Why type-check XML transformations?
- Goal: Check that valid input -> valid output
  - e.g. check that XSLT/XQuery produces valid HTML
- Can we prove this once and for all?
  - rather than revalidating after every run?
- Typechecking can also support other goals:
  - optimization/static analysis
  - identifying silly bugs ("dead" code that never matches anything; missing cases)

Example: XSLT
- Turns a sequence of People into a table
- HTML Tables have to have at least one row!

```xml
<xsl:output method="html"/>
<xsl:template match="/person">
  <html>
    <head><title>Phone Book</title></head>
    <body>
      <table frame="box" rules="all">
        <xsl:apply-templates/>
      </table>
    </body>
  </html>
</xsl:template>
<xsl:template match="record">
  <tr>
    <td><xsl:copy-of select="name"/></td>
    <td><xsl:copy-of select="phone"/></td>
  </tr>
</xsl:template>
```
Influential approach: XDuce

- A functional language for transforming XML based on pattern matching

```haskell
module XDuce where

import Prelude

type Person = person[name[String],
    email[String]*,
    phone[String]?]

fun person2row(var p as Person) : Tr =
    match p with
    person[name[var name],
           email[var email]] ->
    tr[td[name],td[email]]
    | ...
```

XDuce, continued

```haskell
fun people2rows(var xs as Person+) : Tr+ =
    match xs with
    var p as Person, var ps as Person*
    -> person2row p,people2rows ps
    | var p as Person -> person2row p

fun people2table(var xs as Person+) : Table =
table[people2rows xs]
```

Regular expression types

- $\tau ::= \text{String} | \text{X} | a[\tau] | \tau, \tau | \tau|\tau | \tau*$
- $a[\tau]$ means "element labeled a with content $\tau$"
- $\text{X}$ is a type variable
- Can model DTDs, XML Schemas as special case
  - at least as far as element structure goes
  - does not model attributes, or XML Schema interleaving

Recursive type definitions

- Consider type definitions
  
  ```haskell
type X_1 = \tau_1
  type X_2 = \tau_2 ...
  ```

- Definitions may be recursive, but recursion must be "guarded" by element tag
- e.g. $\text{type } x = x, x$ not allowed
Meaning of types

• Suppose \( E(X) \) is the definition of \( X \) for each type variable \( X \) (i.e. type \( X = E(X) \) declared)
• Define \( [\tau]_E \) as follows:
  • \( [X]_E = [E(X)]_E \)
  • \( [\text{String}]_E = \Sigma^* \)
  • \( [a[\tau]]_E = \{a[v] \mid v \in [\tau]_E\} \)
  • \( [\tau_1,\tau_2]_E = \{v_1,v_2 \mid v_1 \in [\tau_1]_E, v_2 \in [\tau_2]_E\} \)
  • \( [\tau_1|\tau_2]_E = [\tau_1]_E \cup [\tau_2]_E \)
  • \( [\tau^*]_E = \{v_1,...,v_n \mid v_1,...,v_n \in [\tau]_E\} \)

DTDs as a special case

• DTD rules:
  \( a \rightarrow b,c^*,(d|e)?,f \)
  \( b \rightarrow c,d \ ... \)
  becomes:
  type \( A = a[B,C^*,(D|E)?,F] \)
  type \( B = b[C,D] \ ... \)
• (just introduce new type \( \text{Elt} \) for each element name \( \text{elt} \))
  • Same idea works for XML Schemas

Complications

• Typechecking undecidable for any Turing-complete language
  • and for many simpler ones
• Schema languages for XML use regular languages/tree automata
  • decision problems typically EXPTIME-complete
• Nevertheless, efficient-in-practice techniques can be developed

Other checks

• Typing: All results are contained in declared types
• Exhaustiveness: every possible input matches some pattern
  
  ```
  match (x:Person) with
  person[name[var n]] -> ...
  ```
• Irredundancy: no pattern is "vacuous"
  
  ```
  match (x:Person) with
  person[name[var n],var es as Email*] -> ...
  | person[name[var n],email[var e]] -> ...
  ```
• All can be reduced to **subtype** checks
Subtyping as inclusion

