University of Washington
Department of Computer Science & Engineering

Managing XML and Semistructured Data

Lecture 9: Query Languages - StruQL and XSL

Prof. Dan Suciu

Spring 2001

In this lecture

- Website management with Strudel
- Background on skolem functions
- Skolem functions in StruQL
- Structural recursion
- XSL

Resources

- Catching the boat with Strudel VLDBJ 2001
- UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion Buneman, Fernandez, Suciu. VLDBJ 2000
- **Data on the Web** Abiteboul, Buneman, Suciu : *sections 5.2, 6.4, 6.5*

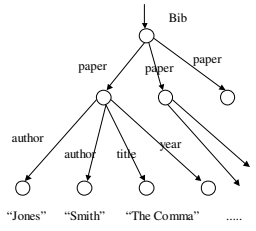
Strudel and StruQL

- Strudel = a Website management tool
- Idea: separate the following three tasks
  - Management of data
    - use some database
  - Management of the site's structure
    - use StruQL
  - Management of the site's presentation
    - use HTML templates (this was before XML...)

Example: Bibliography Data

Input data:

```
{Bib: { paper: { author: "Jones",
                 author: "Smith",
                 title: "The Comma",
                 year: 1994 },
        paper: { author: "Jones",
                 title: "The Dot",
                 year: 1998 },
        paper: { author: "Mark",
                 .... }
        . . .
      }
}
```



Simple Website Definition in StruQL

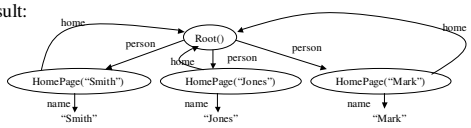
StruQL query:

```
WHERE   Root -> "Bib.paper.author" -> A

CREATE  Root(), HomePage(A)

LINK    Root() -> "person" -> HomePage(A),
        HomePage(A) -> "name" -> A
        HomePage(A) -> "home" -> Root()
```

Result:



Root(), HomePage(A) = ***Skolem Functions*** (more later)

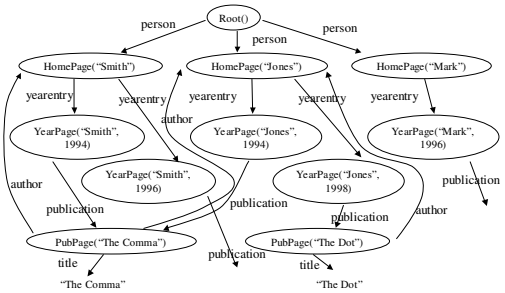
Complex Website Definition in StruQL

```
WHERE Root -> "Bib" -> X, X -> "paper" -> P,
      P -> "author" -> A, P -> "title" -> T, P -> "year" -> Y

CREATE  Root(), HomePage(A), YearPage(A,Y), PubPage(P)

LINK    Root() -> "person" -> HomePage(A),
        HomePage(A) -> "yearentry" -> YearPage(A,Y),
        YearPage(A,Y) -> "publication" -> PubPage(P),
        PubPage(P) -> "author" -> HomePage(A),
        PubPage(P) -> "title" -> T
```

Example: A Complex Web Site



Skolem Functions

- Maier, 1986
  - in OO systems
- Kifer et al, 1989
  - F-logic
- Hull and Yoshikawa, 1990
  - deductive db (ILOG)
- Papakonstantinou et al., 1996
  - semistructured db (MSL)

Skolem Functions in Logic

Origins: First Order Logic

The Satisfiability problem
given a formula φ, does it have a model ?

Skolem Functions in Logic

- Example: does φ have a model ?

$$\varphi = \forall x.\forall y.(R(x, y) \Rightarrow \exists z.(R(x, z) \wedge R(y, z)))$$

Skolem functions:
replace ∃ with functions, drop ∀

$$\varphi' = R(x, y) \Rightarrow (R(x, f(x, y)) \wedge R(y, f(x, y))))$$

Fact: φ has a model iff φ' "has a model"

Skolem Functions in Databases

Recall Datalog:

```
Answer(title, author) :- Paper(author, title, year)
```

Means:

$$\forall author.\forall title.\forall year.(Answer(author, title) \Leftarrow Paper(author, title, year))$$

Skolem Functions in Databases

Now consider:

```
Answer(author, x) :- Paper(author, title, year)
```

I want to "create a new object x". What meaning ?

$$\forall author.\forall title.\forall x.\forall year.(Answer(author, x) \Leftarrow Paper(author, title, year))$$

$$\forall author.\forall title.\exists x.\forall year.(Answer(author, x) \Leftarrow Paper(author, title, year))$$

$$\forall author.\exists x.\forall title.\forall year.(Answer(author, x) \Leftarrow Paper(author, title, year))$$

