

# Towards Types for Web Rule Languages

W. Drabent

Institute of Computer Science, Polish Academy of Sciences

Linköping University (Sweden)

REWERSE Summer School

28th July 2005

# Outline

- introduction
- – finite (string) automata
  - tree automata, regular sets of terms
  - tree automata generalized for unranked terms
- useful restrictions
- notes on
  - inclusion check
  - unordered content models
  - related work

## Types for (semantic) Web applications

Types – decidable sets of data objects (e.g. XML documents)  
specified by a formalism convenient for humans,  
with (rather) efficient algorithms.

Extensional (not intensional) approach

Notice: there are **other** usages of the term “type”.

# Applications

Behind membership testing ...

Known since > 40 years: typing (especially static)  $\Rightarrow$

automatic discovery of (certain) errors,  
more efficient implementation techniques,  
documentation aid,  
design discipline,  
...

Our main current interest:

providing (static) descriptive typing for Xcerpt (XML query language).

# Finite Automata (FA)

Define regular sets (languages) of **strings**.

Run of a FA  $M$  – assigning states to suffixes of the input string.

$$q_0 \leftarrow a_1 a_2 \cdots a_n$$

$$q_1 \leftarrow a_2 \cdots a_n$$

...

$$q_{n-1} \leftarrow a_n$$

$$q_n \leftarrow \epsilon$$

$$q_i \in \delta(q_{i-1}, a_i),$$

$\delta$  – the transition function of  $M$

If  $q_0$  – start state,  $q_n$  – final state then the FA **accepts** string  $a_1 \cdots a_n$ .

$$M \text{ defines } L(M) = \{ x \mid M \text{ accepts } x \}$$

## Tree Automata (TA)

Define regular sets of **trees** (of terms).

Signature  $\Sigma$  – set of function symbols, each  $f \in \Sigma$  has  $arity(f) \geq 0$ .

**Terms** over  $\Sigma$ , inductive definition:

$f \in \Sigma$ ,  $arity(f) = n$  and  $t_1, \dots, t_n$  are terms  $\Rightarrow f(t_1, \dots, t_n)$  is a term.

# Bottom-up Tree Automata (TA)

A **bottom-up tree automaton** (buTA)  $M = (Q, \Sigma, F, \Delta)$ ,

$Q$  – finite set of states,  $F \subseteq Q$  – final states,  $\Sigma$  – signature,

$\Delta$  – set of transition rules, of the form

$$f(q_1, \dots, q_n) \rightarrow q,$$

where  $f \in \Sigma$ ,  $q, q_1, \dots, q_n \in Q$ , and  $n = \text{arity}(f) \geq 0$ .

**Run**  $\rho$  of  $M$  – assigning states to subterm occurrences in the input term.  
(The rules say how to assign a state to a tree node, given the states assigned to its children.)

$$\rho(f(t_1, \dots, t_n)) = q \iff \forall_i \rho(t_i) = q_i \text{ and } f(q_1, \dots, q_n) \rightarrow q \in \Delta.$$

$t$  **accepted** by  $M$  iff  $\rho(t) \in F$  for some run  $\rho$  of  $M$ .

## Bottom-up Tree Automata (example)

$$Q = \{t, f\}, \quad F = \{t\}, \quad \Sigma = \{and, or, not, 0, 1\},$$

$$\Delta: \quad \begin{array}{ll} 0 & \rightarrow f & 1 & \rightarrow t \\ not(f) & \rightarrow t & not(t) & \rightarrow f \\ and(f, f) & \rightarrow f & and(f, t) & \rightarrow f \\ and(t, f) & \rightarrow f & and(t, t) & \rightarrow t \end{array}$$

An accepted term:

$$\begin{array}{c} not( and( 0, not( 1 ) ) ) \\ \uparrow \quad \uparrow \\ f \quad t \\ \underbrace{\hspace{10em}}_f \\ \underbrace{\hspace{15em}}_t \end{array}$$

The accepted set: true Boolean expressions without *or*

## Bottom-up Tree Automata (cont'd)

The **tree language** accepted by  $M$ :  $L(M) = \{ t \mid M \text{ accepts } t \}$ .

