Formal Methods for XML: Algorithms & Complexity

S. Margherita di Pula
September 2004

Thomas Schwentick

Disclaimer

About this talk

It will be Theory

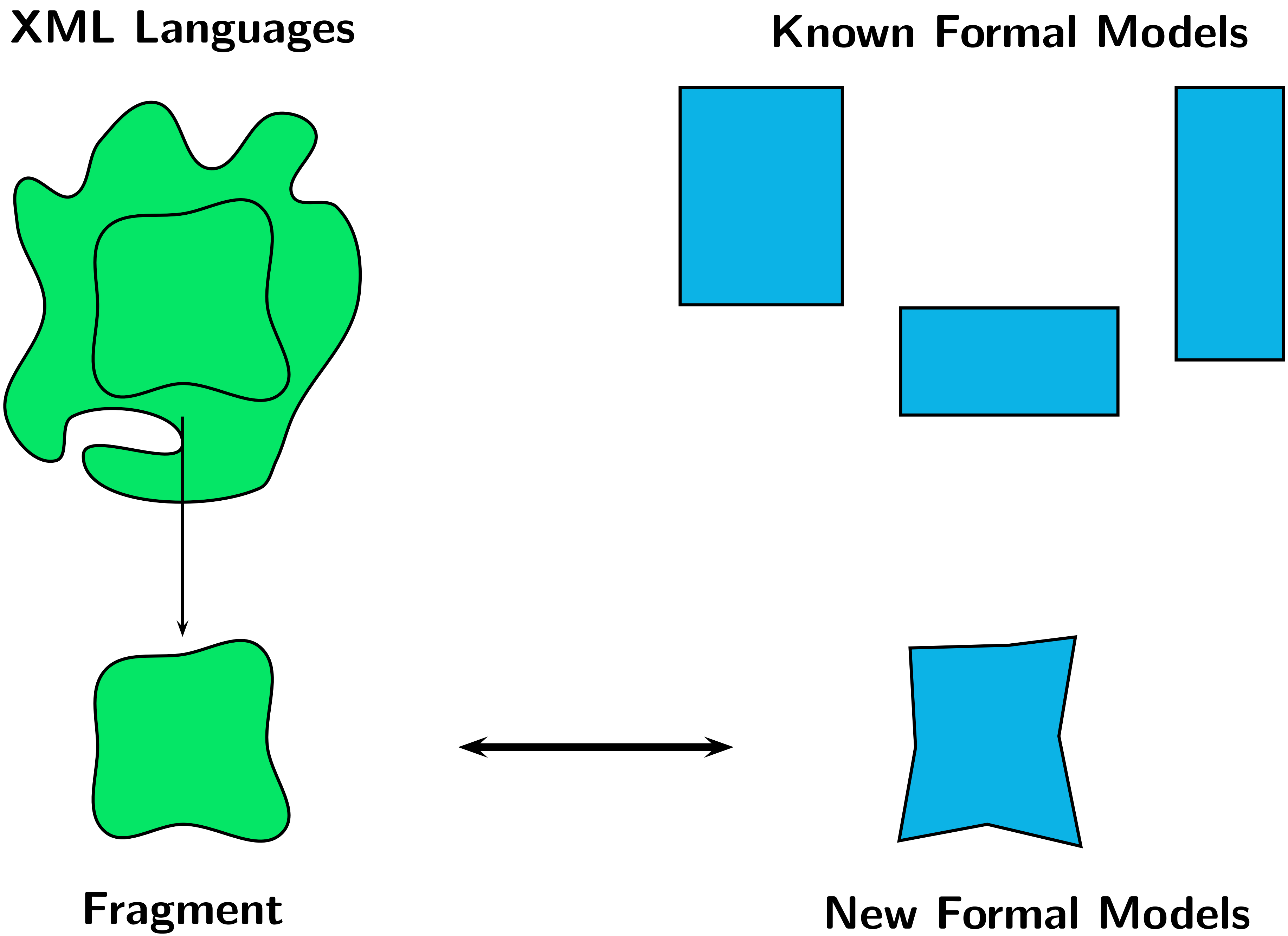
and it will be a lot of Theory

Welcome to this talk!

Note

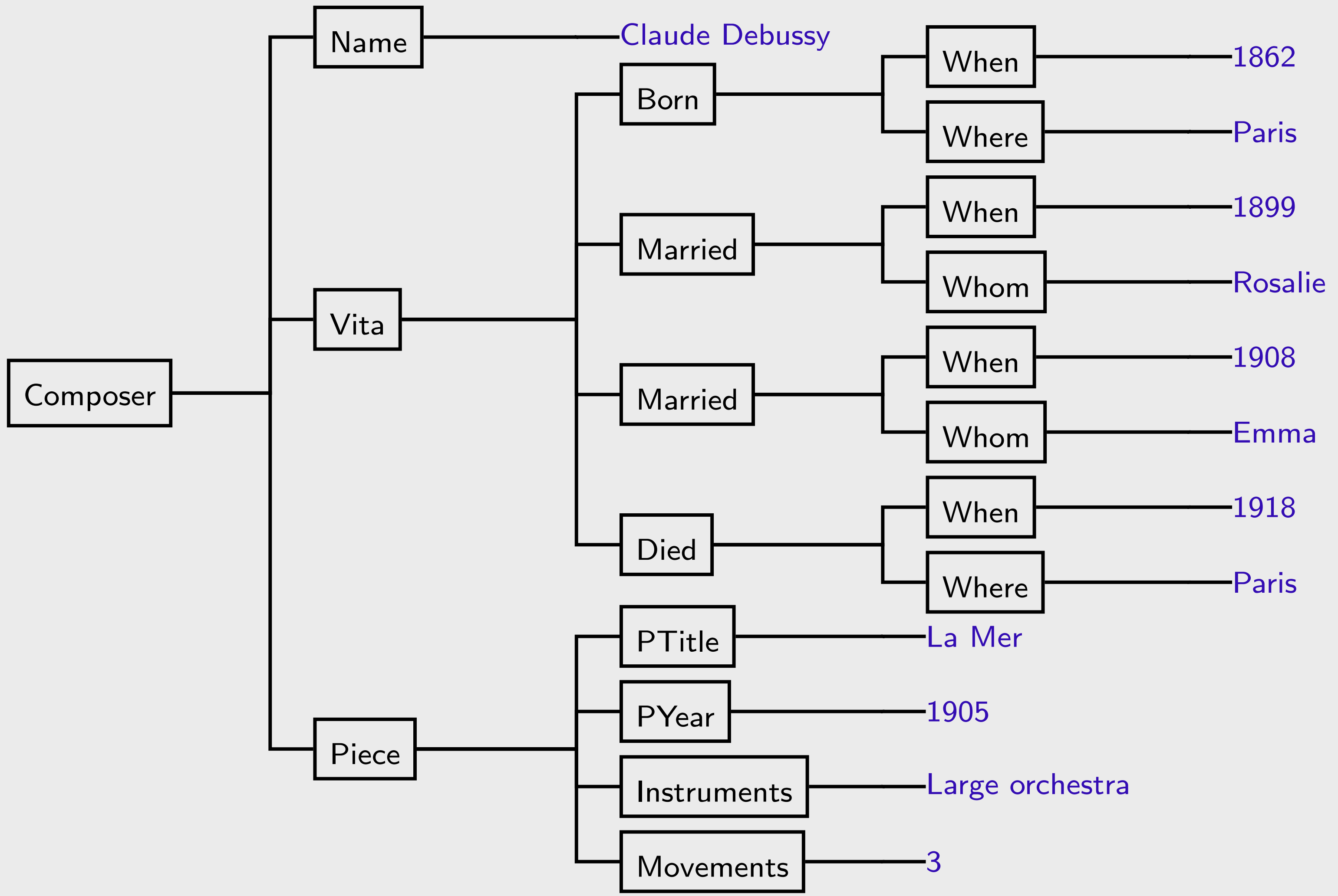
The uptodate slides will be available from my homepage

The Big Picture



XML

Example Tree



XML Processing

Four important kinds of XML processing . . . . and their languages

Validation `DTD, XML Schema`

Check whether an XML document is of a given type

Navigation `XPath`

Select a set of positions in an XML document

Querying `XQuery`

Extract information from an XML document

Transformation `XSLT`

Construct a new XML document from a given one

Validation: DTD

DTD

DTDs describe types of XML documents

Example document

⟨Composer⟩
  ⟨Name⟩ Claude Debussy ⟨/Name⟩
  ⟨Vita⟩
    ⟨Born⟩ ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
    ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
    ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
    ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩ ⟨/Died⟩
  ⟨/Vita⟩
  ⟨Piece⟩
    ⟨PTitle⟩ La Mer ⟨/PTitle⟩
    ⟨PYear⟩ 1905 ⟨/PYear⟩
    ⟨Instruments⟩ Large orchestra ⟨/Instrun
    ⟨Movements⟩ 3 ⟨/Movements⟩
    ...
  ⟨/Piece⟩
  ...
⟨/Composer⟩
...

