descendant is absolute here.

```
/descendant::name[following::journal[child::title][descendant::price]]
          | p/child::*[following-sibling::*/descendant-or-self::price]
                      /descendant-or-self::name
```

→ Rule (33a) with n = journal[child::title][descendant::price]

```
p[descendant::price]/preceding::name
 | p/child::*[following-sibling::*/descendant-or-self::price]
              /descendant-or-self::name
```



```
/descendant::journal[child::title]/descendant::price/preceding::name
```

=n

becomes

```
/descendant::name[following::price==
                   /descendant::journal[child::title]/descendant::price]
```

Rule (33a)

```
/descendant::n/preceding::m  ➔  /descendant::m[following::n]
```

~~doesn't work because descendant is absolute here.~~ seems it does work! ☺

```
/descendant::name[following::journal[child::title][descendant::price]]
          | p/child::*[following-sibling::*/descendant-or-self::price]
                      /descendant-or-self::name
```

What about this one:

```
/descendant::name[following::journal[child::title]/descendant::price]
```

Theorem
(from D. Olteanu, H. Meuss, T. Furche, F. Bry
XPath: Looking Forward. EDBT Workshops 2002: 109-127)

Given an XPath expression p that has no joins of the form (p1 == p2) with both p1,p2 relative, an equivalent expression u without reverse axes can be computed.

Time needed: at most **exponential** in length of p
Length of u: at most **exponential** in length of p

(moreover: *no joins* are introduced when computing u)

Questions

→ Can you find a subclass for which *Time* to compute u is linear or polynomial?

→ What is the problem with joins (p1 == p2) for removal of reverse axes?

43

3. XPath Containment Test

Given two XPath expressions p, q:
Are all nodes selected by p, also selected by q? (on *any* document)
(p "contained in" q)

Has many applications! Boolean query

Want to select documents that "match p".
→ If a document matches p, and p contained in q,
then we know the document also matches q!

→ If a document does not match q, and p contained in q,
then we know that document does not match p!

Applications

- → Decrease online-time of publish/subscribe systems based on XPath
- → Decrease query-time by making use of materialized intermediate results
- → Optimization by ruling out queries with empty result set

etc, etc

44

3. XPath Containment Test

Given two XPath expressions p, q

"0-containment" For every tree, if p selects a node then so does q.
$p \subseteq_0 q$

"1-containment" For every tree, all nodes selected by p are also selected by q.
$p \subseteq_1 q$

"2-containment" For every tree, and every context node N,
$p \subseteq_2 q$ all nodes selected by p starting from N,
are also selected by q starting from N.

1. Inclusion on *Booleans*
2. Inclusion on *Node Sets*
   } start from root
3. Inclusion on *Node Relations*

(If only child and descendant axes are allowed
then $\subseteq_1$ and $\subseteq_2$ are the same! -- **Why?**)

45

3. XPath Containment Test

Given two XPath expressions p, q

"0-containment" For every tree, if p selects a node then so does q.
$p \subseteq_0 q$

"1-containment" For every tree, all nodes selected by p are also selected by q.
$p \subseteq_1 q$

Question

Given p, q and the fact $p \subseteq_1 q$,
how can you determine from a *result set of nodes* for q,
the correct *result set of nodes* for p?

46

3. XPath Containment Test

Given two XPath expressions p, q

Sometimes we want to test containment wrt a given DTD:

p = /a/b//d
q = /a//c Boolean!

Want to check if $p \subseteq_0 q$.

NO!
```
a
|
b
|
d
```

But, what if documents are valid wrt to this DTD?

```
root →  a*
a    →  b* | c*
b    →  d+c+
c    →  b?c?
```

47

PTIME	XP(/, //, ∗) [21] XP(/, [ ], ∗) (see [19]) XP(/, //, [ ]) [2], with fixed bounded SXICs [9] XP(/, //) + DTDs [22] XP[/, [ ]] + DTDs [22]
coNP	XP(/, //, [ ], ∗) [19] XP(/, //, [ ], ∗, |), XP(/, |), XP(//, |) [22] XP(/, [ ]) + DTDs [22] XP(//, [ ]) + DTDs [22]
Π_2^p	XP(/, //, [ ], |) + existential variables + path equality + ancestor-or-self axis + fixed bounded SXICs [9] XP(/, //, [ ], ∗, |) + existential variables + all backward axes + fixed bounded SXICs [9] XP(/, //, [ ], |) + existential variables with inequality [22]
PSPACE	XP(/, //, [ ], ∗, |) and XP(/, //, |) if the alphabet is finite [22] XP(/, //, [ ], ∗, |) + variables with XPath semantics [22]
EXPTIME	XP(/, //, [ ], |) + existential variables + bounded SXICs [9] XP(/, //, [ ], ∗, |) + DTDs [22] XP(/, //, |) + DTDs [22] XP(/, //, [ ], ∗) + DTDs [22]
Undecidable	XP(/, //, [ ], |) + existential variables + unbounded SXICs [9] XP(/, //, [ ], |) + existential variables + bounded SXICs + DTDs [9] XP(/, //, [ ], ∗, |) + nodeset equality + simple DTDs [22] XP(/, //, [ ], ∗, |) + existential variables with inequality[22]