**Regular** tree languages – those accepted by buTA.

buTA **deterministic** – no two rules with the same left hand side.

$$(f(q_1, \dots, q_n) \rightarrow q)$$

Deterministic buTA equivalent to buTA  
(powerset construction similar to that for FA)

## Top down tree automata (tdTA)

$M = (Q, \Sigma, I, \Delta)$ ,  $Q, \Sigma$  as previously (states, signature).  
 $I \subseteq Q$  – initial states,  $\Delta$  – set transition rules, of the form

$$q \rightarrow f(q_1, \dots, q_n)$$

**Run:** If  $q \in Q$  assigned to  $f(t_1, \dots, t_n)$  and  $q \rightarrow f(q_1, \dots, q_n) \in \Delta$   
then  $q_1, \dots, q_n$  can be assigned to  $t_1, \dots, t_n$ , respectively.

A run on a term  $t$  is **accepting** if  
it assigns a state to each subterm occurrence of  $t$ .

A tdTA  $M$  is **deterministic** if

$|I| = 1$  and for any  $q, f$  there is at most one rule  $q \rightarrow f(\dots) \in \Delta$ .

## Regular Tree Languages

1. A term language  $L$  accepted by a buTA iff  $L$  accepted by a tdTA (iff  $L$  regular tree language).

Proof: reverse the rules, make the final states initial

$$f(q_1, \dots, q_n) \rightarrow q \qquad q \rightarrow f(q_1, \dots, q_n)$$

2. There exist regular tree languages  
not accepted by any deterministic tdTA.

Eg.  $\{ f(a, b), f(b, a) \}$

# Regular Term Grammars

Tree automata can be viewed as grammars.

states  $\rightsquigarrow$  nonterminal symbols

initial states  $\rightsquigarrow$  start symbols

$q \rightarrow f(q_1, \dots, q_n)$  – replace nonterminal  $q$  by  $f(q_1, \dots, q_n)$

$t$  generated  
by the grammar iff  $t$  contains no nonterminals,  
 $t$  obtained from a start symbol by  
a sequence of replacements as above.

$t$  generated by a grammar iff  $t$  accepted by the corresponding tdTA.

## Regular Tree Languages (2)

– defined by (deterministic) buTA, tdTA, regular term grammars.

- Closed under  $\cup, \cap, \overline{\phantom{x}}$
- $t \in L(M)$ ? – time  $O(|t| \cdot |M|)$   
 $O(|t|)$  when  $M$  deterministic (buTA or tdTA)
- $L(M) = \emptyset$ ? – time  $O(|M|)$
- $\overline{L(M)} = \emptyset$ ? – EXPTIME-complete
- $L(M_1) \subseteq L(M_2)$ ? – EXPTIME-complete  
polynomial when  $M_2$  deterministic tdTA

## Semi-structured Data, Unranked Terms

The trees/terms considered so far are **ranked**:

Each symbol has a fixed number of arguments.

The trees corresponding to XML documents are **unranked**:

Each symbol has an arbitrary number of arguments.

(Arbitrary number of elements in an element's content.)

# Data Terms – Abstraction of Semi-structured Data

– suggested by Xcerpt

Alphabets:

$\mathcal{B}$  – **basic constants** (represent numbers, strings, ...)

$\mathcal{L}$  – **labels** (represent XML tags and attribute names).

[ ] for ordered arguments, { } for unordered arguments.

Data term:  $b \in \mathcal{B}$ ,

$l[t_1, \dots, t_n]$

$l\{t_1, \dots, t_n\}$

where  $l \in \mathcal{L}$ ,  $t_1, \dots, t_n$  are data terms.

