

XML and Types

Outline:

- eXtensible Markup Language
 - XML Data Model
 - XML Query Languages
- XML Schema (Type) Formalisms
 - A Formal Language Perspective
 - DTD
 - XML Schema
 - XDuce
 - {Document Validation Techniques}?
 - {Inferring Schemas}?
- XML Typechecking
 - Applications
 - Type Inference
 - Type Checking

1. XML

- HTML4.0 \in XML \subset SGML (ISO 8879:1986)
- Extensible Markup Language (XML) 1.0 (2nd Edition)
W3C Recommendation Oct 2000, www.w3.org/TR/REC-xml
- Standard for data exchange, but now also considered a standard representation format for documents (and data)

Example 1

```
<literature>
  <book isbn="1-55860-190" price="110" curr="Euro">
    <title>Transaction Processing</title>
    <authors>
      <author>
        <fname>Jim</fname><lname>Gray</lname>
      </author>
      <author>
        <fname>Andreas</fname><lname>Reuter</lname>
      </author>
    </authors>
  </book>
  <book ...>
</literature>
```

Some terminology:

- tags: book, title, author, ...
- start tags: <book>, <title>, ...
- elements: <book>... </book>, <title>... </title>, ...
elements are properly nested (*well-formedness of a document*)

1.1 XML Data Model

- Document Object Model (DOM); defines API to process and modify XML document (www.w3.org/DOM/)
- XML Information Set (www.w3.org/TR/xml-infoset/)
- XQuery 1.0 and XPath 2.0 Data Model

XQuery 1.0 and XPath 2.0 Data Model

- XML documents are considered trees with different node types
- Defines (1) the information contained in the input to an XSLT or XQuery processor, and (2) all permissible values of expressions in the XSLT, XQuery, and XPath languages.
- Uses functional-style language for model notation
- Node ::= DocNode | ElemNode | ValueNode | AttrNode | NSNode | PINode | CommentNode | RefNode
- Element node:
 $\text{elemNode} : (\text{QNameValue}, \{\text{AttrNode}\}, [\text{ElemNode} \mid \text{ValueNode}]) \rightarrow \text{ElemNode}$
 $\text{QNameValue} \hat{=} \text{tag name}; \{\dots\} \hat{=} \text{set of}; [\dots] \hat{=} \text{list of};$
- Attribute node:
 $\text{attrNode} : (\text{QNameValue}, \text{ValueNode}) \rightarrow \text{AttrNode}$
- Value node:
 $\text{ValueNode} = \text{StringValue} \mid \text{FloatValue} \mid \text{BoolValue} \mid \dots$
 $\text{stringValue} : \text{string} \rightarrow \text{StringValue}, \text{etc}$

Example 2

```
book1 = elemNode(book, {price1, currency1}, authors1)

authors1 = elemNode(authors, [title1, author1, author2])
author1  = elemNode(author, [fname1, fname2])

price1   = attrNode(price, stringValue("110"))
stringValue10 = valueNode(stringValue("110"))

title1   = elemNode(title, stringValue("Transaction ..."))
stringValue40 = valueNode(stringValue("Transaction ..."))
...
```

Common view in most approaches to XML document management and processing:

Definition 3 (XML Document)

Let \mathbf{E} be a finite set of elements (types), \mathbf{A} a finite set of attribute names ($\mathbf{E} \cap \mathbf{A} = \emptyset$), and \mathbf{S} a set of text strings. A document D is an ordered, node-labeled tree $(V, \text{root}, \text{elem}, \text{attr})$ with nodes V such that

- elem is a function mapping nodes to element names and child vertices, i.e., $\text{elem} : V \rightarrow \mathbf{E} \times \text{list}(\mathbf{S} \cup V)$,
- attr is a partial mapping from vertices and attribute name pairs to sets of atomic values, i.e., $\text{attr} : V \times \mathbf{A} \rightarrow \mathcal{P}^{\mathbf{S}}$,
- $\text{root} \in V$ is a distinguished element called document root.