Example

```
<!DOCTYPE Composers [
    <!ELEMENT Composers (Composer*)>
    <!ELEMENT Composer (Name, Vita, Piece*)>
    <!ELEMENT Vita (Born, Married*, Died?)>
    <!ELEMENT Born (When, Where)>
    <!ELEMENT Married (When, Whom)>
    <!ELEMENT Died (When, Where)>
    <!ELEMENT Piece (PTitle, PYear,
         Instruments, Movements)>
    ]>
```

Navigation: XPath

XPath

XPath expressions select sets of nodes of XML documents by specifying navigational patterns

Example doc

⟨Composer⟩
  ⟨Name⟩ Claude Debussy ⟨/Name⟩
  ⟨Vita⟩
    ⟨Born⟩ ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
    ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
    ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
    ⟨Died⟩ ⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩ ⟨/Died⟩
  ⟨/Vita⟩
  ⟨Piece⟩
    ⟨PTitle⟩ La Mer ⟨/PTitle⟩
    ⟨PYear⟩ 1905 ⟨/PYear⟩
    ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
    ⟨Movements⟩ 3 ⟨/Movements⟩
    ...
  ⟨/Piece⟩
  ...
⟨/Composer⟩
...

Example query

`//Vita/Died/*`

Querying: XQuery

Result

⟨Name⟩ Claude Debussy ⟨/Name⟩
⟨Name⟩ Eric Satie ⟨/Name⟩
⟨Name⟩ Hector Berlioz ⟨/Name⟩
⟨Name⟩ Camille Saint-Saëns ⟨/Name⟩
⟨Name⟩ Frédéric Chopin ⟨/Name⟩
⟨Name⟩ Maurice Ravel ⟨/Name⟩
⟨Name⟩ Jim Morrison ⟨/Name⟩
⟨Name⟩ César Franck ⟨/Name⟩
⟨Name⟩ Gabriel Fauré ⟨/Name⟩
⟨Name⟩ George Bizet ⟨/Name⟩
...

XQuery

XQuery is a full-fledged XML query language

ere⟩ Paris ⟨/Where⟩⟨/Born⟩
n⟩ Rosalie⟨/Whom⟩⟨/Married⟩
n⟩ Emma ⟨/Whom⟩⟨/Married⟩
⟩ Paris ⟨/Where⟩ ⟨/Died⟩

⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
⟨Movements⟩ 3 ⟨/Movements⟩
...
⟨/Piece⟩
...
⟨/Composer⟩
...

Example query

```
for $x in doc('composers.xml')/Composer
where $x/Vita/Died/Where = 'Paris'
return $x/Name
```

Transformation: XSLT

Result

<ParisComposer>
<Name> Claude Debussy </Name>
<Born>
<When> August 22, 1862 </When>
<Where> Paris </Where>
</Born>
</ParisComposer>
<ParisComposer>
<Name> Frédéric Chopin </Name>
<Born>
<When> March 1, 1810 </When>
<Where> Żelazowa </Where>
</Born>
</ParisComposer>
<ParisComposer>
<Name> Camille Saint-Saëns </Name>
<Born>
<When> October 9, 1835 </When>
<Where> Paris </Where>
</Born>
</ParisComposer>

XSLT

XSLT transforms documents by means of templates

umen

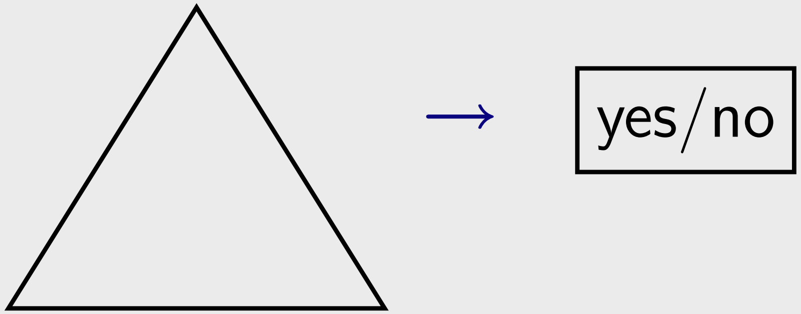
en><Where> Paris </Where></Born>
><Whom> Rosalie</Whom></Married>
><Whom> Emma </Whom></Married>
<Where> Paris </Where> </Died>

Example

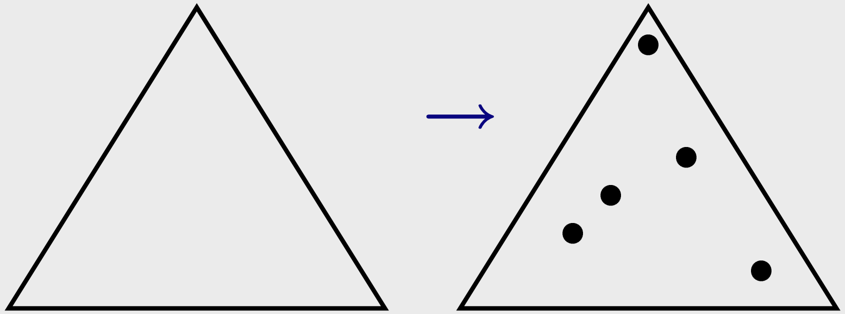
ate match="Composer[Vita//Where='Paris']">
omposer>
copy-of select="Name" />
copy-of select="Vita/Born" />
Composer>
plate>

A Schematic View

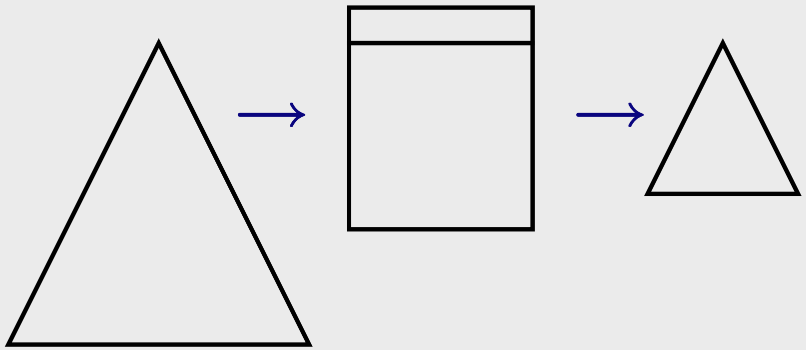
DTD/ XML Schema



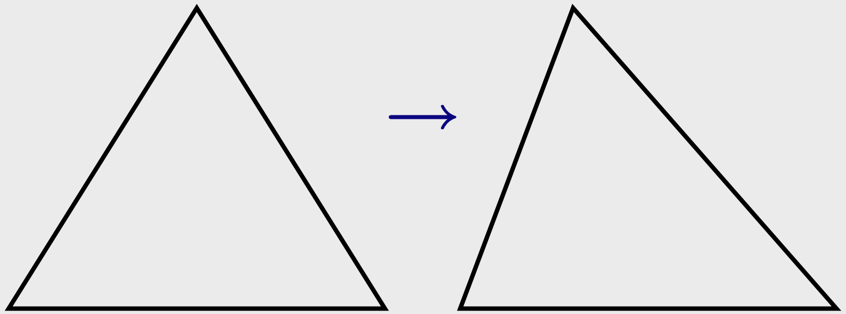
XPath



XQuery



XSLT



Focus of this Talk

Topics

- Expressive power of `XML` languages
- Complexity of algorithmic tasks related to `XML` processing
- Tradeoff between expressiveness and complexity

Goals of this Research

- Understand expressive power and complexity of XML languages
- Identify interesting fragments with good tradeoff

Algorithmic Tasks

Evaluation

Evaluation (Combined)

I: Tree t, Query q

O: $q(t)$

Evaluation (Data(q))

I: Tree t

O: $q(t)$

Incremental Eval. (q)

I: Tree t,
Changes of t

O: $q(t)$

Static Analysis

Satisfiability

I: Query q

Q: Is $q(t) \neq \emptyset$
for some t?