## Data Terms, Example

XML element

```
<CD price="15.90" year="1994">  
  Praetorius Mass  
  <subtitle></subtitle>  
  <artist>Gabrielli Consort and Players</artist>  
</CD>
```

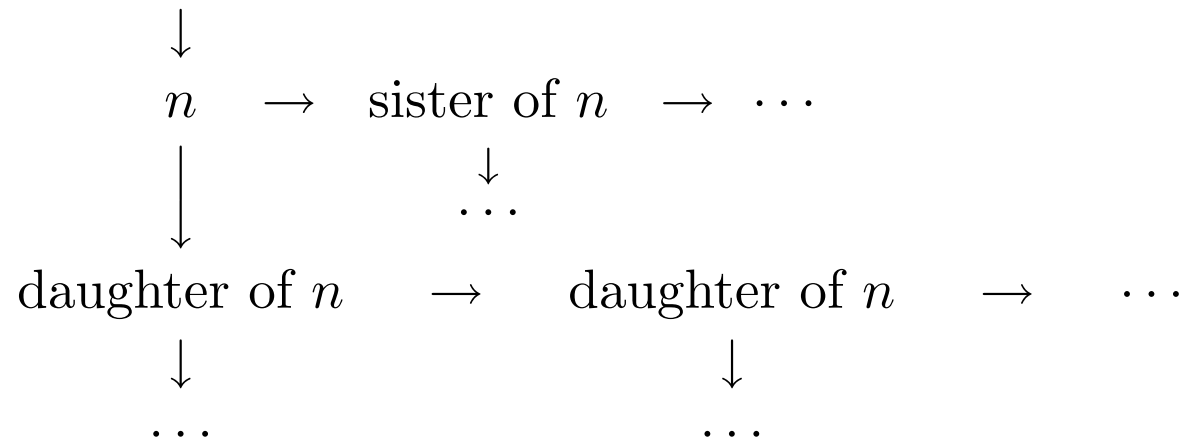
can be represented as a data term

```
CD[ attributes{ price[15.90] year[1994] }  
  Praetorius_Mass  
  subtitle[]  
  artist[Gabrielli_Consort_and_Players]  
  ]
```

where  $15.90, 1994, \text{Praetorius\_Mass}, \text{Gabrielli\_Consort\_and\_Players} \in \mathcal{B}$   
 $\text{attributes}, \text{price}, \text{year}, \text{subtitle}, \text{artist} \in \mathcal{L}$ .

## Defining Sets of Unranked Terms

- Represent unranked terms by terms  
(standard representation of trees as binary trees).



Obscures the difference children / siblings.

- Generalize to unranked terms a formalism for terms.

# Tree Automata Generalized for Unranked Terms

Various generalizations exists.  
Practically equivalent.

Main idea:

TA	Generalized TA
The children of a node described by	
single sequence of states	regular language of such sequences

Why regular?  $\in, \subseteq, =$  decidable, closed under  $\cap, \cup, \bar{\phantom{x}}$ .

N.B. Equivalent formalism, not related to automata:

MSO (1st order logic + quantification over set variables).

# Tree Automata Generalized for Unranked Terms (2)

- [Brüggemann-Klein, Murata, Wood] hedge automata f
- [Murata, Lee, Mani] “regular tree grammars” t
- [Hosoya, Vouillon, Pierce] “type definitions” f
- [Neven] unranked tree automata t
- [Papakonstantinou, Vianu] specialized DTD’s tf
- ...

Design decision: t trees or sequences of trees ?  
(f forests, hedges)

The difference inessential.

We choose trees  
(in our formalism of type definitions).

# Tree Automata Generalized for Unranked Terms: Type Definitions

Alphabets:  $\mathcal{C}$  – type constants,  $\mathcal{V}$  – type variables,  $\mathcal{V} \cup \mathcal{C}$  – type names.

For each  $C \in \mathcal{C}$  a set  $\llbracket C \rrbracket$  of basic constants given.

Regular languages (of strings) over  $\mathcal{V} \cup \mathcal{C}$ .