2. XPath Containment Test

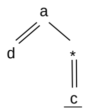
from:

T. Schwentick
XPath query containment.
SIGMOD Record 33(1): 101-109 (2004)

48

Pattern trees

E.g. `p = a[.//d]/*//c`



selection node (unique)

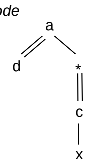
Note: child order has no meaning in pattern trees!

Test $\subseteq_1$ (node set inclusion) using $\subseteq_0$ (Boolean inclusion)

→ Simply add a new node below the *selection node*

New tree is Boolean (no selection node)

In a given XML tree:
pattern matches / does not match.



49

3. XPath Containment Test

4 techniques of testing XPath (Boolean) containment:

(1) The Canonical Model Technique

(2) The Homomorphism Technique

(3) The Automaton Technique

(4) The Chase Technique

50

3. XPath Containment Test

Canonical Model - XPath(`/`, `//`, `[]`, `*`)

Idea: if there exists a tree that matches p but not q, then such a tree exists of **size polynomial in the size** of p an q.

Simple: remember, if you know that the XML document is only of height 5, then //a/b/*/c could be enumerated by /a/b/*/c | /*/a/b/*/c | /*/*/a/b/*/c | /*/*/*/a…

Similarly, we try to construct a counter example tree, by replacing in p

→ every * by some new symbol "z"
→ every // by z/, z/z/, z/z/z/, … z/z/../z/

N+1 many z's

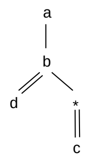
N = length of longest */../* chain in q

51

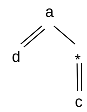
3. XPath Containment Test

Canonical Model - XPath(`/`, `//`, `[]`, `*`)

Example

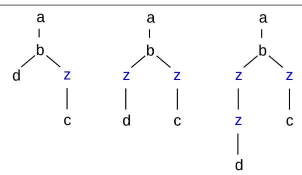


p's patter tree



q's patter tree

Test for q-match:



Formally, must test 1 and 2 more z's at *right branch* of each of the trees.

52

3. XPath Containment Test

Homomorphism h maps each node of q's query tree Q to a node of p's query tree P such that

(1) root of Q is mapped to root of P
(2) if (u,v) is child-edge of Q then (h(u),h(v)) is child-edge of P
(3) if (u,v) is descendant-edge of Q, then h(v) is a "below" h(u) in P
(4) if u is labeled by "e" (not *), then h(u) is also labeled by "e".

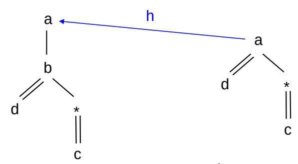
p,q expressions in XPath(`/`, `//`, `[]`)

Theorem
p $\subseteq_0$ q *if and only if* there is a homomorphism from Q to P.

53

3. XPath Containment Test

Homomorphism h maps each node of q's query tree Q to a node of p's query tree P such that



p's patter tree

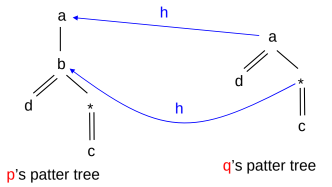
q's patter tree

(1) root of Q is mapped to root of P
(2) if (u,v) is child-edge of Q then (h(u),h(v)) is child-edge of P
(3) if (u,v) is descendant-edge of Q, then h(v) is a "below" h(u) in P
(4) if u is labeled by "e" (not *), then h(u) is also labeled by "e".

54

3. XPath Containment Test

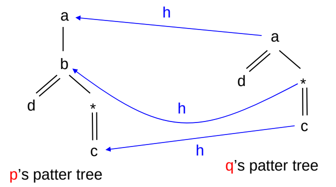
Homomorphism h maps each node of q's query tree Q
to a node of p's query tree P such that



(1) root of Q is mapped to root of P
(2) if (u,v) is child-edge of Q then (h(u),h(v)) is child-edge of P
(3) if (u,v) is descendant-edge of Q, then
h(v) is a "below" h(u) in P
(4) if u is labeled by "e" (not *), then h(u) is also labeled by "e".