- We can define a semantic subtyping relation:
  - \( \tau <: \tau' \iff \llbracket \tau \rrbracket_E \subseteq \llbracket \tau' \rrbracket_E \)
- i.e. every tree matching \( \tau \) also matches \( \tau' \).
- How can we check this?
- One solution: tree automata

Type inclusion & tree automata

XML trees as binary trees

- First-child, next-sibling encoding
- Leaves are "nil" / ()
Binary tree automata (bottom-up)

- $M = (Q, \Sigma, \delta, q_0, F)$ is a (bottom-up) binary tree automaton if:
  - $q_0 \in Q$ -- initial state (leaf nodes)
  - $F \subseteq Q$ -- set of final states
  - $\delta : \Sigma \times Q \times Q \to Q$ -- transition function

Binary tree automata (top-down)

- $M = (Q, \Sigma, \delta, q_0, F)$ is a (top-down) binary tree automaton if:
  - $q_0 \in Q$ -- initial state (root node)
  - $F \subseteq Q$ -- set of final states (for all leaf nodes)
  - $\delta : Q \times \Sigma \to Q \times Q$ -- transition function

Deterministic top-down TA are strictly less expressive than bottom-up TA

- however, nondeterministic versions equivalent

DTDs and ambiguity

- Recall DTDs have to be "unambiguous"
- Example: NOT $(a+b)+(a+c)$
  - but $a+b+c$ is equivalent
- NOT $(a+b)^*a$
  - but $(b^*a)^*$ is equivalent

These restrictions (and similar ones for XML Schemas) lead to deterministic, top-down (tree) automata.

Translating types
Translating types

• First give each subexpression its own type name

type Person = person[name[String],
                  email[String]*,
                  phone[String]?]

To binary form

• Next reorganize into binary steps

type Person = person[Q1],Empty

type Q1 = name[String], Q2
       | name[String], Q3

type Q2 = email[String], Q2
       | email[String], Q3

type Q3 = phone[String], Empty | ()

type Empty = ()

To tree automaton

type Person = person[Q1], Empty

type Q1 = name[String], Q2
       | name[String], Q3

type Q2 = email[String], Q2
       | email[String], Q3

type Q3 = phone[String], Empty | ()

type Empty = ()

Q = {Person, Q1, Q2, Q3, String, Empty}
Σ = {person, name, email, phone, string}
q0 = Person
F = {Empty, Q3, String}
To tree automaton

```
type Person  = person[Q1],Empty
type Q1 = name[String], Q2
        | name[String], Q3
type Q2 = email[String], Q2
        | email[String], Q3
type Q3 = phone[String],Empty | ()
type Empty = ()
```

```
Q = {Person,Q1,Q2,Q3,String}
Σ = {person,name,email,phone,string}
q₀ = Person
F = {Empty,Q3,String}
```

Checking containment

- Like sequential case, tree automata/languages closed under union, intersection, negation
- Some of the constructions incur exponential blowup
- Containment testing is EXPTIME-complete
- Naive algorithm: test $L(M) \subseteq L(N)$ by constructing automaton $M'$ for $L(M) \cap \overline{L(N)}$, testing emptiness
- Does not perform well on problems encountered in programming
- Hosoya, Vouillon & Pierce [2005] give practical algorithm aimed at XDuce typing problems

Motivation

[Collazzo et al. 2006]

- For XDuce, want to ensure valid result
- Find silly bugs in pattern matching
- Ensure exhaustiveness & irredundancy
- For XQuery, still want result validity, but different issues
  - Functions, but no pattern matching (as such)
  - But does have for/iteration & path expressions
  - Parts of queries can have "silly" bugs
Example

- Schema (we'll use XDuce notation):
  
  \[
  \text{type Person = person[name[String], email[String]*, phone[String]?]}
  \]

- Query: return all name/phone of people with email
  
  for $y$ in $x/person[emial]$ return $y/name,y/phone$

- **Silly bug**: "emial" misspelled; path will never select anything (in this type anyway)