Regular expressions over  $\mathcal{V} \cup \mathcal{C}$  (*regular type expressions*).

Notation for regular expressions:

$\tau^{(n:m)}$	for	$\tau^n   \dots   \tau^m$
$\tau^{(n:\infty)}$	for	$\tau^n \tau^*$
$\tau^+$	for	$\tau \tau^*$
$\tau^?$	for	$\tau^{(0:1)}$ , (and $\varepsilon   \tau$ )

*Multiplicity list* – regular expression  $T_1^{(n_1:m_1)} \dots T_k^{(n_k:m_k)}$ ,  
where  $T_1, \dots, T_k \in \mathcal{V} \cup \mathcal{C}$ ,  $T_1, \dots, T_k$  distinct.

## Type Definitions (2)

Type definition – finite set  $D$  of rules

$$T \rightarrow l[\tau] \quad \text{or} \quad T \rightarrow l\{\mu\} \quad \left( \begin{array}{l} \text{rule for } T \text{ with label } l \\ \text{and content model } \tau \text{ or } \mu \end{array} \right)$$

where  $l \in \mathcal{L}$ ,  $\tau$  – regular type expression,  $\mu$  – multiplicity list.

$D$  assigns to each type name  $T$  a set  $\llbracket T \rrbracket$  of data terms  
obtained from  $T$  by rewriting according to rules from  $D$ .

(Automata-like view – more concise, see next slide.)

N.B. the choice of multiplicity lists – rather arbitrary.

## The Sets Defined by a Type Definition

( $D$  – type definition,  $T$  – type name,  $\llbracket T \rrbracket$  – corresponding set)

$t \in \llbracket T \rrbracket$  iff

there exists  $\nu$ : (subterm occurrences in  $t$ )  $\rightarrow \mathcal{V} \cup \mathcal{C}$  such that

1.  $\nu(t) = T$ ,
2.  $\nu(b) = C \in \mathcal{C}$  if  $b \in \llbracket C \rrbracket$  (and  $b$  is a basic constant),
3.  $\nu(l[t_1, \dots, t_n]) = U \in \mathcal{V}$  if  $U \leftarrow l[\tau] \in D$  and  $\nu(t_1) \cdots \nu(t_n) \in L(\tau)$ ,
4.  $\nu(l\{t_1, \dots, t_n\}) = U \in \mathcal{V}$  if  $U \leftarrow l\{\mu\} \in D$  and  $\nu(t_1) \cdots \nu(t_n)$  is  
a permutation of a string from  $L(\mu)$ .

N.B. If  $\nu(t') = U$  then  $t' \in \llbracket U \rrbracket$ .

## The Sets Defined by a Type Definition (2)

Ex.  $D: \text{Person} \rightarrow \text{person}[\text{Name} (M|F) \text{Person}^{(0:2)}]$   
 $\text{Name} \rightarrow \text{name}[\#name]$   
 $M \rightarrow m[ ]$   
 $F \rightarrow f[ ]$

where

$$\text{Person}, \text{Name}, M, F \in \mathcal{V}, \quad \#name \in \mathcal{C}, \quad \text{john}, \text{mary}, \text{bob} \in [\#name].$$

A data term in  $[[\text{Person}]]_D$ :

$$\underbrace{\underbrace{\underbrace{\text{person}[\text{name}[\text{john}]}]}_{\#name} \underbrace{m[ ]}_{M}}_{\text{Name}} \underbrace{\underbrace{\underbrace{\text{person}[\text{name}[\text{mary}]}]}_{\#name} \underbrace{f[ ]}_{F}}_{\text{Name}} \underbrace{\underbrace{\underbrace{\text{person}[\text{name}[\text{bob}]}]}_{\#name} \underbrace{m[ ]}_{M}}_{\text{Name}}}_{\text{Person}}}_{\text{Person}}$$

as

$$\text{Name } M, \text{Name } F \text{Person}, \text{Name } M \text{Person} \in L(\text{Name}(M|F)\text{Person}^{(0:2)})$$

# Useful Restrictions of Type Definitions

Type definitions → a natural class of “regular sets” of data terms.