Skolem Functions in Databases

Better: use Skolem functions directly in Datalog

Choices:

```
Answer(author, NewObj(author)) :- Paper(author, title, year)

Answer(author, NewObj(author,title)) :- Paper(author, title, year)

Answer(author, NewObj(title,year)) :- Paper(author, title, year)

Answer(author, NewObj()) :- Paper(author, title, year)
```

Skolem Functions in StruQL

StruQL's semantics:

- Input graph: (Node, Edge)
- Output graph:(Node', Edge')

Example:

```
WHERE   Root -> "Bib.paper.author" -> A
CREATE  Root(), HomePage(A)
LINK    Root() -> "person" -> HomePage(A),
        HomePage(A) -> "name" -> A
        HomePage(A) -> "home" -> Root()
```

```
Node'(Root()) :-
Node'(HomePage(A)) :-                 Edge(Root,Bib,X), Edge(X,paper,Y),Edge(Y,author,A)
Edge'(Root,person,HomePage(A)) :-     Edge(Root,Bib,X), Edge(X,paper,Y),Edge(Y,author,A)
Edge'(HomePage(A),person, A) :-       Edge(Root,Bib,X), Edge(X,paper,Y),Edge(Y,author,A)
Edge'(HomePage(A),home,Root()) :-     Edge(Root,Bib,X), Edge(X,paper,Y),Edge(Y,author,A)
```

A Different Paradigm: Structural Recursion

Data as *sets* with a *union* operator:

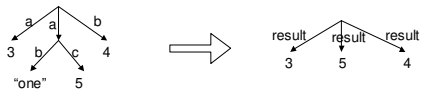
{a:3, a:{b:"one", c:5}, b:4} =

{a:3} U {a:{b:"one",c:5}} U {b:4}

Structural Recursion

Example: retrieve all integers in the data

```
f($T1 U $T2)   = f($T1) U f($T2)
f({$L: $T})    = f($T)
f({})          = {}
f($V)          = if isInt($V) then {result: $V} else {}
```



Structural Recursion

What does this do ?

```
f($T1 U $T2)  = f($T1) U f($T2)
f({$L: $T})   = if $L=a then {b:f($T)} else {$L:f($T)}
f({})         = {}
f($V)         = $V
```

Returns the same tree with a-edges replaced by b-edges

Structural Recursion

What does this do ?

```
f($T1 U $T2)  = f($T1) U f($T2)
f({$L: $T})   = {$L:{$L:f($T)}}
f({})         = {}
f($V)         = $V
```

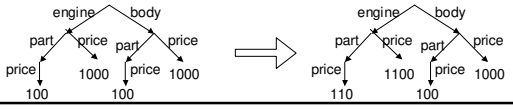
Input = tree with n nodes
Output = tree with 2n nodes (every edge is doubled)

Structural Recursion

Example: increase all engine prices by 10%

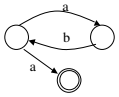
```
f($T1 U $T2) = f($T1) U f($T2)
f({$L: $T})  = if $L= engine
                 then {$L: g($T)}
               else {$L: f($T)}
f({})        = {}
f($V)        = $V
```

```
g($T1 U $T2) = g($T1) U g($T2)
g({$L: $T})  = if $L= price
                 then {$L:1.1*$T}
               else {$L: g($T)}
g({})        = {}
g($V)        = $V
```



Structural Recursion

Retrieve all subtrees reachable by (a.b)*.a




```
f($T1 U $T2) = f($T1) U f($T2)
f({$L: $T})  = if $L= a
                 then g($T} U $T
               else { }
f({})        = { }
f($V)        = { }
```

```
g($T1 U $T2) = g($T1) U g($T2)
g({$L: $T})  = if $L= b
                 then f($T)
               else { }
g({})        = { }
g($V)        = { }
```

Structural Recursion: General Form

```
f1($T1 U $T2) = f1($T1) U f1($T2)
f1({$L: $T})  = E1($L, f1($T),...,fk($T), $T)
f1({})        = { }
f1($V)        = { }
```

. . . .

```
fk($T1 U $T2) = fk($T1) U fk($T2)
fk({$L: $T})  = Ek($L, f1($T),...,fk($T), $T)
fk({})        = { }
fk($V)        = { }
```

Each of E_1, ..., E_k consists only of {_ : _}, U, if_then_else_

Evaluating Structural Recursion

Recursive Evaluation:

- Compute the functions recursively, starting with f_1 at the root

Termination is guaranteed.

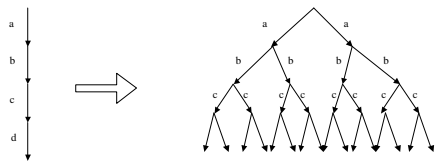
How efficiently can we evaluate this ?