Containment

I: Queries q_1, q_2

Q: Is always
$q_1(t) \subseteq q_2(t)$?

Equivalence

I: Queries q_1, q_2

Q: Is always
$q_1(t) = q_2(t)$?

Type Checking

I: Types d_1, d_2,
Transformation T

Q: Does $t \models d_1$ imply
$T(t) \models d_2$?

Type Inference

I: Types d,
Transformation T

O: Type of
$\{T(t) \mid t \models d\}$

Expressive power

Question: How do we measure expressive power?

Remarks

- Classes of logical formulas are a good yardstick

$\longrightarrow$ They provide methods to prove that a query can not be expressed

Recall Relational Databases

- Core of SQL $\equiv$ First-order Logic
- Most frequently asked queries $\equiv$ Conjunctive queries

Contents

Background: Complexity Classes

Overview of Complexity Classes

Decidable	
EXPSPACE	...... Equivalence of reg. expressions with squaring
EXPTIME	.... 2-Player Corridor Tiling
PSPACE	.... Quantified Boolean Formulas
coNP **NP**	.... Satisfiability of prop. formulas
P	.... Boolean circuit evaluation
NC	.... *Efficiently parallelizable problems*
LOGCFL	.... Acyclic conjunctive queries
NL	.... Reachability in directed graphs
LOGSPACE	.... Reachability in directed forests

XML, Trees and Automata

Question: Why is XML appealing for Theory people?

Years ago...

- Theoretical Computer Science for Database Theorists: Logics, Complexity, Algorithms,...
- Database Theory for Theoretical Computer Scientists: terra incognita

After the advent of XML

Many connections between

Formal Languages & Automata Theory

and

XML & Database Theory

XML, Trees and Automata

Question: Why trees?

A Natural Answer

- Trees reflect the hierarchical structure of XML
- Underlying data model of XML is tree based

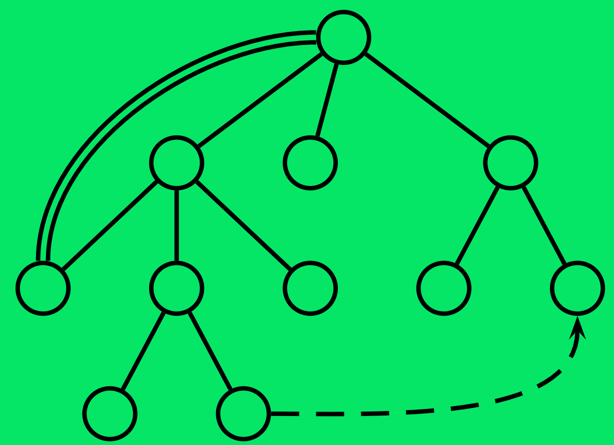
Limitations

- But trees can not model all aspects of XML (e.g., IDREFs, data values)

$\Rightarrow$ Sometimes extensions are needed

- E.g., directed graphs instead of trees

Example



Nevertheless

In this tutorial we will concentrate on the tree view at XML

XML, Trees and Automata

Question: Why automata?

Ingredients of XML

Concepts from formal languages are obviously present around XML:

- Labelled trees
- DTD: context-free grammars
- DTD: regular expressions
- XPath: regular path expressions

We will see

Automata turn out to be useful as:

- a means to define robust classes with clear semantics
- a tool for proofs
- an algorithmic tool for static analysis
- a tool for query evaluation

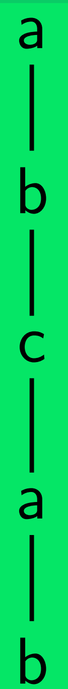
Contents

From Strings to Trees

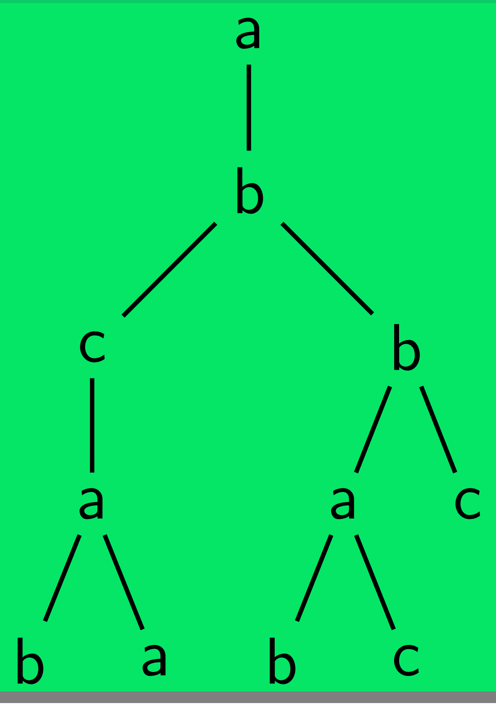
A String

abcab

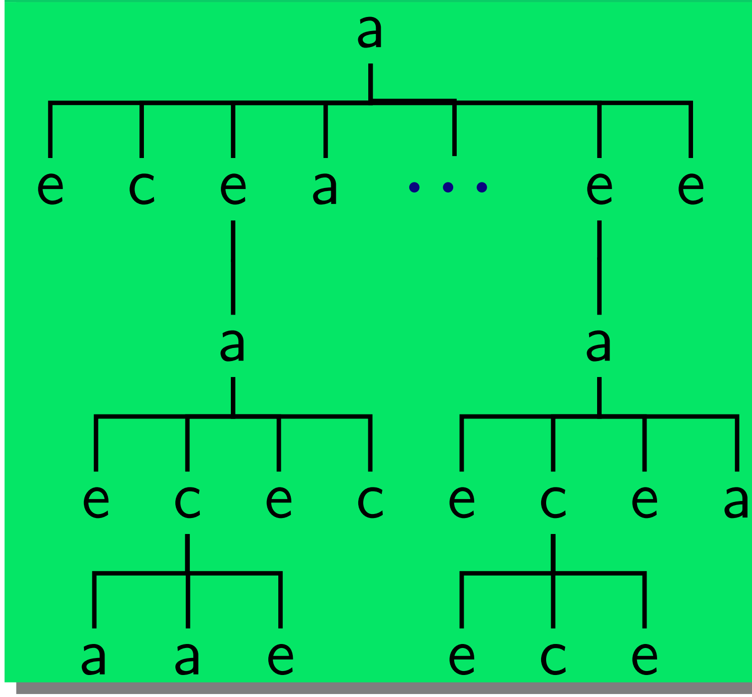
String as Tree



A Ranked Tree



An Unranked Tree



From Strings to Trees (cont.)

