Foundations of XML Types: Tree Grammars

Pierre Genevès
CNRS

University of Grenoble Alpes, 2016–2017

Which Data Model for XML?

Trees: a natural answer

- They cannot model all XML structures (limitation: IDREFs)
- Nevertheless, throughout this session, we will focus on trees which are:
  - finite,
  - ordered (limitation: attributes),
  - labeled from a finite alphabet of symbols (limitation: values),
  - of unbounded depth and arity.

Tree Languages

A Tree Language

- is a set of trees
- can be specified by a tree grammar

Example

```
Person   = person[Name, Gender, Children?]
Name     = name[String]
Gender   = gender[Male | Female]
Male     = male[]
Female   = female[]
Children = children[Person+]
```

Terminology

- `Person` is a type variable (non-terminal) and `person` is a terminal
- A tree grammar defines a set of trees

Tree Grammars: a Syntactic Definition

Given

- An alphabet Σ
- A set of type variables ranged by X

Definition

- A tree grammar is a pair (E, X)
- E is a set of definitions of the form $\{X_1 = T_1; ...; X_n = T_n\}$
- X is the starting type variable
- Each T is a tree type expression:

$$
\begin{array}{rcll}
T & ::= & & \\
 & & l[T] & l \in \Sigma \text{ with } \textit{content model } T \\
 & | & () & \text{empty sequence} \\
 & | & T_1, T_2 & \text{concatenation} \\
 & | & T_1 \mid T_2 & \text{choice} \\
 & | & X & \text{reference}
\end{array}
$$

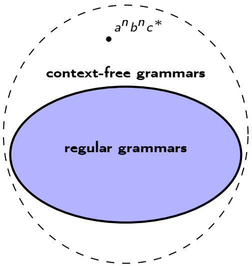
- Usual operators (?, +, *) are *syntactic sugars*

Recursion and Computability Frontier

A Syntactic Restriction

- Every recursive use of a type variable X which is not guarded (behind a label) must be in tail
- Examples (shortcut: a stands for $a[()]$) :

  $\times$ $\{X = a, X, b\}$
  $\times$ $\{X = a, Y, b; Y = X\}$
  $\checkmark$ $\{X = a, c[X], b\}$
  $\checkmark$ $\{X = a, Y; Y = b, X \mid ()\}$



WIth the restriction: *regular* tree grammars

- Decidable operations (e.g.: inclusion)
- A robust and well characterized class

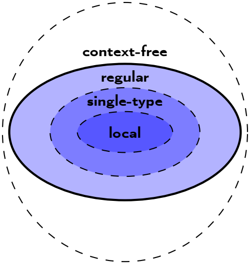
Without the restriction: *context-free* tree grammars

- Inclusion is undecidable [Hopcroft et al., 2000]
- Checking whether a context-free grammar is regular is undecidable

Classes of Tree Grammars

3 sub-classes of particular interest

- Defined by additional restrictions
- Increasing expressive power
- Correspond to XML type languages



1. *Local* tree grammars: DTD
2. *Single-type* tree grammars: XML Schema
3. *Regular* tree grammars

Local Tree Grammars: DTDs

Restriction

- Recall: each element name is associated with a regular expression
- For each $a[T_1]$ and $a[T_2]$ occuring in E, *content models* are identical: $T_1 = T_2$

Construction of Validators

- Simple principle: a word automaton is associated with each terminal
- Validation (matching) in linear time
- You know how to construct a word automaton from a regular expression...

