XML and Databases

Lecture 5
XML Validation using Automata

Sebastian Maneth
NICTA and UNSW

CSE@UNSW -- Semester 1, 2010

Outline

1. Recap: deterministic Reg Expr’s / Glushkov Automaton
2. Complexity of DTD validation
3. Beyond DTDs: XML Schema and RELAX NG
4. Static Methods, based on Tree Automata

Previous Lecture

XML type definition languages

want to specify a certain subset of XML doc’s = a "type" of XML documents

Remember
The specification/type definition should be simple, so that

→ a validator can be built automatically (and efficiently)
→ the validator runs efficient on any XML input

(similar demands as for a parser)

→ Type def. language must be SIMPLE!

(similarity: parser generators use EBNF or smaller subclasses: LL / LR)

O(n^3) parsing

XML Type Definition Languages

DTD  (Document Type Definition, W3C)

Originated from SGML. Now part of XML

→DTD may appear at the beginning of an XML document

XML Schema  (W3C)

Now at version 1.1

HUGE language, many built-in simple types

→Schemas themselves: written in XML

See the "Schema Primer" at http://www.w3.org/TR/xmlschema-0/

RELAX NG  (Oasis)

For tree structure definition, more powerful than Schemas/DTDs

XML Type Definition Languages

DTD  (Document Type Definition)

<!DOCTYPE root-element [ doctype declaration .. ]>

<!ELEMENT element-name content-model>

content-models

• EMPTY
• ANY
• (#PCDATA | elem-name_1 | ... | elem-name_n)*
• deterministic Reg Expr over element names

<!ATTLIST element-name attr-name attr-type attr-default ..>

Types: CDATA, (v1|..), ID, IDREFs

Defaults: #REQUIRED, #IMPLIED, "value", #FIXED

Most interesting / challenging aspect of DTDs

XML Type Definition Languages

DTD  (Document Type Definition)

<!DOCTYPE root-element [ doctype declaration .. ]>

<!ELEMENT element-name content-model>

content-models

• EMPTY
• ANY
• (#PCDATA | elem-name_1 | ... | elem-name_n)*
• deterministic Reg Expr over element names

<!ATTLIST element-name attr-name attr-type attr-default ..>

Types: CDATA, (v1|..), ID, IDREFs

Defaults: #REQUIRED, #IMPLIED, "value", #FIXED

Most interesting / challenging aspect of DTDs
Summary

In order to check whether a (large) document is valid with respect to a given DTD ("it validates")
you need to check if children lists match the given Reg Expr's.

This can be done efficiently, using finite-automata (FAs):

To check if a Reg Expr e is allowed in a DTD we have to construct a particular finite automaton: the Glushkov automaton.

Glu(e) must be deterministic.

Note: If Glu(e) is deterministic, then its size (of transitions) is linear in size(e)!

Question Can you explain why this is the case?

More Notes

(1) From a deterministic FA you cannot obtain a deterministic (= 1-unambiguous) regular expression!

Example: \( e = (a | b)^* a (a | b) \) → NO 1-unambiguous reg exp exists for e

Deterministic Automaton for e:
E.g., first nondeterministic FA, then determinize (subset construction)
CAVE: can cause exponential size blow-up!

Other important constructions on Finite Automata:

→ Union (easy)
→ Intersection (product construction)
→ Complementation

Question Size blow-up for these?

More Notes

(2) Glu(e) is closely related to
Thomson(e) [remove \( \varepsilon \)-transitions]
Berry/Sethi(e) [same]
Brzozowski(e)

To check if a Reg Expr e is allowed in a DTD we have to construct a particular finite automaton: the Glushkov automaton.

Glu(e) must be deterministic.

Note: If Glu(e) is deterministic, then its size (of transitions) is linear in size(e)!
Glushkov automaton Glu(e)

Each letter-position in the Reg Expr e becomes one state of Glu; plus, Glu has one extra begin state.