XML and Trees

- XML trees are **unranked**:

  the number of children of a node is not restricted

- Automata have first been considered on **ranked** trees,
  where each symbol has a fixed number of children (rank)

- Most important ideas were already developed for ranked trees

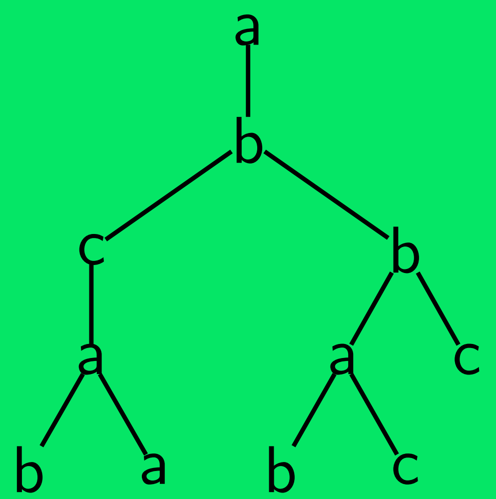
$\longrightarrow$ Let us take a look at this first

Trees as Terms

Remark

Sometimes trees are viewed as terms

Example



Example Tree as Term

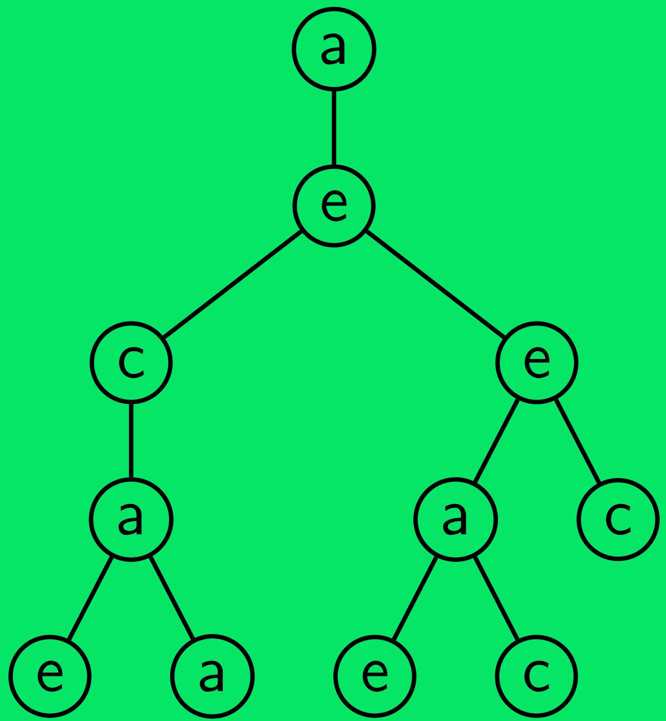
$$a^1(b^2(c^1(a^2(b,a)),b^2(a^2(b,c),c)))$$

From String Automata to Tree Automata

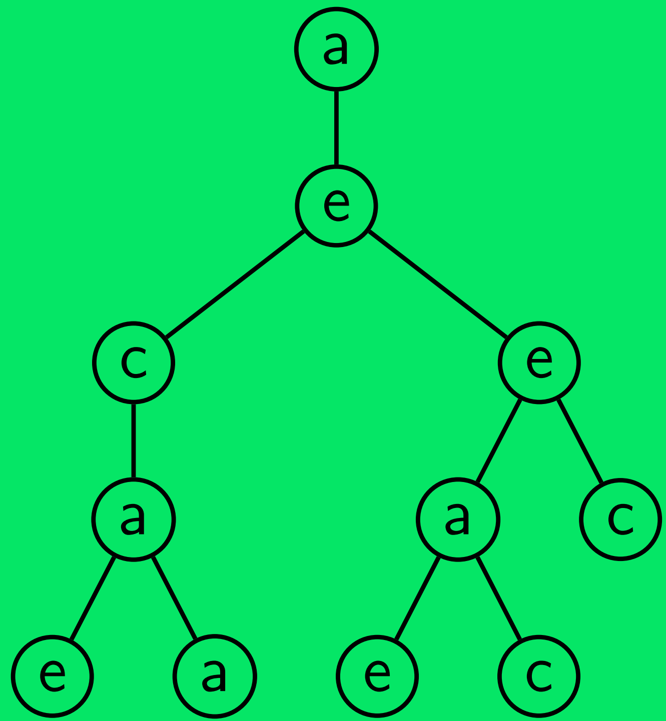
Question

How do string automata generalize to trees?