Local Tree Grammars: DTDs

Restriction

- Recall: each element name is associated with a regular expression
- For each $a[T_1]$ and $a[T_2]$ occuring in E, *content models* are identical: $T_1 = T_2$

Construction of Validators

- Simple principle: a word automaton is associated with each terminal
- Validation (matching) in linear time
- You know how to construct a word automaton from a regular expression...
- Actually, DTD requires regular expressions to be *deterministic*:
  - × `a (bc | bb)` (matched reg. exp. part cannot be determined without look ahead of the next symbol)
  - ✓ `ab (c | b)`

Local Tree Grammars: DTDs

Restriction

- Recall: each element name is associated with a regular expression
- For each $a[T_1]$ and $a[T_2]$ occuring in E, *content models* are identical: $T_1 = T_2$

Construction of Validators

- Simple principle: a word automaton is associated with each terminal
- Validation (matching) in linear time
- You know how to construct a word automaton from a regular expression…
- Actually, DTD requires regular expressions to be *deterministic*:
  - × `a (bc | bb)` (matched reg. exp. part cannot be determined without look ahead of the next symbol)
  - ✓ `ab (c | b)`
- For any deterministic expression, we can build a deterministic automaton in linear time [Brüggemann-Klein, 1993] (see Glushkov automata)

Local Tree Grammars: DTDs

Restriction

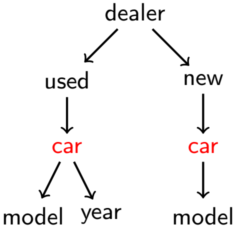
- Recall: each element name is associated with a regular expression
- For each $a[T_1]$ and $a[T_2]$ occuring in E, *content models* are identical: $T_1 = T_2$

Construction of Validators

- Simple principle: a word automaton is associated with each terminal
- Validation (matching) in linear time
- You know how to construct a word automaton from a regular expression...
- Actually, DTD requires regular expressions to be *deterministic*:
  - × `a (bc | bb)` (matched reg. exp. part cannot be determined without look ahead of the next symbol)
  - ✓ `ab (c | b)`
- For any deterministic expression, we can build a deterministic automaton in linear time [Brüggemann-Klein, 1993] (see Glushkov automata)
- Alternatively (and equivalently), we can use a derivative operator with a stack and even implement *streaming* DTD validation... Remember!

Weaknesses of DTDs

- An element name cannot have different *content models* in different contexts
- Example: a DTD cannot recognize only:



- Corollary: union of two DTDs may not be a DTD
- Class is not closed under composition (e.g. : union, complementation)

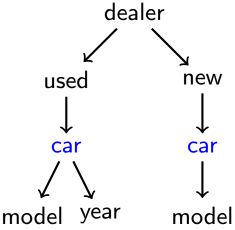
Single-Type Tree Grammars: XML Schemas... to the Rescue!

Restriction

- For each $a[T_1]$ and $a[T_2]$ occuring **under the same parent** in E, *content models* are identical: $T_1 = T_2$

$\mathcal{L}_{\text{dtd}} \subset \mathcal{L}_{\text{xmlschema}}$

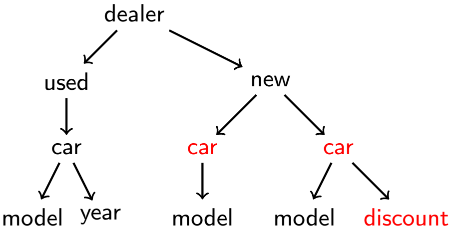
- $\mathcal{L}_{\text{dtd}}$: *content model* depends on the label of the parent
- $\mathcal{L}_{\text{xmlschema}}$: may depend on the label of any ancestor
- Strict inclusion, example of a *single-type* (and not *local*) grammar:




```
Dealer  = dealer[Used, New]
Used    = used[UsedCar]
New     = new[NewCar]
UsedCar = car[Model, Year]
NewCar  = car[Model]
...
```

XML Schemas also have Weaknesses

- "At least one car has a discount" is not *single-type*:



- Corollary: the class still not closed under union (although XML Schema specification is quite long)...