1.2 XML Query Languages

- Queries specify (regular) path and/or tree patterns to be matched against XML document (or collection of documents)
Query result is typically (list of) document fragments (from the original document(s) or a construction thereof), a list of values, or a boolean value.
- Some “older” languages: LOREL, UnQL, XQL, XML-GL, XML-QL (first declarative QL for XML)
- XPath and XQuery

XPath

- Basis for W3C standards such as XML Link (XLink), XML Pointer (XPointer), XSL Transformation (XSLT), XQuery
- XML Path Language (XPath) 2.0, W3C Working Draft, 5/2/03, www.w3.org/TR/xpath20/
- XPath expression specify tree patterns to be matched against XML document; an expression can “navigate” along 13 axes in the document tree
ancestor[-or-self], attribute, child, descendant[-or-self], following[-sibling], preceding[-sibling], namespace, parent, self
- An XPath expression describes a relation between a *context node* and a node in the *answer list*; relationship is established through *location steps*.

- Abbreviated XPath syntax: Let $e \in \mathbf{E}$ be an element name, $a \in \mathbf{A}$ an attribute name, and $s \in \mathbf{S}$.

Regular paths := $p_1 \mid p_1|p_2 \mid /p_1 \mid //p_1 \mid .. \mid p_1/p_2 \mid p_1//p_2 \mid p_1[q] \mid e \mid * \mid @a \mid @* \mid text() \mid node() \mid$
 ancestor-of-self \mid descendant-of-self

Qualifier := $q_1 \text{ and } q_2 \mid q_1 \text{ or } q_2 \mid \text{not } q_1 \mid (q_1) \mid p \mid p = s \mid p_1 = p_2$

Example 4

bib	match bib element
*	match any element
/	match root element
/bib	match bib element under root
bib/paper	match paper directly under bib (child)
bib//paper	match paper anywhere under bib (descendant)
//paper	match paper at any depth
@price	match price attribute
bib/*[./author/lname="Smith"]/section[1]//para[./figure[3]]	

XQuery

- Combines features from many earlier XML query languages
- One of the most comprehensive specs by W3C (syntax, formal semantics, data model, requirements, use cases, functions and operators, . . .); (www.w3.org/XML/Query)
- Core structure are FLWOR (“Flower”) Expressions
FOR . . . LET . . . FOR . . . LET . . .
WHERE
ORDER
RETURN

Example 5 *List all articles that appeared before 1996, including and ordered by their first author and title.*

```
FOR $a IN document("bib.xml")//article
WHERE $a/year < 1996
ORDER BY (author[1], title DESCENDING)
RETURN
  <early_article>
    {first_a:{$a/authors/author[1]/text()}</first_a>
      {$a/title}
    }
  </early_article>
```

(or just a simple XPath expression):

select paragraphs of articles published 2/15/02

```
document("news.xml")//article[./date="2/15/02"]//para
```

FOR \$x IN expression binds \$x to each element in the list expression

LET \$x IN expression binds \$x to entire list expression

```
<large_publishers>
  FOR $p IN distinct(document("bib.xml")//publisher)
  LET $b := document("bib.xml")//book/[publisher=$p]
  WHERE count($b) > 1000
  RETURN $p
</large_publishers>
```

2. XML Schema Formalisms

2.1 The Role of Schemas (Types) for XML

- In relational or OO/OR databases, schemas are defined before the data and are strictly enforced
For XML and semi-structured data in general, schemas can be defined (or derived) after the data and may not be enforced
 - Why is a schema important?
 - describe document content to users and applications
 - facilitate integrating data from several resources
 - optimize query evaluation
 - optimization of storage and access structures
 - Schema can be associated with document (collection) or query
 - Typical questions: does document (or query result) conform to a given schema? To what extent does a document conform to the schema? Are two schemas equivalent or is there a subsumption relationship?
 - Lower bound schema: what document structure components are required?
Upper bound schema: . . . what structures are allowed.
 - There are several schema proposals for XML: DTD (part of the XML standard), XML Schema, XDuCE, . . .
- ☞ many typing issues; but first lets have some formal basis to talk about schemas and types.

2.2 Some Formal Language Theory

Almost all schema proposals for XML are grammar-based. In general, an XML document is said to be *valid* (i.e., it *conforms* to the schema) if it is a parse tree for the grammar underlying the schema. XML documents are considered trees, thus

Definition 6 (Regular Tree Grammar)

A regular tree grammar RTG is a 4-tuple (N, T, S, P) where

- N is a finite set of non-terminal symbols,
- T is a finite set of terminal symbols,
- $S \subseteq N$ is a set of start symbols, and
- P is a finite set of production rules of the form $p \rightarrow ar$, where $p \in N$, $a \in T$ and r is a regular expression over N (r is called the *content model* of p).