Sequential

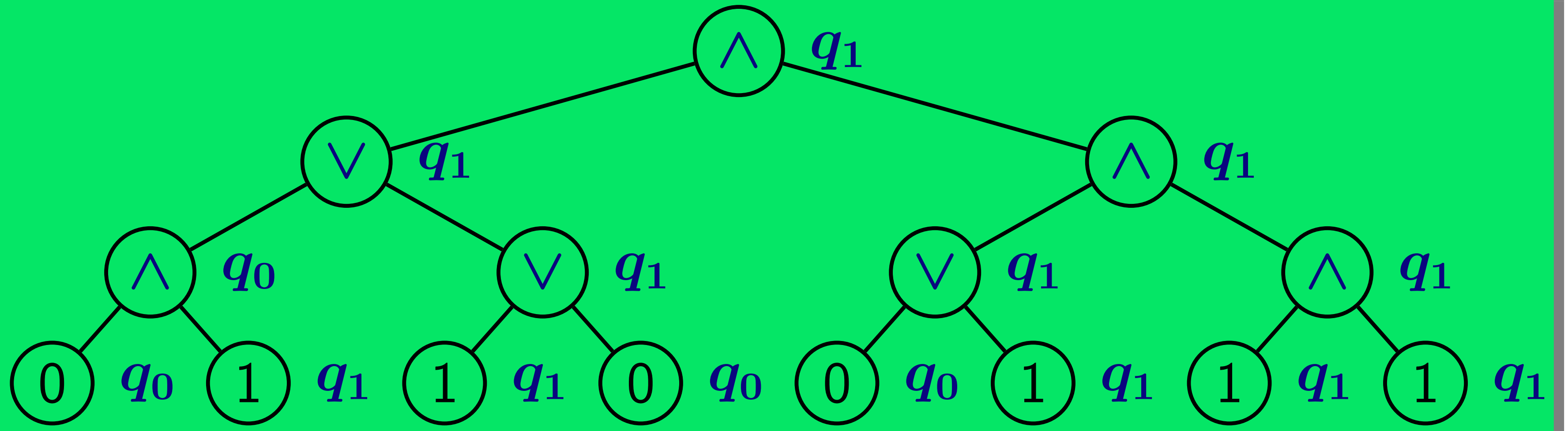


Parallel



Bottom-Up Automata

Example: Tree-structured Boolean Circuits



Idea

Tree-structured Boolean circuits

Two states: q_0, q_1

Accepting at the root: q_1

Transitions

$\delta(\wedge, q_1) = \{(q_1, q_1)\}$

$\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$

$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

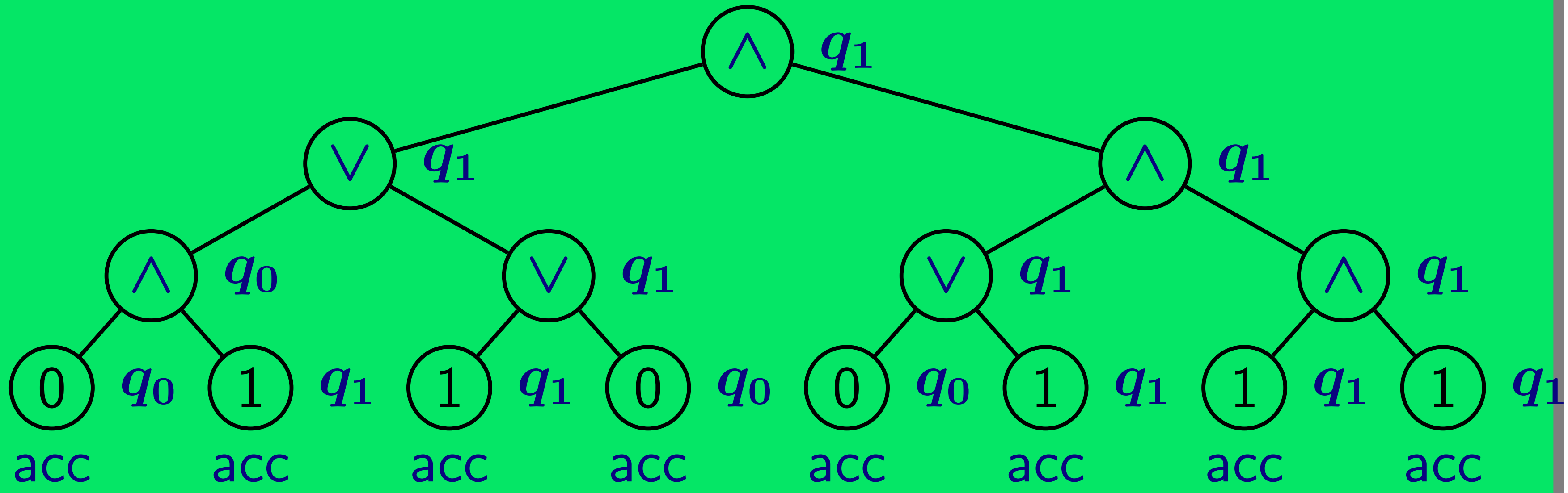
$\delta(\vee, q_0) = \{(q_0, q_0)\}$

$\delta(0, q_0) = \{\epsilon\}; \delta(0, q_1) = \emptyset$

$\delta(1, q_1) = \{\epsilon\}; \delta(1, q_0) = \emptyset$

Non-det. Top-Down Automata

Example



Idea

Guess the correct values starting from the root

Check at the leaves

Three states: q_0, q_1, acc

Initial state q_1 at the root

Accepting if all leaves end in acc

Transitions

$\delta(\wedge, q_1) = \{(q_1, q_1)\}$

$\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$

$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

$\delta(\vee, q_0) = \{(q_0, q_0)\}$

$\delta(0, q_0) = \{\text{acc}\}; \delta(0, q_1) = \emptyset$

$\delta(1, q_1) = \{\text{acc}\}; \delta(1, q_0) = \emptyset$

Regular Tree Languages

Definition

A bottom-up automaton is **deterministic** if

for each a and $p \neq q$: $\delta(a,p) \cap \delta(a,q) = \emptyset$

Theorem

The following are equivalent for a tree language L:

(a) L is accepted by a nondeterministic bottom-up automaton

(b) L is accepted by a deterministic bottom-up automaton

(c) L is accepted by a nondeterministic top-down automaton

Proof idea

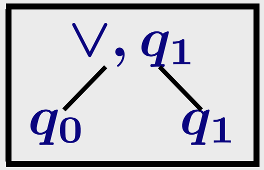
(a) $\Longrightarrow$ (b): Powerset construction

(a) $\Longleftrightarrow$ (c): Just the same thing, viewed in two different ways

Automata as Tiling Systems

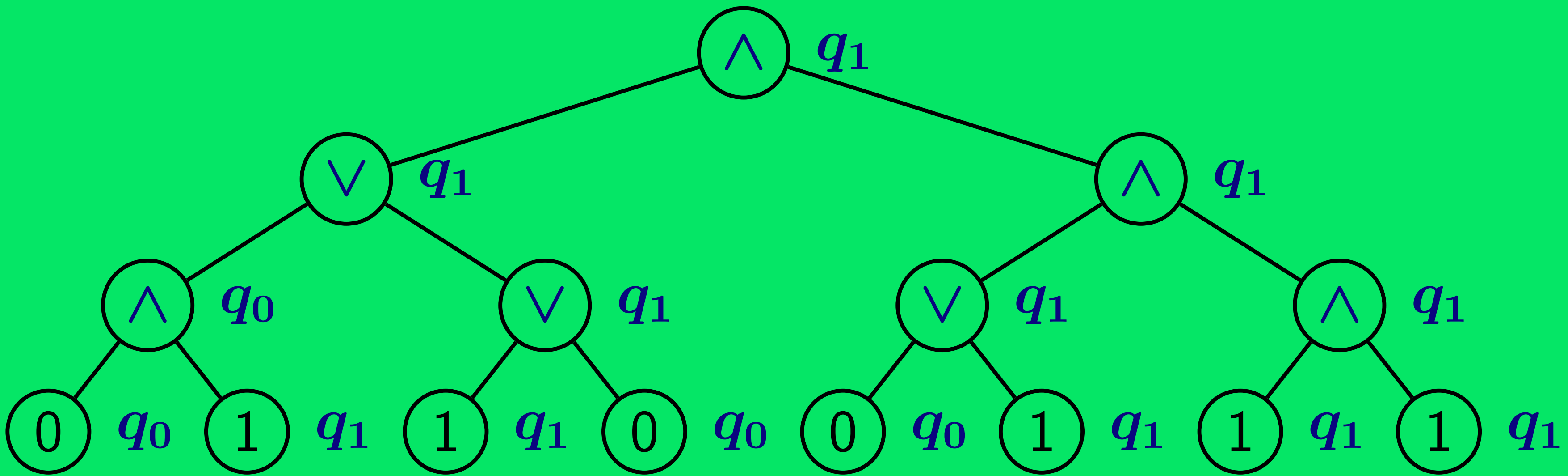
Observation

- $(q_0, q_1) \in \delta(\vee, q_1)$ can be interpreted as an allowed pattern:



- A tree is accepted, iff there is a labelling with states such that
  - all local patterns are allowed
  - the root is labelled with q_1

Example



Regular tree languages and logic

Definition: (MSO logic)

- **Formulas** talk about
  - edges of the tree (E)
  - node labels ($Q_0, Q_1, Q_\wedge, Q_\vee$)
  - the root of the tree (root)
- **First-order-variables** represent nodes
- **Monadic second-order** (MSO) variables represent sets of nodes

Example: Boolean Circuits

$$\begin{aligned}\text{Boolean circuit true} \quad \equiv \quad & \exists X\ X(\text{root}) \wedge \forall x \\ & (Q_0(x) \rightarrow \neg X(x)) \wedge \\ & ((Q_\wedge(x) \wedge X(x)) \rightarrow (\forall y[E(x,y) \rightarrow X(y)])) \wedge \\ & ((Q_\vee(x) \wedge X(x)) \rightarrow (\exists y[E(x,y) \wedge X(y)]))\end{aligned}$$

Theorem [Doner 70; Thatcher, Wright 68]

MSO $\equiv$ Regular Tree Languages

Regular tree languages and logic (cont.)

Theorem

MSO $\equiv$ Regular Tree Languages

Proof idea

Automata $\Rightarrow$ **MSO**:

Formula expresses that there exists a correct tiling

MSO $\Rightarrow$ **Automata**: more involved

Basic idea:

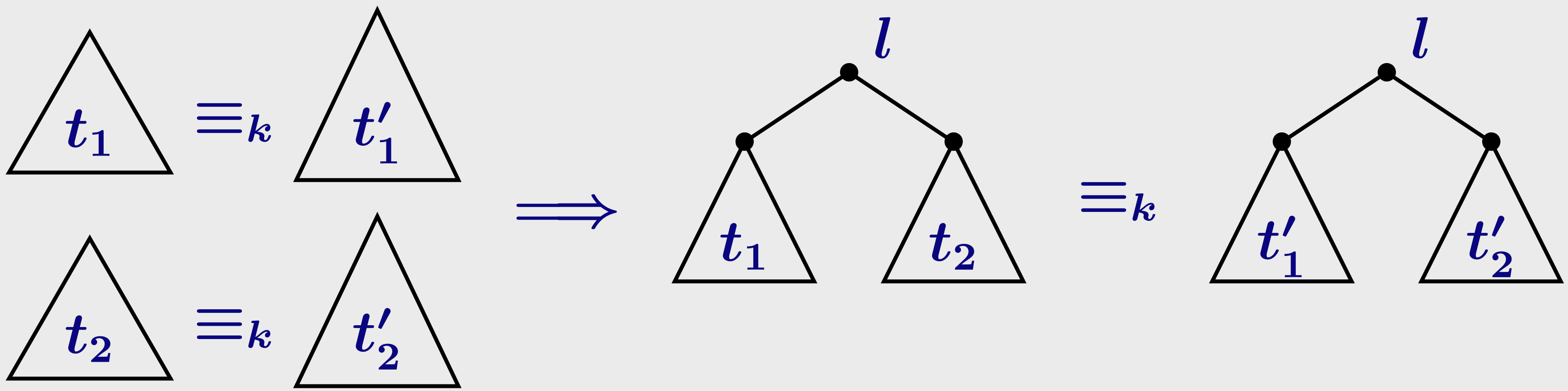
Automaton computes for each node v the set of formulas which hold in the subtree rooted at v

Regular tree languages and logic (cont.)