Structural Recursion

Consider this:

```
f($T1 U $T2) = f($T1) U f($T2)
f({$L: $T})  = {$L:f($T)}, $L:f($T)}
f({})        = {}
f($V)        = $V
```

Naive Recursive Evaluation

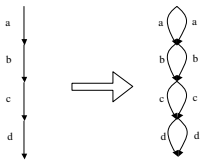


Input tree = n nodes
Output tree = $2^{n+1} - 1$ nodes

Efficient Recursive Evaluation

Recursive Evaluation with function memorization.
PTIME complexity.

```
f($T1 U $T2)  =  f($T1) U f($T2)
f({$L: $T})   = {$L:f($T)}, $L:f($T)}
f({})         =  {}
f($V)         =  $V
```

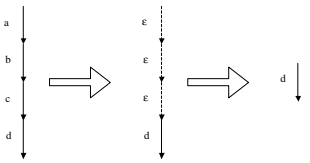


Alternatively: apply the function in parallel to each input edge ➔ **Bulk Evaluation**

Bulk Evaluation

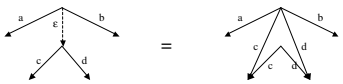
Sometimes f doesn't return anything ➔ use ε edges

```
f($T1 U $T2)  =  f($T1) U f($T2)
f({$L: $T})   =  if $L=c then $T else f($T)
f({})         =  {}
f($V)         =  $V
```



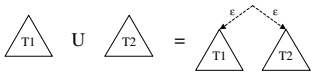
Epsilon Edges

Meaning of ε edges:

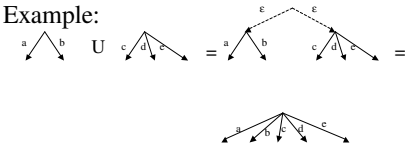


Epsilon Edges

Note: union becomes easy to draw with ε edges:



Example:



Bulk Evaluation

Idea: "apply" E_1, ..., E_k independently on each edge, then connect with ε edges ➔ PTIME

```
f1($T1 U $T2) =  f1($T1) U f1($T2)
f1({$L: $T})  =  E1($L, f1($T),...,fk($T), $T)
f1({})        =  { }
f1($V)        =  { }
```

. . . .

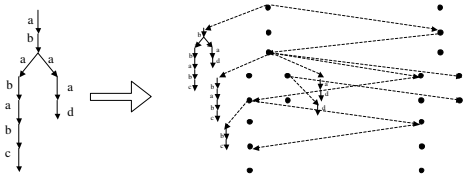
```
fk($T1 U $T2) =  fk($T1) U fk($T2)
fk({$L: $T})  =  Ek($L, f1($T),...,fk($T), $T)
fk({})        =  { }
fk($V)        =  { }
```

Bulk Evaluation

Recall (a.b)*.a:

```
f($T1 U $T2) =  f($T1) U f($T2)
f({$L: $T})  =  if $L= a
                  then g($T} U $T
                else { }
f({})        =  { }
f($V)        =  { }
```

```
g($T1 U $T2) =  g($T1) U g($T2)
g({$L: $T})  =  if $L= b
                  then f($T)
                else { }
g({})        =  { }
g($V)        =  { }
```



Structural Recursion

- Can evaluate in two ways:
  - Recursively: memorize functions' results
  - Bulk: apply all functions on all edges, in parallel, connect, eliminate what is useless
- Complexity: PTIME
  - More precisely: NLOGSPACE
- Works on graphs with cycles too !

XSL

- XSLT 1.0 (a recommendation)
  - http://www.w3.org/TR/xslt.html
- XSLT 1.1: (a working draft)
  - http://www.w3.org/TR/xslt11/
- In commercial products (e.g. IE5.0)

XSL

- Purpose: stylesheet specification language:
  - stylesheet: XML -> HTML
  - in general: XML -> XML
- Uses XPath

XSL Program

- XSL program = template-rule ... template-rule
- template-rule = match pattern + template

Example: Retrieve all book titles:

```
<xsl:template match = "* | /">
        <xsl:apply-templates/>
</xsl:template>

<xsl:template match = "/bib/*/title">
      <result>  <xsl:value-of select = "." />  </result>
</xsl:template>
```

Simple XSL Program

Copy the input:

```
<xsl:template match = "/">
        <xsl:apply-templates/>
</xsl:template>

<xsl:template match = "text()">
         <xsl:value-of select="."/>
</xsl:template>

<xsl:template match = "*">
      <xsl:element name="name(.)">
            <xsl:apply-templates/>
      </xsl:element>
</xsl:template>
```