3. XPath Containment Test

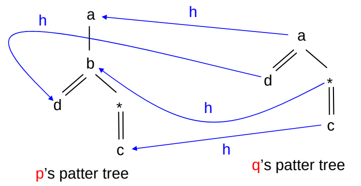
Homomorphism h maps each node of q's query tree Q
to a node of p's query tree P such that



(1) root of Q is mapped to root of P
(2) if (u,v) is child-edge of Q then (h(u),h(v)) is child-edge of P
(3) if (u,v) is descendant-edge of Q, then
h(v) is a "below" h(u) in P
(4) if u is labeled by "e" (not *), then h(u) is also labeled by "e".

3. XPath Containment Test

Homomorphism h maps each node of q's query tree Q
to a node of p's query tree P such that



→ hom. h exists from Q to P, thus $p \subseteq_0 q$ must hold!

(1) root of Q is mapped to root of P
(2) if (u,v) is child-edge of Q then (h(u),h(v)) is child-edge of P
(3) if (u,v) is descendant-edge of Q, then
h(v) is a "below" h(u) in P
(4) if u is labeled by "e" (not *), then h(u) is also labeled by "e".

3. XPath Containment Test

Homomorphism h maps each node of q's query tree Q
to a node of p's query tree P such that

(1) root of Q is mapped to root of P
(2) if (u,v) is child-edge of Q then (h(u),h(v)) is child-edge of P
(3) if (u,v) is descendant-edge of Q, then
h(v) is a "below" h(u) in P
(4) if u is labeled by "e" (not *), then h(u) is also labeled by "e".

p,q expressions in XPath(`/`,`//`,`[]`)

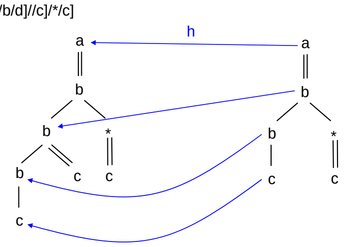
Theorem
$p \subseteq_0 q$ *if and only if* there is a homomorphism from Q to P.

Cave If we add the star (*) then homomorphism need not exist!

→ there are p,q ∈ XPath(`/`,`//`,`[]`,`*`) such that $p \subseteq_0 q$ and there is **no** homomorphism from Q to P ☹

3. XPath Containment Test

[/a//b[./b/d]//c]/*/c] [/a//b[./b/d]/*//c]

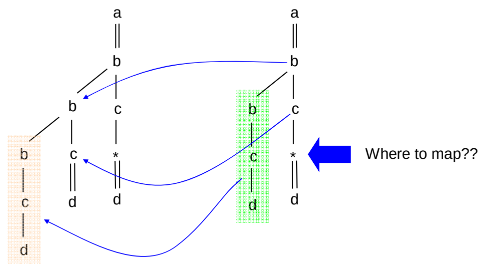


IS there a homomorphism??

Cave If we add the star (*) then homomorphism need not exist!

→ there are p,q ∈ XPath(`/`,`//`,`[]`,`*`) such that $p \subseteq_0 q$ and there is **no** homomorphism from Q to P ☹


```
p = /a[.//b[c/*//d]/b[c//d]/b[c/d]]
q = /a[.//b[c/*//d]/b[c/d]]
```

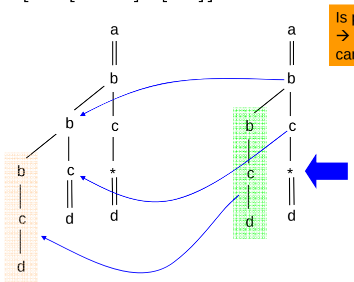


Cave If we add the star (*) then homomorphism need not exist!

→ there are p,q ∈ XPath(`/`,`//`,`[]`,`*`) such that $p \subseteq_0 q$ and there is **no** homomorphism from Q to P ☹

p = /a[.//b[c/*//d]/b[c//d]/b[c/d]]
q = /a[.//b[c/*//d]/b[c/d]]

Is p contained in q??
→ Test this, using the canonical model!!

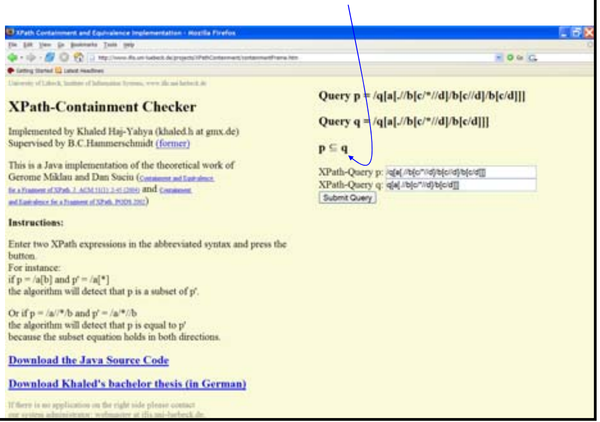


Where to map??