Formula $\Rightarrow$ automaton

- Let φ be an MSO-formula, $k :=$ quantifier-depth of φ
- k-type of a tree $t :=$ (essentially)
  set of MSO-formulas ψ of quantifier-depth $\leq k$ which hold in t
- $t_1 \equiv_k t_2$: k-type$(t_1) = k$-type(t_2)
- Automaton computes k-type of tree and concludes whether φ holds

Crucial fact



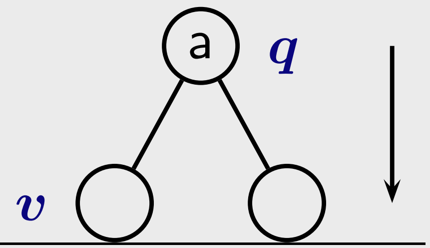
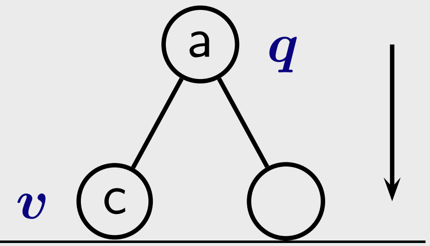
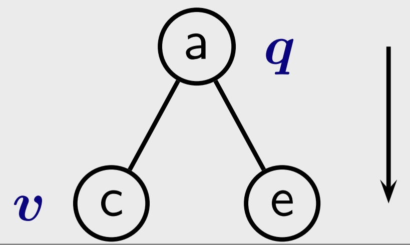
Det. Top-Down Automata

Question

What is the right notion for deterministic top-down automata?

3 Possibilities

State at a node v might depend on

state and symbol of parent	
state and symbol of parent and symbol of v	
state and symbol of parent and symbols at v and its sibling	

Det. Top-Down Automata: Acceptance

Question

What is a good acceptance mechanism for deterministic top-down automata?

Several possibilitites

(1) At all leaves states have to be accepting

(2) There is a leave with an accepting state

Observations

(2) is problematic for complement and intersection

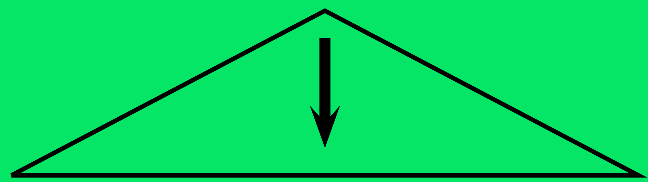
(1) is problematic for complement and union

Det. Top-Down Automata: Acceptance (cont.)

Definition: (Root-to-frontier automata with regular acceptance condition)

- Tree automata $\mathcal{A}$ are equipped with an additional regular string language L over $Q \times \Sigma$
- $\mathcal{A}$ accepts t if the (state,symbol)-string at the leaves (from left to right) is in L [Jurvanen, Potthoff, Thomas 93]

Illustration



(q_1, a_1) $\cdots$ (q_n, a_n)

A robust class

- The resulting class is closed under Boolean operations
- Good algorithmic properties
- Does not capture all regular tree languages

Summary

Regular tree languages

- Regular tree languages are a robust class
- Characterized by
  - parallel tree automata
  - MSO logic
  - several other models
- They are the natural analog of regular string languages
- Deterministic top-down automata with regular acceptance conditions define a weaker but nevertheless robust class

Contents

Tree-Walk Automata

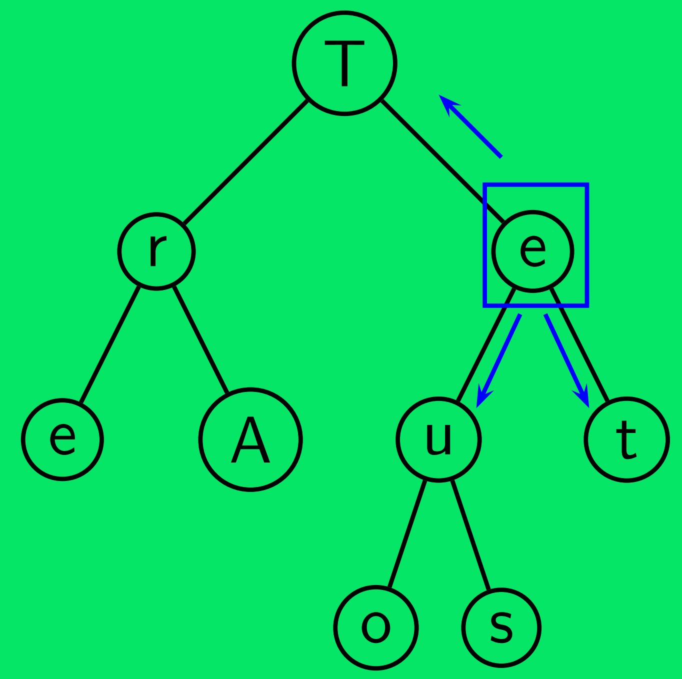
Definition: (Tree-walk automata)

Depending on

- current state
- symbol of current node
- position of current node wrt its siblings

the automaton moves to a neighbor and takes a new state

TWA



Question

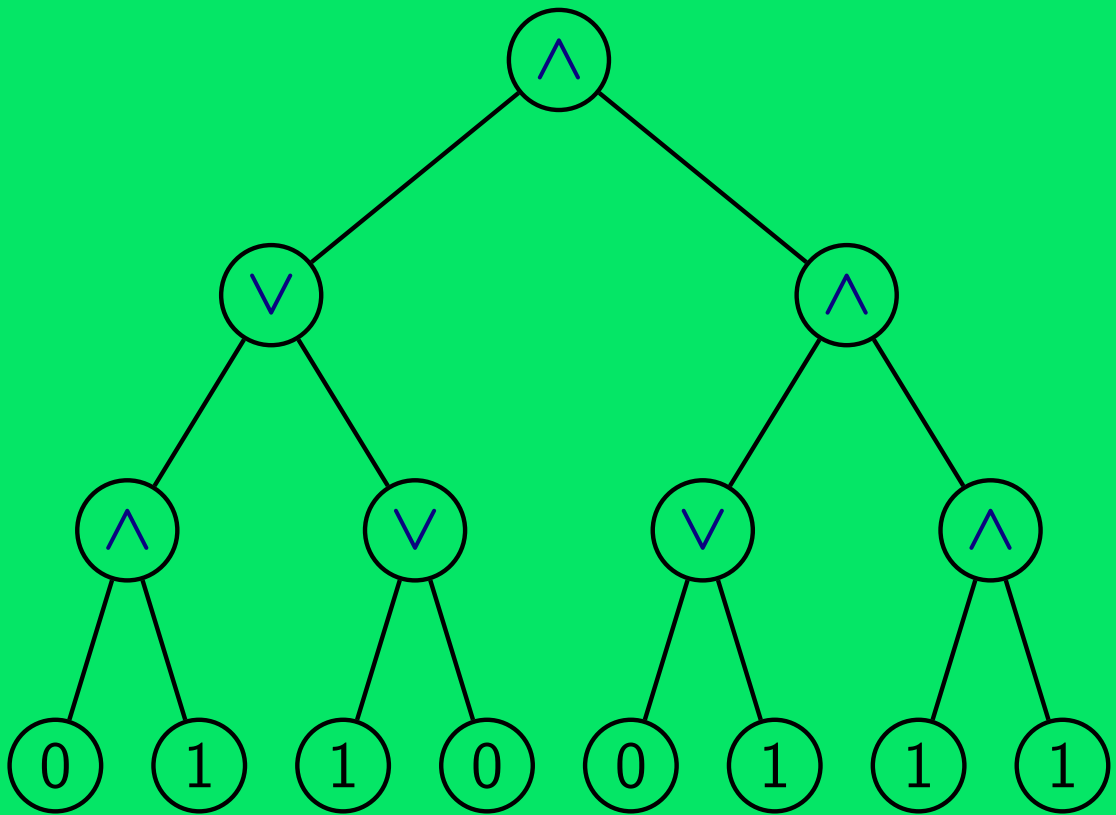
What is the expressive power of tree-walk automata?