Roughly speaking,

the sets defined by DTD, XML Schema, Relax NG, ...

can be defined by type definitions.

(without non context-free conditions, like

*xsd:unique*, *xsd:key*, *xsd:keyref* of XML Schema)

Some problems expensive for regular sets,

e.g. inclusion check is EXPTIME-hard.

Interesting: classes of regular sets with efficient algorithms.

A classification based on [Murata, Lee, Mani'01].

## Useful Restrictions of Type Definitions (2)

1. At most one rule  $T \rightarrow l \dots$  for any  $T$  and  $l$ .

No restriction for sets of ordered data terms;

instead of  $T \rightarrow l[\tau_1], T \rightarrow l[\tau_2]$  we can use  $T \rightarrow l[\tau_1 | \tau_2]$ .

(Excluded: sets containing both  $l\{\dots\}$  and  $l[\dots]$ ,  
some sets defined by  $T \rightarrow l\{\mu_1\}, T \rightarrow l\{\mu_2\}$  )

Later on we require all type definitions to satisfy 1.

2. At most one rule for any  $T$  (single-label type def's).

If a set  $S = \llbracket T \rrbracket_D$  is defined by a  $D$  satisfying restriction 1

then  $S = \llbracket T_1 \rrbracket_{D'} \cup \dots \cup \llbracket T_n \rrbracket_{D'}$  for a  $D'$  which is single-label.

Type variables  $T_1, T_2$  are **competing** in  $D$  if  $T_1 \neq T_2$  and  $D$  contains

$$T_1 \rightarrow l \dots \quad T_2 \rightarrow l \dots$$

Type constants  $C_1, C_2$  are **competing** if  $\llbracket C_1 \rrbracket \cap \llbracket C_2 \rrbracket \neq \emptyset$ .

3.  $D$  **local** if **no competing type names in  $D$** .

Given  $t$ ,  $t \in \llbracket T \rrbracket_D$  for at most one  $T$ .

Checking  $l[t_1, \dots, t_n] \in \llbracket U \rrbracket_D$ :

- for each  $t_i$  take its label  $l_i$ ,  
     find  $T_i \rightarrow l_i \dots \in D$ ,  
     check  $t_i \in \llbracket T_i \rrbracket_D$ ,
- find  $U \rightarrow l[\tau] \in D$ ,
- check  $T_1 \cdots T_n \in L(\tau)$ .

DTD's: labels are (also) type variables.

DTD's can be expressed as *local, single-label* type definitions.

## Single-type Type Definitions

For the membership check algorithm (previous slide),  
the condition below sufficient.

4.  $D$  **single-type** if **no competing type names**  
**in a content model in  $D$**

Given  $t$ ,  $t \in \llbracket T \rrbracket_D$  for at most one  $T$  from a r.h.s. of a rule of  $D$ .

5.  $D$  **proper** = **single-type and single-label**.

XML Schema	$\rightsquigarrow$	single-type definitions [Murata et al.]
(without		proper <sup>1</sup>
<i>xsi:type</i> /derived types, ...)		

---

<sup>1</sup> maybe XML Schema without substitutionGroup.

## Restrained-competition Type Definitions

6. Type definition  $D$  is **restrained-competition** if
- i) **for each content model  $\tau$  in a  $T \rightarrow l[\tau] \in D$**   
 $xU_1v \in L(\tau)$   
 $xU_2w \in L(\tau)$  implies  $U_1, U_2$  not competing.
  - ii) **no competing type names in any  $\mu$  in a  $T \rightarrow l\{\mu\} \in D$**   
 (like in 4. single-type)

Checking  $l[t_1, \dots, t_n] \in \llbracket U \rrbracket_D$ :