Example 7 The following grammar $G = (N, T, P, S)$ is a regular tree grammar:

$$N = \{Doc, Para1, Para2, Data\}$$

$$T = \{doc, para, data\}$$

$$S = \{Doc\}$$

$$P = \{Doc \rightarrow doc\{Para1, Para2^*\}, \\ Para1 \rightarrow para(Data), Para2 \rightarrow para(Data), \\ Data \rightarrow data \epsilon\}$$

☞ derivation tree

Definition 8 (Interpretation)

An *interpretation* I of a tree t against a RTG G is a mapping from each node e in t to a non-terminal, denoted $I(e)$, s.t.

- $I(e_{root})$ is a start symbol and e_{root} is the root of t ,
- for each node e and its children e_0, \dots, e_k , there is a production rule $p \rightarrow ar$ s.t.
 - (1) $I(e) = p$,
 - (2) the terminal symbol (label) of e is a , and
 - (3) $I(e_0), \dots, I(e_k)$ matches r .

Definition 9 (Generation) A tree t is *generated* from a RTG G if there is an interpretation of t against G .

Definition 10 (Regular Tree Language)

A tree language is a *regular tree language* if it is generated by a regular tree grammar.

Typically, competing non-terminals make validation hard. Two non-terminals A and B are said to compete with each other if

- one prod rule has A in its lhs, and one prod rule has B in its lhs, and
- these two prod rules share the same terminal symbol in the rhs.

Example 11

$$N = \{Book, Author1, Son, Article, Author2, Daughter\}$$

$$T = \{book, author, son, daughter\}, S = \{Book, Article\}$$

$$P = \{Book \rightarrow book(Author1), Author1 \rightarrow author(Son), \\ Son \rightarrow son \epsilon, Article \rightarrow article(Author2), Author2 \rightarrow \\ author(Daughter), Daughter \rightarrow daughter \epsilon\}$$

Definition 12 (Local Tree Grammar)

A *local tree grammar* (LTG) is a regular tree grammar without competing non-terminals.

A less restricted class only prohibits competition of non-terminals within a single content model.

Definition 13 (Single-Type Tree Grammar)

A regular tree grammar is a *single-type tree grammar* if

- for each prod rule, non-terminals in its content model do not compete with each other, and
- start symbols do not compete with each other

Example 11 represents a single-type grammar. Single-type grammars are strictly more expressive than local tree grammars

2.3 Document Type Definition

- Part of the original XML specification
- Important for data exchange, if document comes with DTD
- A DTD basically specifies admissible elements, element nesting, and element attributes in form of an extended BNF.
- DTD is a local tree grammar; any DTD can be expressed as LTR, but not every LTR as DTD; closed under intersection but not under union and difference
- DTD can be considered as an extended content free grammar (regular expression in rhs); recall that CFG is not closed under intersection but under union.
- A DTD can have *recursive elements*!
- DTD is not in XML syntax.

```
<!DOCTYPE auctions [  
<!ELEMENT auctions (auction*)>  
<!ATTLIST auctions aid ID #IMPLIED>  
<!ELEMENT person (name, email, phone?)>  
<!ELEMENT item (name, descr?)>  
<!ELEMENT auction (seller, buyer, item, price, payment, anno?)>  
<!ELEMENT buyer (person)>  
<!ELEMENT seller (person)>  
<!ELEMENT payment (creditcard | paypal | moneyorder)>  
<!ELEMENT creditcard (cardtype,accountnum)>  
<!ELEMENT paypal (accountnum)>  
<!ELEMENT moneyorder EMPTY>  
<!ELEMENT annotation (author, happiness)>  
<!ELEMENT author person>]>
```

Attribute types: CDATA, ID, IDREF(S), enumeration, . . . , but no data types can be specified (everything is a string!); attributes can be optional, required, or have a default value.

Formally, for an element name $e \in E$, the content model for e is defined by a mapping $E \rightarrow \alpha$, where the content model α is an expression of the form $e' \mid \alpha_1\alpha_2 \mid \alpha_1, \alpha_2 \mid \alpha_1? \mid \alpha_1* \mid \alpha_1+ \mid (\alpha_1) \mid \text{EMPTY} \mid \text{pcdata}$, where $e' \in E, e \neq e'$.

