Foundations of XML Data Manipulation

Giorgio Ghelli

Type Systems for SSD

Plan of the lesson

- XDuce type system (xduce-toit.2003, google:xduce)
- Tree automata (TATA, google:tree automata)
- DTD's
- XSD
- Mu-calculus and TQL Logic
- Path inclusion

A Query

```
for  $b in $doc /bib/book,
     $a in $b /author
where $b /@year > 2000
return booksbyaut [$a,
                for  $bb in $doc /bib/book
                where  $a isin $bb /author
                return $bb/title
         ]
```

Its Type

- Type:

  Bookbyaut[author[String], title[String]*]*
- If:

```
$doc: bib[  book [   title[String],
                     author[String]*
            ]*
      ]
```

Types for SSD

- XDuce type system:

T ::=	B	base types
	m[T]	tree types
	T, T	forest concatenation types
	0	empty tree singleton type
	T | T	union type
	X	m[ ]-guarded recursion
	T*	Kleene star type

Recursion

- Section = para[] | section[Section*]
- Illegal:
  - paraList = para[] | (para[], paraList)
- Equivalent to:
  - paralist = para[], (para[])*
- Illegal:
  - X = a[], X , b[] | 0

Type rules

$$\frac{}{0: 0} \qquad \frac{t: T}{m[t] : m[T]} \qquad \frac{t: T \quad u: U}{t, u : T, U}$$

$$\frac{t: T}{t: T|U} \qquad \frac{}{0 : T^*} \qquad \frac{t: T \quad u: T^*}{t, u : T^*}$$

$$\frac{X=T \vdash t: T}{X=T \vdash t: X}$$

Not so different?

- Is T, T’ the same as T × T’?
- Is T* the same as List(T)?
- Is a[T], b[U], c[V] the same as record[a:T, b:U, c:V]?

T,T’ vs. TxT’

- (T, T’), T” = T, (T’, T”) = T, T’, T”
- T, 0 = T but T × 1 =/= T
- <t,u>: T × U iff t: T and u: U
- It may be that t, u : T, U but not t:T and u: U:
  - (a[],b[]),c[]: (a[],(b[],c[]))
  - a[], (b[], c[]): (a[] | a[],b[]) , (b[],b[] | c[])
  - Type checking similar to regular-language testing
- T × U < T’ × U’ iff T < T’ and U < U’
- It may be that T, U < T’, U’ but   T ≮ T’ and U ≮ U’
- Subtyping defined as language inclusion, and checked as automata inclusion

The semantic intuition

- At run time, a value of type T × U is represented as pair(t,u)
- A value of type T × U is “bigger” than a value of type T
- If t:T then t has NOT type T×U
- A value of type T,U has no tag(_,_) structure
- If t:T and 0:U then t: (T,U)

a[T],b[U] vs. tup[a:T] + tup[b:U]

- Record concatenation:
  - tup[a:T] + tup[b:U] = tup[a:T, b:U]
- Usually T+U only defined when T is a simple-record type tup[a:T]; T,U accepts any type as T, including (a[] | 0)

T* vs. List(T)

- T* = μTS. 0 | (T,TS)
- List(T) = μLT. 1 | (T$\times$LT)
- Hence:
  - T* = T**
  - List(T) $\neq$ List(List(T))
  - T < T*: no List tag at run-time
  - T $\not\leq$ List(T)

Guarded recursion

- Regular grammars:
  - X = a1 | a2 | a3.X1 | a4.Xn
- XDuce "linear" recursive types:
  - X = a1[] | a2[] | a3[X1] | a4[X2]
  - Correspond to automata
- XDuce tree-like recursive types
  - X = a1[] | a2[] | a3[X1,X2] | a4[X3,X4]
  - Correspond to tree automata

Horizontal and Vertical RegExps

- Horizontal:
  - X = a[](b[])* | (a[])*b[]
- Vertical
  - X = a[Y] | W
  - Y = 0 | b[Y]
  - W = b[] | a[W]
- Tree automata because of Vertical
- *Regular* tree automata because of Horizontal

Tree Automata

- (A,Q,R,F) with
  - F $\subseteq$ Lists(Q)
  - R $\subseteq$ A $\times$ Lists(Q) $\times$ Q (set of a[q1,…,qn]$\rightarrow$q rules)
- A run:
  - Substitute a[] with q (a^q[]) if a[]$\rightarrow$q $\in$ R
  - Substitute a[q1,…,qn] with q if a[q1,…,qn]$\rightarrow$q $\in$ R
  - Accept t1,…,tn if rewritten as q1,…,qn $\in$ F

Unranked Tree Automata

- Ranked Tree Automata:
  - Rules are like a_2[q1,q2]$\rightarrow$q3: for each binary simbol, I need $2^{3*|Q|}$ rules at most
- But: A x Lists(Q) x Q is not finite:
  - Rules like a[q1,…,q1]$\rightarrow$q2
- Problem:
  - Representing R and deciding a[q1,…,qn]$\rightarrow$q $\in$ R