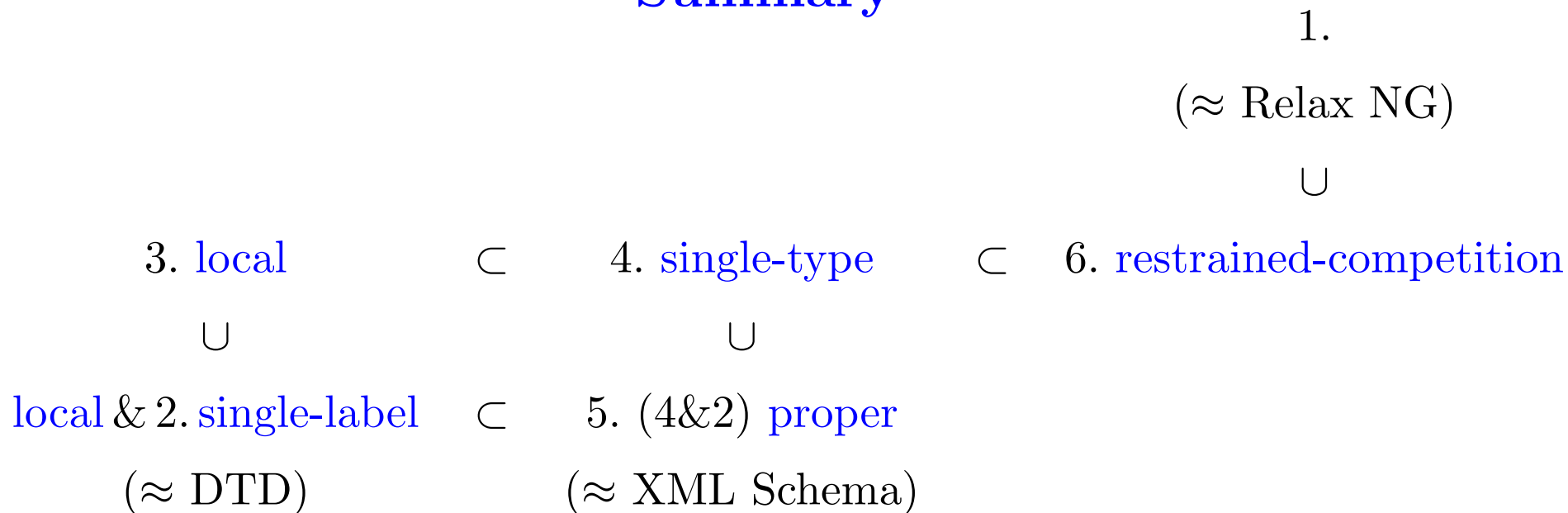
- find  $U \rightarrow l[\tau] \in D$ ,
- for  $i = 1, \dots, n$ 
  - using  $T_1 \cdots T_{i-1}$  and the label of  $t_i$  find  $T_i$ ,
  - check  $t_i \in \llbracket T_i \rrbracket_D$ ,
- check  $T_1 \cdots T_n \in L(\tau)$ .

## Summary

1. At most one rule  $T \rightarrow l \dots$  for any  $T$  and  $l$ .
2. At most one rule for any  $T$  (single-label type def's).
3.  $D$  local if no competing type names in  $D$ .
4.  $D$  single-type if no competing type names in a content model in  $D$
5.  $D$  proper = 4. single-type and 2. single-label.
6. Type def.  $D$  is restrained-competition if
  - i) for each content model  $\tau$  in a  $T \rightarrow l[\tau] \in D$ 

$$\begin{array}{l} xU_1v \in L(\tau) \\ xU_2w \in L(\tau) \end{array}$$
 implies  $U_1, U_2$  not competing.
  - ii) no competing type names in any  $\mu$  in a  $T \rightarrow l\{\mu\} \in D$  (like in 4. single-type)

## Summary



## Closure properties

Type definitions	sets closed under
1 without { }, 2 without { }, 3, ..., 6	$\cap$
1 without { }	$\cap, \cup, \bar{\phantom{x}}$

# Applications

Behind membership testing ...