### Typing rules for XQuery: Basics

\[
\frac{x:\alpha \in \Gamma}{\Gamma \vdash x : \alpha} \quad \frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \frac{w \in \Sigma^*}{\Gamma \vdash w : \text{string}} \quad \frac{b \in \text{Bool}}{\Gamma \vdash b : \text{bool}}
\]

\[
\frac{\Gamma \vdash (e) : \cdot}{\Gamma \vdash () : ()} \quad \frac{\Gamma \vdash n[e] : n[\tau]}{\Gamma \vdash e : \tau}
\]

\[
\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e, e' : \tau, \tau'} \quad \frac{\Gamma \vdash c : \text{bool} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{if } c \text{ then } e_1 \text{ else } e_2 : \tau_1 | \tau_2}
\]

### Typing rules for XQuery: Let, If

\[
e ::= () \mid e, e' \mid n[e] \mid w \mid x \mid \text{let } x = e \text{ in } e' \mid \text{if } c \text{ then } e \text{ else } e' \mid \bar{x}/\text{child} \mid e : n \mid \text{for } \bar{x} \in e \text{ return } e'
\]

\[
\frac{}{\Gamma \vdash \cdot} \quad \frac{\Gamma, x : \tau}{\Gamma, \bar{x} : \alpha}
\]

Tree variables $\bar{x}$ have single tree values $\bar{v}$
Forests variables $x$ have arbitrary values $\nu$
**Path steps**

\[ \frac{x : n[\tau] \in \Gamma}{\Gamma \vdash x / \text{child} : \tau} \quad \frac{\Gamma \vdash e : \tau \quad \tau :: n \Rightarrow \tau'}{\Gamma \vdash e :: n : \tau'} \]

**Example**

```
type Doc = a[ b[*], c[[]], f[]]?
prime( b[*], c[[]], f[] ) = b[*]|c[[]]|f[]
quantifier( b[*], c[[]], f[] ) = *

$doc : Doc \vdash \text{doc/a/* : b[*]|c[[]]|f[]}$
$doc : Doc, $x : b[*] | c[[]], f[] \vdash $x/* : d[] | e[]
$doc : Doc \vdash \text{for $x$ in doc/a/* return $x/* : (d[] | e[])|f[]}$
```

**Overapproximation:**

\( e, f, d \) and \( e, f, e, f \) can never happen

**Comprehensions:**

**simple idea** (what XQuery does)

\[
\begin{align*}
\text{statEnv I- Exp}_{1} : \text{Type}_{1} & \\
\text{statEnv I- VarName}_{1} \text{ of var expands to Variable}_{1} & \\
\text{statEnv + varType(Variable}_{1} \Rightarrow \text{prime(Type}_{1} ))) & \text{ I- Exp}_{2} : \text{Type}_{2} \\
\text{statEnv I- for$ VarName_{1}$ in Exp}_{1} \text{ return Exp}_{2} : \text{Type}_{2} \cdot \text{quantifier(Type}_{1} )
\end{align*}
\]

**Comprehensions: more precisely**

\[
\begin{align*}
\Gamma \vdash e_{1} : \tau_{1} & \quad \Gamma \vdash \bar{x} \text{ in } \tau_{1} \rightarrow e_{2} : \tau_{2} \\
\Gamma \vdash \text{for } \bar{x} \in e_{1} \text{ return } e_{2} : \tau_{2}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \bar{x} \text{ in } E(X) \rightarrow e : \tau & \\
\Gamma, \bar{x} : \alpha \vdash e : \tau & \\
\Gamma, \bar{x} \text{ in } X \rightarrow e : \tau & \\
\Gamma, \bar{x} \text{ in } \tau_{1} \rightarrow e : \tau_{2} & \\
\Gamma, \bar{x} \text{ in } \tau_{1} \rightarrow e : \tau_{2} & \\
\Gamma, \bar{x} \text{ in } \tau_{1} \rightarrow e : \tau_{1}' & \\
\Gamma, \bar{x} \text{ in } \tau_{2} \rightarrow e : \tau_{2}'
\end{align*}
\]
Example

type Doc = a[ b[d][*], c[e[]], f[] ];
prime( b[d][*], c[e[]], f[] ) = b[d][]c[e], f[]
quantifier( b[d][*], c[e[]], f[] ) = *

Remembers more of the regexp structure
(but overall system still approximate)