Regular Tree Automata

- Regular Tree Automata:
  - For each a, q, the language (in Q*) {q1,…,qn | a[q1,…,qn]->q $\in$ R} is regular
- R can be represented as a function of type AxQ -> RegExp(Q)
- Tree automata correspond to vertical recursion
- *Regular* in *R*TA corresponds to horizontal regular recursion

Unranked binary expressions

- Or(F,And(T,T,F),And(F,T))
  where T = And(), F = Or()
- A={And,Or}; Q={t,f}
- R:
  - Or(f*) -> f
  - Or((t|f)*, t, (t|f)*) -> t
  - And(t*) -> t
  - And ((t|f)*, f, (t|f)*) -> f

We like automata

- Recognize trees in linear time
- Closed by union, intersection, *complement*
- Emptyness is decidable
- A <: A' iff
  A \ A' = A $\cap$ Co(A') is empty

DTDs

- Canonical way to describe the structure of an XML document
- Example:

```
<!ELEMENT people_list (person*)>
<!ELEMENT person (name, birthdate?,
  children?)>
<!ELEMENT children (person+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT birthdate (#PCDATA)>
```

An example

- DTD:

```
<!ELEMENT people_list (person*)>
<!ELEMENT person (name, birthdate?, children?)>
<!ELEMENT children (person+)>
```

- Document:

```
– <!DOCTYPE people_list SYSTEM "example.dtd">
  <people_list>
      <person>
          <name>Fred Bloggs</name>
          <birthdate>…</birthdate>
          <children>
              <person><name>Jim
              </name></person>
          </children>
      </person>
      <person><name>Luis Gutierrez</name>
      </person>
  </people_list>
```

XDuce vs. DTD

- XDuce: a set of mutual recursive defs:
  - A = a[B1*,B2]
  - B1 = b[X]
  - B2 = b[Y]
  - …
- DTD: the type is identified by the label:
  - a = a[b*,c]
  - b = b[X]
  - c = c[Y]

DTD into automata

- DTD:

```
<!ELEMENT people_list (person*)>
<!ELEMENT person (name, birthdate?, children?)>
<!ELEMENT children (person+)>
```

- Automaton:
  - F = PL
  - R:
    - people_list[P*] -> PL
    - person[N,BD?,C?] -> P
    - children[P+] -> C
    - name[ ] -> N …

XDuce into automata

- XDuce:
  - Paper where
  - Paper = title[], section[abstract[]], Content
  - Content = (paragraph[] | section[Content]) *
- Automa:
  - F = T,SA,(P|SC)*
  - R:
    - title[] -> T, abstract[] -> A,
    - section[A] -> SA
    - paragraph[] -> P
    - section[(P|SC)*] -> SC

XSD

- Local element types are not label-identified, but global element types are:
  - a = a[b[X]]
  - b = b[a[Z]]
  - c = c[a[W]]
- Not every regexp is OK (Unique Particle Attribution):
  - a = a[b[X]*, b[X]]: illegal
- In one element, label identifies type (Element Declarations Consistent):
  - a = a[b[X], b[Y]]: illegal

XSD

- Names are qualified with respect to namespaces
- XSD can specify key and keyref constraints
- Two limited forms of subtyping by name: derivation and substitution groups

XSD Syntax

- Global element declarations:
  - element El of T
  - element El of (type [T] of {…})
  - May be local, in which case El is not a key
- Complex type definitions:
  - type T of
  - Either anonymous, or T is a key (even if local)

XSD Assessmnet

- Assessment:
  - local validation, schema-validity assessment and infoset augmentation: Infoset -> PSVI
- PSVI contains:
  - Normalized and default values for attributes and elements
  - Type definitions for attributes and elements
  - Validation outcome

XSD and Subtyping

- Derivation:
  - Every complex type either extends or restricts another type, starting from xsi:anyType
  - Explicit cast: the derived type can be used for validation only if a corresponding xsi:type attribute is present in the element to validate, or if the element name is in the substitution group of the expected name
- Substitution:
  - An element name may be head of a substitution group, and the other names from the group are valid where the head is required
  - The types of the group elements must be derived from the type of the head

Semantic subtyping

- Rule-based subtyping:
  - Subsumption: T <: T' ⇒ ∀t. t:T ⇒ t:T'
- Semantic subtyping:
  - Definition: (∀t. t∈[[T]] ⇒ t∈[[T']]) ⇒ T <: T'
- For example:
  - (Forall X. X→X) <:? Int → Int
- XSD: rule-based subtyping
- XDuce, CDuce: semantic subtyping

DTD and μ-calculus

- [everywhere] A = νξ (A ∧ [↓]ξ ∧ [→]ξ)
- We extend μ with equations:
  - A where $\$x_1 = A_1, \ldots, \$x_n = A_n$
  - A(ξ) where ξ = A_1
    is the same as
    A(μξ. A_1)
- Still checkable in O(2^n)