Cave If we add the star (*) then homomorphism need not exist!

→ there are p,q ∈ XPath(/,//,[],*) such that $p \subseteq_0 q$ and there is **no** homomorphism from Q to P ☹

Let's check the web… → **YES** p contained in q!



3. XPath Containment Test

Automaton Technique

Recall: for any DTD there is a tree automaton which recognized the corresponding trees.

Similarly, for any XPath(/,//,[],*,|) expression ex we can construct a (*non-deterministic* bottom-up) tree automaton A which accepts a tree if and only if ex matches the tree.

Theorem
Containment test of XPath(/,//,[],*,|) in the presence of DTDs can be solved in EXPTIME.

Exponential (deterministic) time
Blow-up due to non-determinism of tree automaton.

BUT: no hope for improvement:
The problem is actually ***complete*** for EXPTIME.

3. XPath Containment Test

Automaton Technique

Recall: for any DTD there is a tree automaton which recognized the corresponding trees.

Similarly, for any XPath(/,//,[],*,|) expression ex we can construct a (*non-deterministic* bottom-up) tree automaton A which accepts a tree if and only if ex matches the tree.

Theorem
Containment test of XPath(/,//,[],*,|) in the presence of DTDs can be solved in EXPTIME.

Union of automata

Intersection of automata ("product construction")

Proof Idea construct automaton for all possible counter example trees. Test if this automaton accepts any tree.

3. XPath Containment Test

Automaton Technique

Recall: for any DTD there is a tree automaton which recognized the corresponding trees.

Similarly, for any XPath(/,//,[],*,|) expression ex we can construct a (*non-deterministic* bottom-up) tree automaton A which accepts a tree if and only if ex matches the tree.

Theorem
Containment test of XPath(/,//,[],*,|) in the presence of DTDs can be solved in EXPTIME.

→Automata can also be Tested for Finiteness!

Is $p \subseteq_0 q$, for all trees but finitely many exceptions?

Emptiness test for automata

solvable!

Proof Idea construct automaton for all possible counter example trees. Test if this automaton accepts any tree.

3. XPath Containment Test

Chase Technique -- 1979 relational DB's to check query containment in the presence of *integrity constraints*.

("the chase" extends the relational homomorphsim technique)

Example

DTD E =

```
root →  a*
a    →  b* | c*
b    →  d+c+
c    →  b?c?
```

p = /a/b//d
q = /a//c

Is p contained in q for E-conform documents?

First Possibility: use tree automata

→ Construct automata Ap, Aq, AE
→ Construct Bq for the complement of Aq (=not q)
→ Intersect Bq with Ap with AE (gives automaton A)
→ Check if A accepts any tree.

3. XPath Containment Test

Chase Technique -- 1979 relational DB's to check query containment in the presence of *integrity constraints*.

("the chase" extends the relational homomorphsim technique)

Example

DTD E =

```
root → a*
a    → b* | c*
b    → d+c+
c    → b?c?
```

p = /a/b//d
q = /a//c

Is p contained in q for E-conform documents?

Each b-element has a d-child and a c-child
→ *constraints*

c1: b→d
c2: b→c

p's pattern tree

3. XPath Containment Test

Chase Technique -- 1979 relational DB's to check query containment in the presence of *integrity constraints*.

("the chase" extends the relational homomorphsim technique)

Example

DTD E =

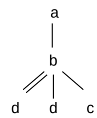
```
root → a*
a    → b* | c*
b    → d+c+
c    → b?c?
```

p = /a/b//d
q = /a//c

Is p contained in q for E-conform documents?

Each b-element has a d-child and a c-child
→ *constraints*

c1: b→d
c2: b→c



p's pattern tree
after *chasing* with c1,c2

3. XPath Containment Test

Chase Technique -- 1979 relational DB's to check query containment in the presence of *integrity constraints*.

("the chase" extends the relational homomorphsim technique)

Example

DTD E =

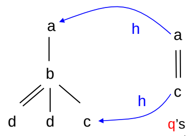
```
root → a*
a    → b* | c*
b    → d+c+
c    → b?c?
```

p = /a/b//d
q = /a//c

Is p contained in q for E-conform documents?

Each b-element has a d-child and a c-child
→ *constraints*

c1: b→d
c2: b→c



q's pattern tree

p is contained in q in the presence of the DTD E ←

p's pattern tree
after *chasing* with c1,c2

END
Lecture 9