FIRST(e) = all possible begin positions of words matching e
e.g. FIRST(R (E | G) (EX)* ) = { R1 }

Glushkov’s automaton

- Character in RE = state in automaton
- One state for the beginning of the RE
- Transitions show which characters/positions can precede each other

R (E | G) (EX)*

Glushkov’s automaton

- Character in RE = state in automaton + one state for the beginning of the RE
- Transitions show which characters/positions can precede each other

\[ R \ (E \ | \ G) \ (EX)^* \]

\[ \begin{array}{c}
R \\
E \\
G \\
E \\
X \\
R...
\end{array} \]

\[ \text{FIRST}(e) \]

\[ \text{FOLLOW}(e, R1) = \]

\[ \text{FOLLOW}(e, E2) = \]

\[ \text{FOLLOW}(e, G3) = \]

Glushkov automaton \( G(e) \)

Each position in the Reg Expr \( e \) becomes one state of \( G \); plus, \( G \) has one extra begin state.

\[ \text{FIRST}(e) = \text{all possible begin positions of words matching } e \]

\[ \text{e.g. FIRST}(R \ (E | G) \ (EX)^*) = \{ R_1 \} \]

\[ \text{FOLLOW}(e, x) = \text{all possible positions following position } x \text{ in } e \]

\[ \text{e.g. FOLLOW}(R \ (E | G) \ (EX)^*, R1) = \{ E2, G3 \} \]

→ From state “R1,” add

  E-transition to E2

  G-transition to G3

Glushkov’s automaton

- Character in RE = state in automaton + one state for the beginning of the RE
- Transitions show which characters/positions can precede each other

\[ R \ (E \ | \ G) \ (EX)^* \]

\[ \begin{array}{c}
R \\
E \\
G \\
E \\
X \\
RE... \\
RG...
\end{array} \]

\[ \text{FOLLOW}(e, R1) = \]

\[ \text{FOLLOW}(e, E2) = \]

\[ \text{FOLLOW}(e, G3) = \]

Glushkov’s automaton

- Character in RE = state in automaton + one state for the beginning of the RE
- Transitions show which characters/positions can precede each other

\[ R \ (E \ | \ G) \ (EX)^* \]

\[ \begin{array}{c}
R \\
E \\
G \\
E \\
X \\
RE... \\
RG...
\end{array} \]

\[ \text{FOLLOW}(e, R1) = \]

\[ \text{FOLLOW}(e, E2) = \]

\[ \text{FOLLOW}(e, G3) = \]
Glushkov automaton $G(e)$

Each position in the Reg Expr $e$ becomes one state of $G$; plus, $G$ has one extra begin state.

$\text{FIRST}(e) =$ all possible begin positions of words matching $e$

e.g. $\text{FIRST}(R (E \mid G) (EX)^*) = \{ R_1 \}$

$\text{FOLLOW}(e, x) =$ all possible positions following position $x$ in $e$

e.g. $\text{FOLLOW}(R (E \mid G) (EX)^*, R_1) = \{ E_2, G_3 \}$

$\text{LAST}(e) =$ all possible end positions of words matching $e$

e.g. $\text{LAST}(R (E \mid G) (EX)^*) = \{ E_2, G_3, X_5 \}$

Is this automaton deterministic??

Another example

$(a^* \mid ba)^*$

This FA is deterministic.

Which of these is deterministic?

1. $(ab) \mid (ac)$
2. $a (b \mid c)$
3. $a (a \mid b)^*ac$
Glushkov automaton $G(e)$

Each position in the Reg Expr $e$ becomes one state of $G$; plus, $G$ has one extra begin state.

$\text{FIRST}(e) =$ all possible begin positions of words matching $e$
e.g. $\text{FIRST}(R(E|G)(EX)^*) = \{ R_1 \}$

$\text{FOLLOW}(e,x) =$ all possible positions following position $x$ in $e$

$\text{LAST}(e) =$ all possible end positions of words matching $e$

Naïve implementation: $O(m^3)$ time, where $m = \text{size}(e)$

(for each position: computing FOLLOW goes through every position at each step, needs to compute union $\rightarrow O(m^3)$)

$\text{FIRST}(e) =$ all possible begin positions of words matching $e$
e.g. $\text{FIRST}(R(E|G)(EX)^*) = \{ R_1 \}$

$\text{FOLLOW}(e,x) =$ all possible positions following position $x$ in $e$

$\text{LAST}(e) =$ all possible end positions of words matching $e$

Naïve implementation: $O(m^3)$ time, where $m = \text{size}(e)$

(for each position: computing FOLLOW goes through every position at each step, needs to compute union $\rightarrow O(m^3)$)

Not really needed. Can be improved to $O(m^2)$

Note

If $G(e)$ is deterministic, then its size (# transitions) is quadratic in $\text{size}(e)$!

Linear in $\text{size}(e) \times \#\text{letters}(e)$, if $G(e)$ is deterministic.

$\rightarrow O(\text{size}(e) \times \#\text{letters}(e))$

Can this be improved? $\rightarrow$ E.g., to $O(\text{size}(e) \times \log(\#\text{letters}(e)))$?