Tree-Walk Automata (cont.)

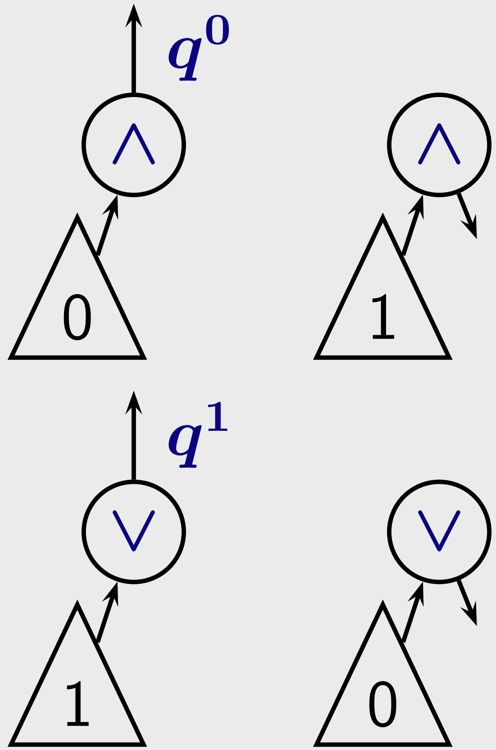
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



Idea



A Recent Result and an Even More Recent Result

Theorem [Bojanczyk, Colcombet 04]

Deterministic TWAs are weaker than nondeterministic TWAs

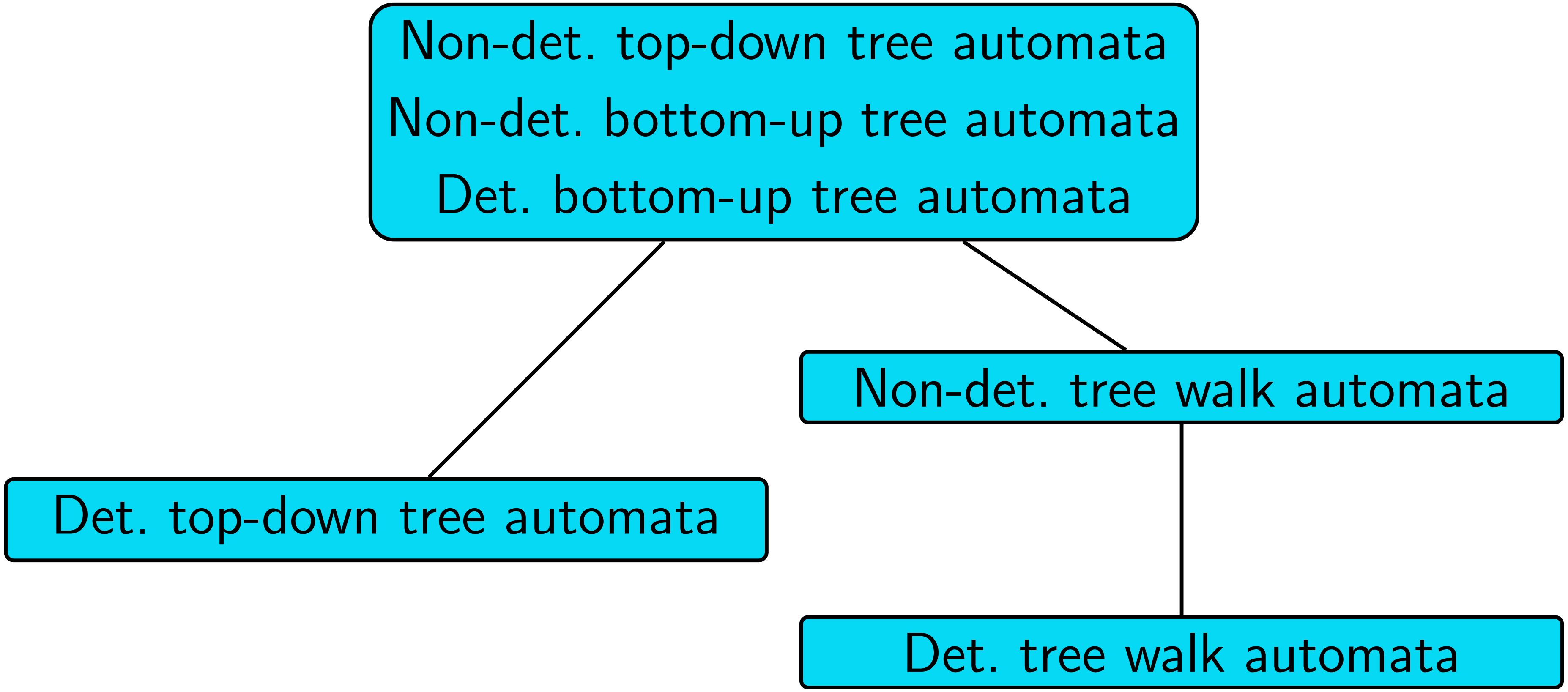
Corollary

Deterministic TWAs do not capture all regular tree languages

Theorem [Bojanczyk, Colcombet 04]

Nondeterministic TWAs do not capture all regular tree languages

Overview of Models



Contents

Decision Problems

Algorithmic problems

- We consider the following algorithmic problems
- All of them will be useful in the XML context

Membership test for $\mathcal{A}$

Given: Tree t

Question: Is $t \in L(\mathcal{A})$?

Membership test (combined)

Given: Automaton $\mathcal{A}$, tree t

Question: Is $t \in L(\mathcal{A})$?

Non-emptiness

Given: Automaton $\mathcal{A}$

Question: Is $L(\mathcal{A}) \neq \emptyset$?

Containment

Given: Automata $\mathcal{A}_1, \mathcal{A}_2$

Question: Is $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$?

Equivalence

Given: Automata $\mathcal{A}_1, \mathcal{A}_2$

Question: Is $L(\mathcal{A}_1) = L(\mathcal{A}_2)$?

Membership Test

Facts

Time Bounds for the combined complexity of membership test for tree automata:

- Deterministic (parallel) tree automata: $O(|\mathcal{A}||t|)$
- Nondeterministic (parallel) tree automata: $O(|\mathcal{A}|^2|t|)$
  (Compute, for each node, the set of reachable states)
- Deterministic TWAs: $O(|\mathcal{A}|^2|t|)$
  (Compute, for each node v, the aggregated behavior of $\mathcal{A}$ on its subtree: Behavior function)
- Nondeterministic TWAs: $O(|\mathcal{A}|^3|t|)$
  (Compute, for each node v, the aggregated behavior of $\mathcal{A}$ on its subtree: Behavior relation)

Membership Test (cont.)

Question: What is the structural complexity for the various models?