Flow Control in XSL

```
<xsl:template match = "* | /"> <xsl:apply-templates/>
</xsl:template>

<xsl:template match="a">  <A><xsl:apply-templates/></A>
</xsl:template>

<xsl:template match="b">  <B><xsl:apply-templates/></B>
</xsl:template>

<xsl:template match="c">  <C><xsl:value-of/></C>
</xsl:template>
```

<a> <e> <c> 1 </c>
<c> 2 </c>

<a> <c> 3 </c>

</e>
<c> 4 </c>

<A> <B> <C> 1 </C>
<C> 2 </C>
</B>
<A> <C> 3 </C>
</A>
<C> 4 </C>
</A>

XSL is Structural Recursion

Equivalent to:

```
f(T1 U T2) = f(T1) U f(T2)
f({L: T})  =  if L= c then {C: t}          ← <xsl:template match="c">
              else L= b then {B: f(t)}     ← <xsl:template match="b">
              else L= a then {A: f(t)}     ← <xsl:template match="a">
              else f(t)                    ← <xsl:template match = "* | /">
f({})      =  {}
f(V)       =  V
```

XSL query = single function
XSL query with *modes* = multiple function (next)

Modes in XSLT

Compute the path (a.b)* :

```
<xsl:template match = "/">
      <xsl:apply-templates mode="f"/>
</xsl:template>

<xsl:template match="*" mode="f"/>

<xsl:template match="a" mode="f">
  <result> <xsl:copy-of match="."/> </result>
  <xsl:apply-templates mode="g"/>
</xsl:template>

<xsl:template match="*" mode="g">

<xsl:template match="b" mode="g">
  <xsl:apply-templates  mode="f"/>
</xsl:template>
```

```
f(T1 U T2) = f(T1) U f(T2)
f({a: T})  =  {result:T} U g(T)
f({})      =  {}
f(V)       =  V

g(T1 U T2) = g(T1) U g(T2)
g({b: T})  =  f(T)
g({})      =  {}
g(V)       =  V
```

<xsl:copy-of ... > copies the input to the output

← ignoring modes, this computes (a|b)*

Modes in XSLT

- Mode = a name for a group of template rules
- No mode = empty mode
- Same as having multiple recursive functions

Conflict Resolution for Template Rules

If several template rules match, choose that with highest "priority".

- Explicit priority: <xsl:template match="abc" priority="1.41">
- Computing implicit priority: ad-hoc rules given by the W3C, based on match
  - match="P1 | P2 | ..." → transform to a set of template rules.
  - match="abc" → the priority is 0.
  - match="[... some namespace name...]" → the priority is -0.25.
  - match="node()" → the priority is -0.5.
  - Otherwise, the priority is 0.5

It is an error if this leaves more than one matching template rule.

Built-in Template Rules

- Keeps us going:

  <xsl:template match = "* | /"> <xsl:apply-templates/>
  </xsl:template>

  there is one such rule for each mode

- Copies what we forgot:

  <xsl:template match = "text()|@*"><xsl:value-of select="."/>
  </xsl:template>

  there is only one rule, for the empty mode

- Lowest priorities among all rules; hence, can be easily overridden

XSL Template

<xsl:template match = "expression" mode = "name" priority = "number" name = "name" >
Body
</xsl:template>

Default: mode = "" priority = (computed as explained earlier) name = when no match, no mode

Body =
- XML constructors: <myTag>...</myTag> ... ...
- XSL instructions:
  - <xsl:apply-templates> (= recursive call)
  - <xsl:value-of> (= copy the value)
  - <xsl:copy> (= shallow copy)
  - <xsl:copy-of> (=deep copy)
  - <xsl:element> (= more flexible than XML constructors)
  - <xsl:attribute> (= add an attribute to the element)
  - <xsl:if> (= conditional)
  - <xsl:for-each>
  - Instructions for variables

XSL Apply Templates

<xsl:apply-templates select = "expression" mode = "name" >
Body
</xsl: apply-templates>

- Default
  - select = "*" (children)
  - mode = "" (empty mode)

Body:
- "Sort" instructions
- "Parameмter" instructions

XSL Variables

Declaring a variable
- <xsl:variable name = "vname" select = "value"> value </xsl:variable>
- Value = either in select, or in body
- Either in <xsl:template> ... </xsl:template> or at top level

Declaring a parameter:
- <xsl:param select = "value"> value </xsl:param>
- In <xsl:template> ... </xsl:template>, at the beginning

Passing a parameмter
- <xsl:with-param select = "value"> value </xsl:param>
- In <xsl:apply-templates> ... </xsl:apply-templates >

Using variables: {$vname}

XSL and Structural Recursion

XSL:
- mainly on trees
- may loop

Structural Recursion:
- arbitrary graphs
- always terminates

add the following rule:

```
<xsl:template match = "e">
    <xsl:apply-patterns select="/"/>
</xsl:template>
```

stack overflow on IE 5.0