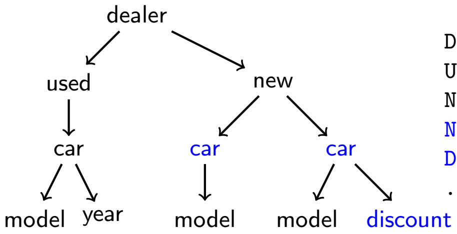
Regular Tree Grammars

No additional restriction

- A simple and powerful class
- In the XML world, it corresponds to Relax NG (see relaxng.org)

$\mathcal{L}_{\text{xmlschema}} \subset \mathcal{L}_{\text{r}}$

- $\mathcal{L}_{\text{xmlschema}}$: *content model* may depend on the label of any ancestor
- $\mathcal{L}_{\text{r}}$: *content model* may also depend on ancestor's siblings for instance
- Strict inclusion:




```
Dealer  = dealer[Used, New]
Used    = used[UsedCar]
New     = new[NewCar,DNewCar]
NewCar  = car[Model]
DNewCar = car[Model, Discount]
...
```

What do you think of this Tree Grammar?

```
Person   = MPerson | FPerson
MPerson = person[Name,gender[Male],  FSpouse?, Children?]
FPerson = person[Name,gender[Female],MSpouse?, Children?]
Male     = male[]
Female   = female[]
FSpouse = spouse[Name, gender[Female]]
MSpouse = spouse[Name, gender[Male]]
Children= children[Person+]
Name     = name[String]
```

1. Is it local (DTD-definable) ?
2. Single-type (XML-Schema definable)?
3. Regular?

What do you think of this Tree Grammar?

```
Person   = MPerson | FPerson
MPerson = person[Name,gender[Male],  FSpouse?, Children?]
FPerson = person[Name,gender[Female],MSpouse?, Children?]
Male     = male[]
Female   = female[]
FSpouse = spouse[Name, gender[Female]]
MSpouse = spouse[Name, gender[Male]]
Children= children[Person+]
Name     = name[String]
```

1. Is it local (DTD-definable) ? No
2. Single-type (XML-Schema definable)?
3. Regular?

What do you think of this Tree Grammar?

```
Person   = MPerson | FPerson
MPerson = person[Name,gender[Male],  FSpouse?, Children?]
FPerson = person[Name,gender[Female],MSpouse?, Children?]
Male     = male[]
Female   = female[]
FSpouse = spouse[Name, gender[Female]]
MSpouse = spouse[Name, gender[Male]]
Children= children[Person+]
Name     = name[String]
```

1. Is it local (DTD-definable) ? No
2. Single-type (XML-Schema definable)? No: two elements of the same name `person` with different *content models* under the same parent `children`
3. Regular?

What do you think of this Tree Grammar?

```
Person   = MPerson | FPerson
MPerson = person[Name,gender[Male],  FSpouse?, Children?]
FPerson = person[Name,gender[Female],MSpouse?, Children?]
Male     = male[]
Female   = female[]
FSpouse = spouse[Name, gender[Female]]
MSpouse = spouse[Name, gender[Male]]
Children= children[Person+]
Name     = name[String]
```

1. Is it local (DTD-definable) ? No
2. Single-type (XML-Schema definable)? No: two elements of the same name `person` with different *content models* under the same parent `children`
3. Regular? Yes! (all recursive uses of type variables are guarded)

Tree Grammars: Conclusion

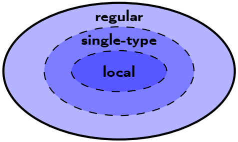
Sample Questions

- Are valid documents against type X also valid against type X'? (type inclusion, backward compability)
- Does a type X defines a non-empty set of trees? (consistency)
- Can I build the union, intersection, difference... of types X and Y and express the result with my favorite XML type language?

If we can answer for regular grammars then we can for local/single-type too!

Regular Tree Grammars

- A simply defined class
- High expressive power
- Robust (closed under set-theoretic operations)
- and well-characterized (e.g. tree automata...)



Brüggemann-Klein, A. (1993).
Regular expressions into finite automata.
Theor. Comput. Sci., 120(2):197–213.

Hopcroft, J. E., Motwani, R., Rotwani, and Ullman, J. D. (2000).
Introduction to Automata Theory, Languages and Computability.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.