2.3 XML Schema

- Generalizes DTD; uses XML syntax
- Simple types, complex types; local, global types
- XML Schema represents a single-type grammar (same closure properties as local tree grammars)
- Main features: complex type defs, anonymous type defs, group defs, subtyping by extension and restriction, abstract type defs, constraints (primary and foreign keys, unique)

```
<xsd:element name="article" type="articletype"/>
<xsd:complexType name="article">
  <xsd:sequence>
    <xsd:choice> <xsd:element name="journal"/>
                  <xsd:element name="conference"/>
    </xsd:choice>
    <xsd:element name="title" type="xsd:string"/>
    <xsd:element name="year"/>
    <xsd:element name="author" minOccurs="0"/>
  </xsd:sequence>
</xsd:element>

<!ELEMENT article ((journal|conference), title, year author*) >
```

Local versus global types (allow reuse):

```
<xsd:element name="book">
  definition of book's type
</xsd:element>

<xsd:element name="book" type="booktype"/>
<xsd:complexType name="booktype">
  definition of type booktype
</xsd:complexType>
```

Regular expressions are used in element-type-element alterations:
document root has a complex type, which is a regular expression of elements, each of which has its (complex) type,

Document leafs have simple types (standard data types)

Regular expressions:

```
<xsd:sequence> B M W </sequence> B M W
<xsd:choice> B M W </sequence> B | M | W
<xsd:group> B M W </sequence> (B M W)
<xsd:element name="Authors"
  minOccurs="0" maxOccurs="unbounded"> ...</...> (...)*
<xsd:element name="Authors"
  minOccurs="0" maxOccurs="1"> ..</...> (...)?
```

Types by extension (inheritance):

```
<xsd:complexType name="Address"> <xsd:sequence>
  <xsd:element name="name"/>
  <xsd:element name="street"/>
  <xsd:element name="city"/>
</xsd:sequence> </xsd:complexType>

<xsd:complexType name="USAddress">
<xsd:complexContent>
  <xsd:extension base="Address"/>
  <xsd:sequence> <xsd:element name="zipcode"/></xsd:sequence>
</xsd:extension>
</xsd:complexType>
```

Many more features: simple data types, each with up to 15 facets (additional properties restricting simple types); list types, union types, . . .

2.4 XDuce

- XDuce is much like a functional programming language where types are regular expressions
- For ranked alphabets, tail recursive XDuce types are equivalent to regular tree languages (recall that regular tree languages are closed under union, intersection, and difference!)

Example 14

```
<!ELEMENT bib ((book | paper)*)>
<!ELEMENT book (title, author*, year, publisher?)>
<!ELEMENT title PCDATA>
```

```
type Bib = bib[(Book|Paper)*]
type Book = book[Title, Author*, Year, Publisher]
type Title = title[String]
```

- In XDuce, types are first class citizens, element names are second class citizens; consistent with notion of regular expressions and tree automata (type = state)
- XDuce allows for subsumption of types ($T1 < T2$ if set defined by $T1$ is a subset of that defined by $T2$; e.g., $A, B < A, B, C?$, and $A, B, C < A, B, C?$, and $A,A,A < A^*$)
- Regular tree language \rightarrow Bottom-Up Tree automata

Definition 15 (Bottom-Up Tree Automata)

A *bottom-up tree automata* is a 4-tuple (E, Q, δ, Q_F) , where

- E is a ranked alphabet $E = \{E_1 \cup \dots \cup E_k\}$
- Q is a set of states
- δ are transition rules with $\delta : \bigcup_{i=0, \dots, k} E_i \times Q^i \rightarrow Q$, and
- Q_F are final states

Computation on a tree t

- for each node $t = a[t_1, \dots, t_i]$ ($a \in E_i$ with rank i), if the roots of t_1, \dots, t_i are labeled with states q_1, \dots, q_i and $q \in \delta(a, q_1, \dots, q_i)$, then assign state q to t .
- if the root is labeled with $q' \in Q_F$, then accept.