Are there known lower bounds?

$O(\text{size}(e) + \text{size}(G(e)))$

Not really needed. Can be improved to $O(m^2)$

Summary

Deterministic (1-unambiguous) content models give rise to efficient matching algorithms.

(they avoid $O(mn)$ or $O(2^m + n)$ complexities)

Disadvantages

$\rightarrow$ Hard to know whether given reg expr is OK (deterministic)

$\rightarrow$ Det. reg exprs are NOT closed under union. (not so nice.)

Question

Can you see why?

Maybe, can find det reg exprs, such that their union equals $(a | b)^* a (a | b)$?
Now that we know how to check all the different content-models (in particular det. Reg Expr's) how to build a full validator for a given DTD?

\[
\begin{align*}
&\text{elem-name}_1 \rightarrow \text{RegExpr}_1 \\
&\text{elem-name}_2 \rightarrow \text{RegExpr}_2 \\
&\text{elem-name}_k \rightarrow \text{RegExpr}_k
\end{align*}
\]

### The Validation Problem

Given a DTD \( T \) and a document \( D \), is \( D \) valid wrt \( T \)?

**Top-Down Implementation**
- At element node \( w \) label \( \text{elem-name}_i \), run automaton \( A_i \)
- Check attribute constraints
- Check ID/IDREF constraints

### Total Running Time

Linear in the sum of \( \text{size}(DTD) \times \text{#letters}(DTD) \) and size of document: \( O(\text{size}(D) \times \text{#letters}(D) + \text{size}(T)) \)

---

### Beyond DTDs

Often, the expressive power of DTDs is not sufficient.

**Problem** Each element name has precisely one content-model in a DTD. Would like to distinguish, depending on the context (parent).

- Dealer
  - Used
    - Car
      - Model
      - Year
  - New
    - Car
      - Model

**Specialized DTDs**

\[
\begin{align*}
&\text{dealer} \rightarrow \text{used, new} \\
&\text{used} \rightarrow (\text{Car}_{\text{used}})^* \\
&\text{new} \rightarrow (\text{Car}_{\text{new}})^* \\
&\text{car}_{\text{used}} \rightarrow \text{model, year} \\
&\text{car}_{\text{new}} \rightarrow \text{model}
\end{align*}
\]

**New notation. Use capitalized TYPE Names**

\[
\begin{align*}
&\text{Dealer} \rightarrow \text{used, new} \\
&\text{Used} \rightarrow (\text{Car}_{\text{used}})^* \\
&\text{New} \rightarrow (\text{Car}_{\text{new}})^* \\
&\text{Car}_{\text{used}} \rightarrow \text{Model, Year} \\
&\text{Car}_{\text{new}} \rightarrow \text{Model}
\end{align*}
\]

---

### DTDs have the "label-guarded subtree exchange" property:

11, 12 trees in a DTD language \( T \)
- \( v_1 \) node in 11, labeled "lab"
- \( v_2 \) node in 12, labeled "lab"

Trees obtained by exchanging the subtrees rooted at \( v_1 \) and \( v_2 \) are also in \( T \)

**aka “local” content model only depends on label of parent**

\[
\begin{align*}
&\text{t}_1, \text{t}_2 \text{ trees in a DTD language } T \\
&\text{v}_1 \text{ node in t}_1 \text{, labeled “lab”} \\
&\text{v}_2 \text{ node in t}_2 \text{, labeled “lab”}
\end{align*}
\]

**Trees obtained by exchanging the subtrees rooted at \( v_1 \) and \( v_2 \) are also in \( T \)**

---

**Beyond DTDs**

Often, the expressive power of DTDs is not sufficient.

**Problem** Each element name has precisely one content-model in a DTD. Would like to distinguish, depending on the context (parent).

- Dealer
  - Used
    - Car
      - Model
      - Year
  - New
    - Car
      - Model

**Specialized DTDs**

\[
\begin{align*}
&\text{dealer} \rightarrow \text{used, new} \\
&\text{used} \rightarrow (\text{Car}_{\text{used}})^* \\
&\text{new} \rightarrow (\text{Car}_{\text{new}})^* \\
&\text{car}_{\text{used}} \rightarrow \text{model, year} \\
&\text{car}_{\text{new}} \rightarrow \text{model}
\end{align*}
\]

**New notation. Use capitalized TYPE Names**

\[
\begin{align*}
&\text{Dealer} \rightarrow \text{used, new} \\
&\text{Used} \rightarrow (\text{Car}_{\text{used}})^* \\
&\text{New} \rightarrow (\text{Car}_{\text{new}})^* \\
&\text{Car}_{\text{used}} \rightarrow \text{Model, Year} \\
&\text{Car}_{\text{new}} \rightarrow \text{Model}
\end{align*}
\]
A grammar $G$ is *local* if for any label $\text{RegExpr}_1$, label $\text{RegExpr}_2$ present in $G$ it holds that $\text{RegExpr}_1 = \text{RegExpr}_2$.