Known since > 40 years: typing (especially static)  $\Rightarrow$

automatic discovery of (certain) errors,  
more efficient implementation techniques,  
documentation aid,  
design discipline,  
...

Our main interest: providing (static) descriptive typing for Xcerpt  
(XML query language).

Assuming that input data for program  $Q$  are from type  $T$ ,  
compute result type  $U$ :

$Q$  applied to  $t \in \llbracket T \rrbracket$  results in  $t' \in \llbracket U \rrbracket$ .

Typechecking by checking  $\subseteq$ .

## Polynomial Inclusion Check

Checking  $\llbracket T_1 \rrbracket_{D_1} \subseteq \llbracket T_2 \rrbracket_{D_2}$ , where  $D_2$  is proper (or single-type)

☺ is **polynomial** w.r.t.  $|D_1|, |D_2|$  and  
the sizes of DFA's for the content models of  $D_1, D_2$ .

[Bry,Drabent,Małuszyński]

☹ Constructing DFA's from regular expressions – **exponential**,

☺ but linear for **1-unambiguous** regular expressions [Brüggemann-Klein,Wood].

Apparently, such reg. expr. used in DTD or XML Schemas:

“Unique particle attribution constraint” (§ 3.8.6 of XML Schema Part 1),

“Deterministic Content Models” (App. E of XML 1.0).

## Polynomial Inclusion Check (2)

The problem is EXPTIME-hard.

[Hosoya et. al] – algorithm reported to behave efficiently on practical examples.

Employs: TA + representing unranked terms as binary trees.

Here – formalism working directly with unranked terms.

Makes possible selecting a sub-problem – with efficient solution,  
– of practical importance.

## Note: Unordered Content Models

$$T \rightarrow l\{\mu\}$$

Why restrictions on  $\mu$ ?

Simplicity, efficiency.

Arbitrary **reg.expr.**  $\Rightarrow$  non context-free languages,  
non linear membership check,  
checking equivalence of  $T \rightarrow \{\tau_1\}$ ,  $T \rightarrow \{\tau_2\}$   
CoNEXPTIME [Huynh'85].

The choice of **multiplicity lists** – rather arbitrary.

Result: Type definitions not closed under  $\cup, \cap, \bar{\phantom{x}}$ .

**E.g.**  $T \rightarrow l\{T_1^{(0:2)}T_2^{(0:2)}\}$ ,  $U \rightarrow l\{T_3^{(0:3)}\}$ , ...

assume  $\llbracket T_1 \rrbracket \subseteq \llbracket T_3 \rrbracket$ ,  $\llbracket T_2 \rrbracket \subseteq \llbracket T_3 \rrbracket$ ,  $\llbracket T_1 \rrbracket \cap \llbracket T_2 \rrbracket = \emptyset$ .

$\llbracket T \rrbracket \cap \llbracket U \rrbracket$  cannot be defined by a type def. with multiplicity lists  
( $\leq 2$  from  $\llbracket T_1 \rrbracket$ ,  $\leq 2$  from  $\llbracket T_2 \rrbracket$ , total  $\leq 3$ ).

## Unordered Content Models (2)

$$T \rightarrow l\{\mu\}$$

Neven, Schwentick –  $\mathcal{L}_{\geq}$  formulae (for unordered content models):

$$\varphi ::= \text{true} \mid \text{false} \mid a = i \mid a \geq i \mid \neg\varphi \mid \varphi \vee \varphi \quad a \in \Sigma, i \in \mathbb{N}$$

For  $w \in \Sigma^*$

$$w \models a = i \quad \text{iff} \quad \#a(w) = i \quad w \models \neg\varphi \quad \text{iff} \quad \text{not } w \models \varphi$$

$$w \models a \geq i \quad \text{iff} \quad \#a(w) \geq i \quad \dots$$

$$L(\varphi) = \{w \mid w \models \varphi\}$$

$L(\varphi)$  – regular.