If A is a non-deterministic bottom-up tree automata, then there exists an equivalent deterministic one; also, if T1, T2 are regular tree languages, then it is decidable whether $T1 \subseteq T2$.

In general, for all schema approaches, quite a lot of automata theory is involved, see, e.g.,

Ferenc Gecseg and Magnus Steinby: *Tree Automata*. Akademiai Kiado, Budapest 1984

Hubert Comon et al.: *Tree Automata techniques and Applications (TATA)*, www.grappa.univ-lille3.fr/tata/

2.5 Document Validation Techniques

- Problem: Does a given XML document conform to a given type?
- ...

2.6 Schema Inference

- Problem: Given a collection of XML document. What is the schema (type) underlying the collection?
- ...

3. XML Typechecking

- When an XML document conforms to a given type (DTD, XML Schema) it is called valid; verified by *validating parser*
- Can be done efficiently for documents for which a type is known or specified (DOM, SAX, . . .)
- Often documents (fragments thereof or constructed from fragments) are dynamically generated by some program (e.g., query or XSL transformation)
- *Typechecking problem*: Are all documents generated by the program valid? Thus, the problem is a function of the language in which the program is expressed (e.g., XQuery, XSLT)

In the following:

- Finite alphabet Σ of tag names, attribute names, and atomic type names; \mathcal{T}_Σ set of ordered trees where each node is labeled with element from Σ
- A type is a subset of \mathcal{T}_Σ that is a regular tree language

```
TYPE Catalog = ELEMENT catalog(Products)
TYPE Products = (ELEMENT product(Product))*
TYPE Product = (ATTRIBUTE name(String)?,
                (ELEMENT mfr-price(Int) | ELEMENT sale-price(Int))* ,
                (ELEMENT color(String))*)
```

- Above definition of types is more powerful than XML Schema
- Important property for typechecking: given two types τ_1, τ_2 , check whether $\tau_1 \subseteq \tau_2$. EXPTIME in general, but PTIME when τ_2 corresponds to deterministic tree automata.

The XML Typechecking Problem

- validation problem: given $t \in \mathcal{T}_\Sigma$, type τ , is $t \in \tau$?
- typechecking problem: given program P defining a function $P: \mathcal{D} \rightarrow \mathcal{T}_\Sigma$, and a type $\tau \subseteq \mathcal{T}_\Sigma$, does $\forall x \in \mathcal{D}, P(x) \in \tau$ hold?

Corresponding type-checker analyzes P and XML output type, and decides whether all output documents are valid. If not, one would like to know where in the program the typechecking failed.

- XQuery allows users to annotate expressions with expected result type.
- Typechecking is not always possible; one has to settle with incomplete type-checkers that return false negatives.
- type inference problem: given a program P , what is the type $P(\mathcal{D} = \{P(x) \mid x \in \mathcal{D}\})$?
If type inference is possible, then so is typechecking: given P, τ , compute $P(\mathcal{D})$, then check $\tau_P \subseteq \tau$.
- Type inference is not always possible; one has to settle with incomplete type inference that computes a superset of $P(\mathcal{D})$.

3.1 Applications

XML Publishing

Given a relational database, extract data from that database and represent it as XML document. Thus program's domain is the set of all instances of some relational database schema S .

Example 16 Relational database schema S :

```
product(pid:STRING, name:STRING, mfrprice:INTEGER)
colors(cid:STRING, pid:STRING, color:INTEGER)
sale(sid:STRING, pid:STRING, price:INTEGER)
```

Assume representation of relational DB as XML document. The following program (XQuery expression, SJP query) generates a view of the DB:

```
<catalog>
{ FOR $p IN $db/product/tuple
  RETURN <product name = {data($p/name)} >
    <mfr-price> {data($p/price)} </mfr-price>
    {FOR $s IN $db/sale/tuple
     WHERE $p/pid = $s/pid
     RETURN <sale-price> {data($s/sprice)} </sale-price>},
    {FOR $c IN $db/color/tuple
     WHERE $p/pid = $c/pid
     RETURN <color> {data($c/color)} </color>}
}
</catalog>
```