**Classes of Grammars**

- **local** no competing TYPE names! (DTDs)
- **single-type** TYPE names in the same content model do not compete! (XML Schema’s)
- **regular** no restriction. (RELAX NG)

**Question**

Are there single-type grammars (XML Schemas) which cannot be expressed by local grammars (DTDs). **YES!**

**Question**

Are there single-type grammars (XML Schemas) which cannot be expressed by local grammars (DTDs). **YES!**

**Classes of Grammars**

- **local** no competing TYPE names! (DTDs)
- **single-type** TYPE names in the same content model do not compete! (XML Schema’s)
- **regular** no restriction. (RELAX NG)

Through the use of TYPE Names (nonterminals / states) you can distinguish deep context!
New notation. Use capitalized TYPE Names

Through the use of TYPE Names (nonterminals / states) you can distinguish deep context!

Can we model context that is far away from the specialized node?

Sure!

Can we model context that is far away from the specialized node?

Sure!

Questions

Given two DTDs D1 and D2 can we check if

+ all documents valid for D1 are also valid for D2? (DTD inclusion problem)
+ D1 and D2 describe the same set of documents? (DTD equality problem)

Given a Relax NG grammar G, can we check if

+ there exists any document that is valid for G? (emptiness problem)
+ there is a document valid for G and valid for G2? (intersection & emptiness)
Questions

Given two DTDs D1 and D2 can we check if:
- All documents valid for D1 are also valid for D2? (DTD inclusion problem)
- D1 and D2 describe the same set of documents? (DTD equality problem)

Given a Relax NG grammar G, can we check if:
- There exists any document that is valid for G? (emptiness problem)
- There is a document valid for G and valid for G2? (intersection & emptiness)

If we can do it for regular tree grammars, then also works for single-type/local!

All of the checks can be done automatically, for regular tree grammars! Equivalent to tree automata

Tree Automata: very powerful framework,
- Have all the good properties of string automata!
- Yet, they are more expressive!

Note

String automata are not sufficient to check DTDs / Schemas! Even if we only consider well-bracketed strings!

Example 1
- c → c[ a, c, b ]
- a → empty
- b → empty
- c → empty

Example 2
- a → a[ c, a ]
- a → a[ a, b ]
- a / b / c → empty

Finite-state automata are important:
- Think you are in a maze, with only fixed memory and you can only read the maze (cannot mark anything).

Model by finite automaton. In state q1, (to [N|S|E|W],)
- q1, (to [N|S|E|W],)

• Can an automation search the maze?

No!! Need markers (“pebbles”). How many? 5? 2?

Finite-state automata are important:
- In our context, e.g., for KMP (efficient string matching) [Knuth/Morris/Pratt] generalization using automata. Used, e.g., in grep
- Compression
- Static analysis of schemas & queries (= “everything you can do before running over the actual data”)

4. Static Methods, based on Tree Automata

Person → MPerson | FPerson
MPerson → person [Name, gender[Male], FSpouse?] FPerson → person [Name, gender[Female], FSpouse?]?

Regular Tree Grammar
Rules of the form TypeName → Tree
Leaves may be labeled by TypeNames

<table>
<thead>
<tr>
<th>State</th>
<th>Element-name</th>
<th>state1</th>
<th>state2</th>
</tr>
</thead>
<tbody>
<tr>
<td>person</td>
<td>Name</td>
<td>gender</td>
<td>FSpouse</td>
</tr>
<tr>
<td></td>
<td>Name</td>
<td>gender</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Male</td>
<td></td>
<td>Male</td>
</tr>
</tbody>
</table>

Alternatively, regular tree languages are defined by Tree Automata. Conventionally, defined for binary/rankless trees.
4. Static Methods, based on Tree Automata

Given grammars $D_1$ and $D_2$ can we check if
- all documents valid for $D_1$ are also valid for $D_2$? (inclusion problem)
- $D_1$ and $D_2$ describe the same set of documents? (equality problem)
- does there exists any document that is valid for $D_1$? (emptiness problem)
- there is a document valid for $D_1$ and valid for $D_2$? (intersection & emptiness)

ALL these checks are possible for regular tree grammars!!