DTD and μ-calculus

- DTD:
  <!ELEMENT people_list (person*)>
  <!ELEMENT person (name, birthdate?, children?)>
  <!ELEMENT children (person+)>
- μ with equations:
  [everywhere] (people_list ∧ [↓] $PersonPlus)
  ∨ (person ∧ <↓> $NBC)
  ∨ (children ∧ <↓> $PersonPlus)
  ∨ (name ∧ [↓]False) ∨ …
  where $PersonPlus = person ∧ [→] $PersonPlus
  $NBC = name ∧ [→] $BC
  $BC = (birthdate ∧ [→] $C) ∨ children

TQL Logic

- Ordered TQL logic:

A ::=	B	base values
	η[A]	trees (η: x or n)
	A , A	sequence
	0	empty tree singleton sentence
	T ∨ T	disjunction
	¬A	negation
	μξ. A	(positive) recursion
	ξ	recursion variable
	∃x.A	label quantification
	∃X.A	forest quantification
	X	forest variable

The actual TQL logic

- TQL data model in unordered:
  - 0|t = t ; (t|t')|t'' = t|(t'|t'') ; t | t' = t' | t
- In TQL ordered logic:
  - t ⊨ (A , B)
    iff ∃t', t''. t',t'' = t and t' ⊨ A and t'' ⊨ B
- In TQL logic:
  - t ⊨ A | B
    iff ∃t', t''. t' | t'' = t and t' ⊨ A and t'' ⊨ B

TQL Logic

- F ⊨ True: always (True = 0 ∨ ¬0)
- F ⊨ 0 iff F=0
- F ⊨ A | B iff ∃F', F''. F = F'|F'', F' ⊨ A, F'' ⊨ B
  F ⊨ m[A] iff F=m[F'], F' ⊨ A
- E.g.:
  - a[0] | b[0] ⊨ b[0] | True ?
  - b[0] ⊨ b[0] | True ?
  - a[0] | b[0] ⊨ b[0] ?
  - a[b[0]] ⊨ a[True] | True?

Other operators

- F ⊨ A ∧ B iff F ⊨ A and F ⊨ B
- F ⊨ ¬A iff not (F ⊨ A)
- Derived operators:
  - A ∨ B $=_{def}$ ¬(¬A ∧ ¬B)
  - A || B $=_{def}$ ¬(¬A | ¬B)
  - m[⇒ A] $=_{def}$ ¬m[¬A]
- F ⊨ (a[True] ∨ b[True]) | True
- F ⊨ (a[0] | True) ∧ ¬(a[0] | a[0] | True)
- F ⊨ author[⇒ Hull] || False

More than types?

- Complement ¬A dualizes every other operator (∃→∀, ∨→∧, μ→ν,…)
- Horizontal recursion is more than T*
- Quantification expresses correlation:
  - ∃x. x[True] | x[True] | True
  - ∃x. .x[A] ∧ .x[A] (.x[A] = x[A] | True)
  - ∃x. .x[A] | .x[A] (.x[A] = x[A] | True)
- Logic can express key constraints:
  - ¬∃X. .book[.t[X]] | .book[.t[X]]

Decidability

- Quantification makes emptyness undecidable
- Quantification makes model-checking (type-checking) PSpace-complete
- Model-checking is often doable in practice

Path containment

Path containment

- As binary relation: sub_2
  - p sub_2 q ⇔ (m p n ⇒ m q n) ⇔ [[p]] ⊆ [[q]]
- Starting from the root: sub_1
  - p sub_1 q ⇔ (root p n ⇒ root q n)
- Boolean containment, starting from the root:
  - t ⊨ p: matching p against the root of t yields non-empty result
  - p sub_0 q iff t ⊨ p ⇒ t ⊨ q

Notions of containment

- If we restrict to child/desc, sub_2 e sub_1 are equivalent
- In the presence of predicates, sub_1 can be mapped to sub_0 :
  - p sub_1 q ⇔ p[x] sub_0 q[x], where:
    - p[x] adds a child::x condition to the selection node, and x is fresh (Miklau-Suciu PODS 02)

Complexity for PositiveXPath

- PTime:
  - No disjunction, 2 of // [] * but not all: XP(/,//,*), XP(/,[],*), XP(/,//,[]), XP(/,//)+DTD,
- coNP:
  - XP(/,|); XP(//,|);
  - XP(/,[])+DTD, XP(//,[])+DTD
  - XP(/,//,[],*);
  - XP(/,//,[],*,|) (becomes PSPACE if the alphabet is finite);
- ExpTime:
  - XP(/,//,|) + DTD
  - XP(/,//,[],|,*) + DTD

Path inclusion and μ-calculus

- Let <<p>> be the tranlation of a path and <<s>> the translation of a schema:
  - E,L, m ⊨ <<s>> if E,L satisfies s
- p sub_2 q:
  - Valid (<<p>> ⇒ <<q>>)
  - i.e., for any E,L,m,n:

    E,L,i->n, m ⊨ <<p>> ⇒ <<q>>
- p sub_2 q under s:
  - For any E,L,m,n:

    E,L,i->n, m ⊨ <<p>> /\ <<s>> ⇒ <<q>>
- <<_>> is linear ⇒ inclusion is $O(2^n)$ for NavXPath