XML Transformations

- Typical application scenarios include mapping of XML documents between two different schemas, element renaming, simple computations, and translation of XML document into HTML (realized in Microsoft's IE).
- Input to program is XML document, i.e., \mathcal{D} is either \mathcal{T}_{Σ} or XML type τ ; program traverses document recursively and modifies input tree.
- Subset of XSLT:
 - recursive templates
 - modes
 - apply-templates can be called along any XPath axis
 - variables are bound to nodes in input tree, passed as parameters
 - equality test on node IDs, not node values

Example 17

```
<xsl:template match="p">
  <r>
    <xsl:apply-templates mode="mode-a"/>
    <xsl:apply-templates mode="mode-b"/>
  </r>
</xsl:template>
<xsl:template mode="mode-a" match="q">
  <a/>
</xsl:template>
<xsl:template mode="mode-b" match="q">
  <b/>
</xsl:template>
```

3.2 Type Inference

- Output type of program operating on XML view on DB:

TYPE T1 = ELEMENT catalog(T2)

TYPE T2 = (ELEMENT product(T3))*

TYPE T3 = ATTRIBUTE name(STRING), ELEMENT mfr-price(INT)
(ELEMENT sale-price(INT))*,(ELEMENT color(STRING))*

- Idea: infer type of RETURN expression from types of its components; XQuery formal semantics applies to entire XQuery language by providing inference rules for each language construct (building on earlier work on XDuce)
- Here typechecking is based on type inference: infer program's output type T1, then check if $T1 \subseteq \text{Catalog}$.
- Xquery's type inference rules need to be extended to account for integrity constraints; correct output type can only be determined if primary/foreign keys are known (multiplicity of (groups of) types/elements).
- Type inference is not complete! Assume relation $R(x,y)$:

```
<result>
  { FOR $x in $db/R/tuple RETURN <a/>,
    FOR $y in $db/R/tuple RETURN <b/>
  }
</result>
```

- XQuery's type inference determines output type

TYPE T = ELEMENT result((ELEMENT a)*, (ELEMENT b)*)

but it should be

$P(\mathcal{D}) = \{\text{ELEMENT result}((\text{ELEMENT } a)^n, (\text{ELEMENT } b)^n), n \geq 0\}$

but this is not a regular tree language.

- Consider following output type specified by user

T1 = ELEMENT result() |
ELEMENT result(ELEMENT a, (ELEMENT a)*,
ELEMENT b, (ELEMENT b)*)

- XQuery expression Q typechecks wrt T1, but typechecker rejects Q since $T \not\subseteq T1$.
- User is required to help system in typechecking through rewriting program; also, it is hard to understand for the user what went wrong
- There is no single best regular tree language T' that approximates $P(\mathcal{D})$ because such an approximation can always be improved.

3.3 Type Checking

Do typechecking without relying on type inference; possible by putting restrictions on prog language and output type.

Typechecking for XML Publishing

- For relational DB, program P , and output type τ , enumerate all relatively small databases and check output of P for conformance wrt τ
- Inefficient, but provides existence proof that typechecking problem is decidable.
- The following restrictions are applied to τ :
 - τ is a DTD
 - regular expressions in τ are star-free (no Kleene closure, but complement and empty set are allowed instead, e.g., for $\Sigma = \{a, b, c\}$, $\text{compl}(\emptyset) = \Sigma^*$, $\text{compl}(\Sigma^*.b.\Sigma^* \mid \Sigma^*.c.\Sigma^*) = a^*$)
- Theorem: Typechecking for XML publishing is decidable if query expressions are restricted to SJP queries and output type is star-free DTD.
- For increasing expressiveness of query language or more expressive schema formalisms, typechecking becomes undecidable.
 - \implies in practice one has to deal with incomplete typechecking
 - \implies type inference remains best-known approximation

Typechecking for XML Transformation

- For any program P in the XSLT fragment considered before and any regular tree language τ , the inverse type $P^{-1}(\tau)$ is also a regular tree language. As a consequence, typechecking for this language is decidable.
- Algorithm has hyper-exponential complexity
- If one adds joins to XSLT fragment, then typechecking becomes undecidable (but there is an algorithm that does incomplete typechecking for programs containing joins)

Practical considerations:

- Typechecking is important for dynamically generated XML documents.
- Often not decidable, and for cases where it is decidable, algorithms have very high complexity.
- In practice, type inference is the preferred method; despite its incompleteness properties, it is complete for many practical applications (though this hasn't been studied in great detail; what are practical applications?).