(1) use binary tree encodings
(2) translate XML Type Definition to a Tree Grammar (easy)

Alternatively, regular tree languages are defined by Tree Automata.

state, element-name $\rightarrow$ state1, state2 conventionally, defined for binary/ranked trees.

For example:
documents $d_1, d_2, \ldots, d_n$ are valid for your schema “Small_xhtml”.
Are they also valid for schema XHTML?
- Check inclusion problem for Small_html and XHTML!

The checks above give rise to very powerful optimization procedures for XML Databases!

(1) use binary tree encodings
(2) translate XML Type Definition to a Tree Grammar (easy)

The checks above give rise to very powerful optimization procedures for XML Databases!
Automata on words

Expressiveness
deterministic = nondeterministic
left-to-right = right-to-left

word is accepted by the automaton

Automata on trees

1. bottom-up

\[
\text{LABEL}(\text{state1}, \text{state2}) \Rightarrow \text{state}
\]

\[
\begin{align*}
0() & \Rightarrow F \\
1() & \Rightarrow T \\
\text{OR}(F, F) & \Rightarrow F \\
\text{OR}(F, T) & \Rightarrow T \\
\ldots
\end{align*}
\]

Accepting States = \{T\}

This automaton is deterministic.

Automata on words

Expressiveness
deterministic = nondeterministic
left-to-right = right-to-left

word is accepted by the automaton

Automata on trees

1. bottom-up

\[
\text{LABEL}(\text{state1}, \text{state2}) \Rightarrow \text{state}
\]

\[
\begin{align*}
0() & \Rightarrow F \\
1() & \Rightarrow T \\
\text{OR}(F, F) & \Rightarrow F \\
\text{OR}(F, T) & \Rightarrow T \\
\ldots
\end{align*}
\]

\[
\begin{align*}
\text{AND}(F, T) & \Rightarrow F \\
\text{AND}(T, F) & \Rightarrow T
\end{align*}
\]

Accepting States = \{T\}

This automaton is deterministic.
Automata on trees

**Question**

How much memory do you need exactly, to run such a bottom-up tree automaton?

For every **nondeterministic** bottom-up tree automaton there is an equivalent **deterministic** bottom-up tree automaton.

Again, the construction can cause exponential size blow-up.

2. **top-down**

```
state, LABEL -> (state1, state2)
```

must contain a S-leaf

a,b = binary node labels
e,$ = leaf node labels

top-most a-node on the left-most path
must have a right-subtree which contains a S-node.

```
begin, a  -> (any, find$)
begin, b  -> (begin, any)
find$, ab  -> { (find$, any), (any, find$) }
find$, $   -> ACC
find$, e   -> REJ
any, a/b   -> { (any,any), (any, $) }
any, $/e   -> ACC
```

nondeterministic

```
accept tree if there exists an accepting run
```

→ Yes! you can...☺

For every **nondeterministic** bottom-up tree automaton there is an equivalent **deterministic** bottom-up tree automaton, and → there is an equivalent **nondeterministic** top-down tree automaton.

```
2. **top-down**

state, LABEL -> (state1, state2)
```

must contain a S-leaf

a,b = binary node labels
e,$ = leaf node labels

top-most a-node on the left-most path
must have a right-subtree which contains a S-node.

```
begin, a  -> (any, find$)
begin, b  -> (begin, any)
find$, ab  -> { (find$, any), (any, find$) }
find$, $   -> ACC
find$, e   -> REJ
any, a/b   -> { (any,any), (any, $) }
any, $/e   -> ACC
```

nondeterministic

```
accept tree if there exists an accepting run
```

→ Is there an equivalent **deterministic top-down** automaton??

For every **nondeterministic** bottom-up tree automaton there is an equivalent **deterministic** bottom-up tree automaton, and → there is an equivalent **nondeterministic top-down** tree automaton.

**Question**

Can you find an equivalent bottom-up automaton for this example?

```
2. **top-down**

state, LABEL -> (state1, state2)
```

must contain a S-leaf

a,b = binary node labels
e,$ = leaf node labels

top-most a-node on the left-most path
must have a right-subtree which contains a S-node.

```
begin, a  -> (any, find$)
begin, b  -> (begin, any)
find$, ab  -> { (find$, any), (any, find$) }
find$, $   -> ACC
find$, e   -> REJ
any, a/b   -> { (any,any), (any, $) }
any, $/e   -> ACC
```

nondeterministic

```
accept tree if there exists an accepting run
```

→ Yes! you can...☺
For every nondeterministic bottom-up tree automaton
\[ \Rightarrow \] there is an equivalent deterministic bottom-up tree automaton, and
\[ \Rightarrow \] there is an equivalent nondeterministic top-down tree automaton.