Membership linear (in  $|w|$ , for fixed  $\varphi$ ).

$\mathcal{L}_{\geq}$  formulae – more convenient than the equivalent reg. expr., or FA,

– useful expressiveness

$\mathcal{L}_{\geq}$  formulae  $\equiv$  unions of multiplicity lists (as unordered content models)

Conjecture: Type definitions with  $\mathcal{L}_{\geq}$  formulae – closed under  $\cup, \cap, \bar{\phantom{x}}$ .

Conjecture:

Unrestricted type definitions with  $\mathcal{L}_{\geq}$  formulae – closed under  $\cup, \cap, \overline{\phantom{x}}$ .

Difficulty: complementation;

construction of a type definition for  $\overline{\llbracket T \rrbracket_D}$  from  $D$  – complicated.

Note: Even restriction 1. makes type definitions not closed under  $\overline{\phantom{x}}$ .

## Unordered Content Models (3)

Example (cnt'd)

$$T \rightarrow l\{T_1^{(0:2)}T_2^{(0:2)}\}, \quad U \rightarrow l\{T_3^{(0:3)}\}, \quad \dots$$

where  $\llbracket T_1 \rrbracket \subseteq \llbracket T_3 \rrbracket$ ,  $\llbracket T_2 \rrbracket \subseteq \llbracket T_3 \rrbracket$ ,  $\llbracket T_1 \rrbracket \cap \llbracket T_2 \rrbracket = \emptyset$ .

$\llbracket T \rrbracket \cap \llbracket U \rrbracket$  expressed by

$$TU \rightarrow l\{ (T_1 \leq 2 \wedge T_2 \leq 1) \vee (T_1 \leq 1 \wedge T_2 \leq 2) \}$$

where  $a \leq i$  is  $\neg(a \geq i + 1)$ ,  $\varphi_1 \wedge \varphi_2$  is  $\neg(\neg\varphi_1 \vee \neg\varphi_2)$ .

## Note: Related Research on Typechecking

Substantial theoretical work on typechecking  
of transformations defined by unranked tree transducers.

u. t. transducers – abstraction of XSLT, XQuery, ...

Milo, Suciu, Vianu, Neven, Martens, Alon, Hosoya, Pierce ...

see e.g. the references in [Martens,Neven PODS'04].

1. In general – typechecking **undecidable**.

Hence **incomplete** typechecking needed.

(Incomplete: YES means yes,  
NO means don't know.)

Classes (of programs, input and output types) studied  
for which the problem is decidable/undecidable.

Complexity results.

2. Typechecking by type inference + inclusion check is **weak**.

For some cases complete typechecking possible

but not by

computing a type  $T_{out}$  of results + checking  $\llbracket T_{out} \rrbracket \subseteq \llbracket T_{spec} \rrbracket$ .

Because

the set  $R$  of results may be non regular (inexpressible by a type definition)

and for each regular  $\llbracket T_{out} \rrbracket \supseteq R$  there exists a regular  $\llbracket T_{spec} \rrbracket \supseteq R$  such that  $\llbracket T_{out} \rrbracket \not\subseteq \llbracket T_{spec} \rrbracket$ .

Some references not present in the printed paper.

Dung T. Huynh. The complexity of equivalence problems for commutative grammars. *Information and Control*, 66(1/2):103–121, 1985.

F. Neven. “Automata, Logic, and XML.” In Proceedings CSL 2002.

W. Martens, F. Neven, T. Schwentick. “Which XML Schemas Admit 1-Pass Preorder Typing?” In Proceedings ICDT 2005.

Y. Papakonstantinou, V. Vianu. “DTD Inference for Views of XML Data.” In Proceedings PODS 2000, pp. 35–46, 2000.

A nicer version of M. Murata, D. Lee, and M. Mani; “Taxonomy of XML schema languages using formal language theory” is in <http://www.mulberrytech.com/Extreme/Proceedings/>

**Thank You**