Path errors

Detect path errors by:
• Noticing when a non-() expression has type ()
• Tracking which parts of expression have this property
• (propagating sets of expression locations through
typing judgements)

Subtleties

• [Collazzo et al. 2006] also considers splitting
• E.g. considering a[T1]...|Tn] as a[T1]|...|a[Tn]
• This complicates system, but can make results more precise
**Motivation**

[Benedikt & Cheney, 2009]

- Suppose we want to maintain multiple (materialized) views or constraints
  - expressed by queries $Q_1, Q_2, \ldots$
- When the database is updated
  - If we can determine (quickly) that query and update are independent
  - then can skip view/constraint maintenance
Query-update independence

Approach

- a static analysis
- that safely detects independence
- and takes schema into account
- for XQuery Update with all XPath axes
- fast enough to be useful as an optimization
Independence example

for $x$ in /c/a
return d[$x$]
Approach

- Exploits a **schema** $S$ that describes the input
- Statically calculate:
  - $c =$ **Copied nodes** of $Q$
  - $a =$ **Accessed Nodes** of $Q$
  - $u =$ **Updated Nodes** of $U$
- $c$, $a$, $u$ are sets of type names in $S$

Independence test

- We show that $Q$ and $I$ are independent modulo $S$ if:
  - $a$ is disjoint from $u$
    - that is, the update has no impact on accessed nodes
  - and $c/*$ is disjoint from $u$
    - that is, the update does not impact any copied nodes or their descendants

Analysis example

For $x$ in /c/a return $d[x]$

![Diagram](diagram1)

Analysis example

For $x$ in /c/a return $d[x]$

![Diagram](diagram2)
Analysis example

for $x$ in /c/a
return d[$x]
/
c
 a
b
 a
d
a

Analysis example

for $x$ in /c/a
return d[$x]
/
c
 a
b
 a
d
a

Analysis example

for $x$ in /c/a
return d[$x]
/
c
 a
b
 a
d
a

Analysis example

for $x$ in /c/a
return d[$x]
/
c
 a
b
 a
d
a

Copied

Copied
Analysis example

for $x$ in /c/a
return d[$x]
/
c
 a
a
 b
 a
d
a
/
c
 a
a
 foo
 a

Updated nodes disjoint

Analysis example II

for $x$ in /a
return d[$x]
a
b
 a
d
bar
50

Updated node avoids accessed & copied nodes
for $x$ in /a return $d[x]$

Updated node is a descendant of a copied node!
Correctness

- We can use type names for sets of accessed, copied, updated nodes
- Symbolically evaluate query/update over schema
- However, there is a complication:
  
  **Aliasing**

- How do we know that it is safe to use type names to refer to and change “parts” of schema?

Problem

- For example:

\[
S \rightarrow \text{doc}[T^*, U, T^*]
T \rightarrow t[A|B]
U \rightarrow t[B|C]
A \rightarrow a[]
B \rightarrow b[]
C \rightarrow c[]
\]
Problem

• For example:

\[
\begin{align*}
S & \rightarrow \text{doc}[T^*, U, T^*] \\
T & \rightarrow t[A|B] \\
U & \rightarrow t[B|C] \\
A & \rightarrow a[] \\
B & \rightarrow b[] \\
C & \rightarrow c[] \\
\end{align*}
\]

Solution:

Closure under Aliasing

• Keep track of which type names “alias”
  • or, can refer to the same node
  • currently, based on tree language overlap test
• Close sets a, c, u under aliasing
  • example: any T could be a U, and vice versa
• Note: This may be overkill
  • unnecessary (has no effect) for DTDs
  • probably unnecessary for unambiguous XML Schema

Summary

• Independence analysis can statically determine whether view needs to be maintained
• Subsequent work explores more precise static information:
  • path-based independence analysis ("destabilizers") - no schema required [Cheney & Benedikt VLDB 2010]
  • independence based on more precise type analysis [Bidoit-Tollu et al. VLDB 2012]
• Future work:
  • combine path- and schema-based approach?
  • what about XML views of relations?
  • can static analysis help with incremental maintenance?
Next week

• Provenance
  • Why and Where: A characterization of data provenance
  • Provenance management in curated databases
  • Annotated XML: queries and provenance