**Question**
Is there an equivalent deterministic top-down automaton??

\[ \Rightarrow \text{NO! } \] — not a good model 😞

**Why?**
This set of two trees can NOT be recognized by any deterministic top-down tree automaton!

<table>
<thead>
<tr>
<th>First</th>
<th>Last</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
</tbody>
</table>

**Questions**
What about local tree languages (defined by DTDs)?
Can they be accepted by deterministic top-down automata?

What about single-type tree languages (defined by XML Schema’s)?
Can they be accepted by deterministic top-down automata?

**Yes!**
Hence, there is no DTD / Schema for \{ name[first,last], name[last,first] \}

In total, given a DTD D, we can build one deterministic top-down automaton of size \( O(\text{size}(D) \times \#\text{letters}(D)) \)

Thus, matching time is inside \( O(m^2 + n) \)

For every nondeterministic bottom-up tree automaton
\[ \Rightarrow \] there is an equivalent deterministic bottom-up tree automaton, and
\[ \Rightarrow \] there is an equivalent nondeterministic top-down tree automaton.

**Question**
Is there an equivalent deterministic top-down automaton??

\[ \Rightarrow \text{NO! } \] — not a good model 😞

Nevertheless, XML Schemas are a subclass of deterministic top-down automata.

**Questions**
What is the reasoning behind this?
Similarity to restriction to deterministic regular expressions?

Recall: Deterministic (1-unambiguous) content models give rise to efficient matching algorithms, (they avoid \( O(m) \), \( O(2^m + n) \) complexities)

In fact, YOU have already produced minimal unique equivalent ones!

Equivalence is decidable

The minimal DAG of a tree \( t \) can be seen as the minimal unique deterministic BU tree automaton that only accepts the tree \( t \).
For every deterministic bottom-up tree automaton there exists a minimal unique equivalent one!

Equivalence is decidable

In fact, YOU have already produced minimal bottom-up tree automata!

The minimal DAG of a tree t can be seen as the minimal unique tree automaton that only accepts the tree t.

Question

Given deterministic BU tree automaton.

How expensive (complexity) to find unique minimal one?

As for DFAs: merge equivalent states

Typically: quadratic running time. → Hopcroft’s algorithm, $O(n \log n)$

More generally, once we have partitioned on a block and an input symbol, we need never partition on that block and input symbol again until the block is split and then we need only partition on one of the two subblocks. Since the time needed to partition on a block is proportional to the transitions into the block and since we can always select the half with fewer transitions, the total number of steps in the algorithm is bounded by $n \log n$.


Tree Automata are a very useful concept in CS!

Heavily used in verification

"Derive a property of a complex object from the properties of its constituents..."

Do all graphs / chip-layouts produced in this way, have property P?

Use the Hierarchical construction history of an object, in order to work on a "parse" tree instead of a complex graph. From there, use tree automata.

XML

Tree Automata play crucial rule for

Efficient validators against XML Types

Optimizations: if doc1 is of TYPE1, then no need to validate against TYPE2, if we know TYPE2 included in TYPE1

- if only "slightly different" then only need to validate "there"
- incremental validation against updates
- etc, etc.

Efficient query evaluators, use richer automata which can select nodes and produce query answers

Optimizations: if answer of QUERY1 is in cache, then no need to evaluate QUERY2, if "included" in QUERY1.

- if every possible answer set to QUERY1 (of TYPE X) is EMPTY, then no need to evaluate on the real data!

XML Type Checking for Programming Languages

The Future

In 5-10 years from now:

You can write a function in Programming Language X

Compiler (XML Type Checker) will complain, if your function does not compute documents of TYPE2.

- If no complaint, then guaranteed: ALL outputs are ALWAYS of correct type!!

Experimental PL’s

In this direction:

→ CDuce

→ XDuce

The Future

In 5-10 years from now:

You can write a function in Programming Language X

Compiler (XML Type Checker) will complain, if your function does not compute documents of TYPE2.

- If no complaint, then correct type guaranteed.

Compilers will have to be able to give static guarantees about input/output behaviour of program!
END
Lecture 5