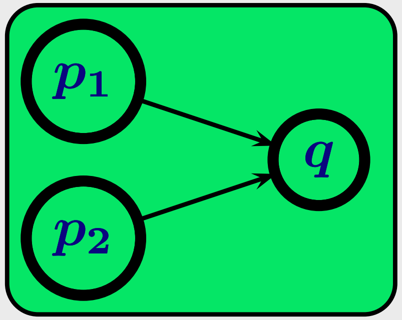
[Lohrey 01, Segoufin 03]

Model	Time Complexity	Structural Complexity
Det. top-down TA	$O(\|\mathcal{A}\|\|t\|)$	**LOGSPACE**
Det. bottom-up TA	$O(\|\mathcal{A}\|\|t\|)$	**LOGDCFL**
Nondet. bottom-up TA	$O(\|\mathcal{A}\|^2\|t\|)$	**LOGCFL**
Nondet. top-down TA	$O(\|\mathcal{A}\|^2\|t\|)$	**LOGCFL**
Det. TWA	$O(\|\mathcal{A}\|^2\|t\|)$	**LOGSPACE**
Nondet. TWA	$O(\|\mathcal{A}\|^3\|t\|)$	**NLOGSPACE**

Non-emptiness

Facts

- Non-emptiness for string automata corresponds to `Graph Reachability` (complete for **NLOGSPACE**)
- Non-emptiness for tree automata corresponds to `Path Systems` :



Result

- Non-emptiness for bottom-up tree automata can be checked in linear time
- It is complete for **PTIME**

Containment/Equivalence

Observations

- Of course:

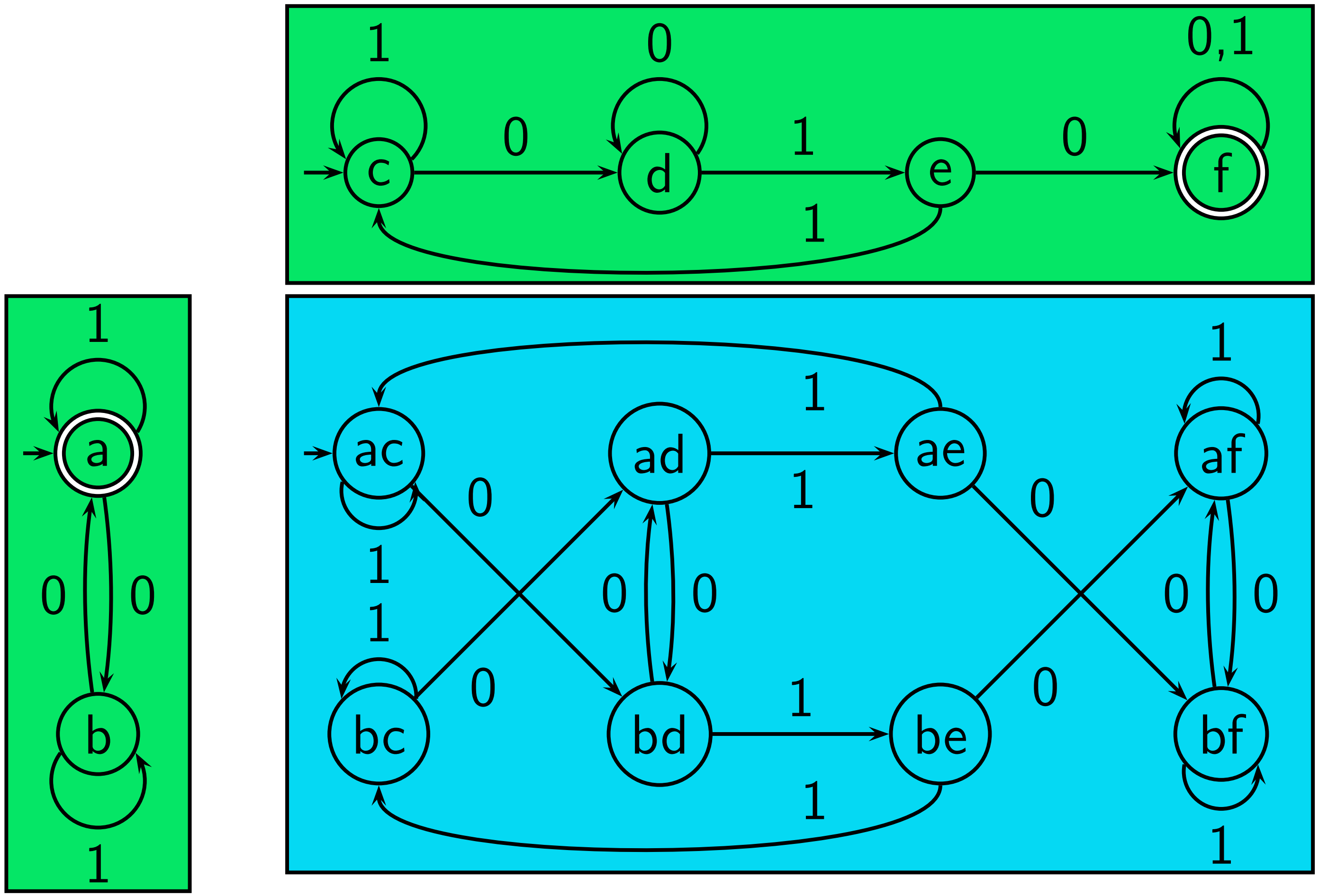
$$L(\mathcal{A}_1) = L(\mathcal{A}_2) \iff [L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2) \text{ and } L(\mathcal{A}_2) \subseteq L(\mathcal{A}_1)]$$

- Complexity of containment problem is very different for deterministic and non-deterministic automata
- For deterministic automata: construct product automaton

Reminder: Product automaton

Product of 2 string automata

- "even number of zeros"
- "contains substring 00"



0110100

Containment: Complexity

Deterministic bottom-up tree automata

- Product automaton analogous as for string automata
  - Set of states: $Q_1 \times Q_2$
  - Transitions component-wise
- To check $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$:
  - Compute $\mathcal{B} = \mathcal{A}_1 \times \mathcal{A}_2$
  - Accepting states: $F_1 \times (Q_2 - F_2)$
  - Check whether $L(\mathcal{B}) = \emptyset$
  - If so, $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$ holds

Theorem

Complexity of Containment for deterministic bottom-up tree automata:

$$O(|\mathcal{A}_1| \times |\mathcal{A}_2|)$$

Containment: Complexity (cont.)

Non-deterministic automata

- Naive approach:
  - Make $\mathcal{A}_2$ deterministic (size: $O(2^{|\mathcal{A}_2|})$)
  - Construct product automaton

  $\Rightarrow$ Exponential time

Unfortunately...

There is essentially no better way

Theorem [Seidl 1990]

Containment for non-deterministic tree automata is complete for **EXPTIME**

Det. Top-Down Automata: Non-Emptiness

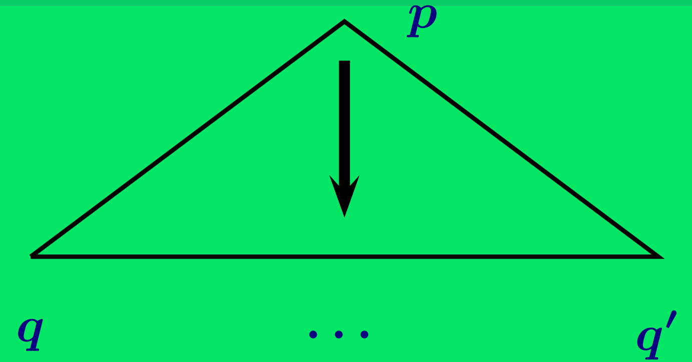
Theorem

Nonemptiness for deterministic top-down automata $\mathcal{A}$ can be checked in polynomial time

Proof idea

Check for each state p of $\mathcal{A}$ and each pair (q, q') of the leaves automaton $\mathcal{B}$:
Is there a tree t such that $\mathcal{A}$ starts from state p and obtains a leave string which brings $\mathcal{B}$ from q to q'?

Illustration



Det. Top-Down Automata: Containment

Theorem

Containment for deterministic top-down automata $\mathcal{A}$ can be checked in polynomial time

Proof idea

- Tree automata $\mathcal{A}_1$, $\mathcal{A}_2$ with leaves automata $\mathcal{B}_1, \mathcal{B}_2$
- Check
  - for each pair (p_1, p_2) of states of $\mathcal{A}_1$ and $\mathcal{A}_2$ and
  - for each two pairs (q_1, q_1') and (q_2, q_2') of $\mathcal{B}_1$ and $\mathcal{B}_2$, resp.:

  Is there a tree t such that for both $i = 1$, $i = 2$: $\mathcal{A}_i$ starts from state p_i and obtains a leave string which brings $\mathcal{B}_i$ from q_i to q_i'?

Summary

Complexities of basic algorithmic problems

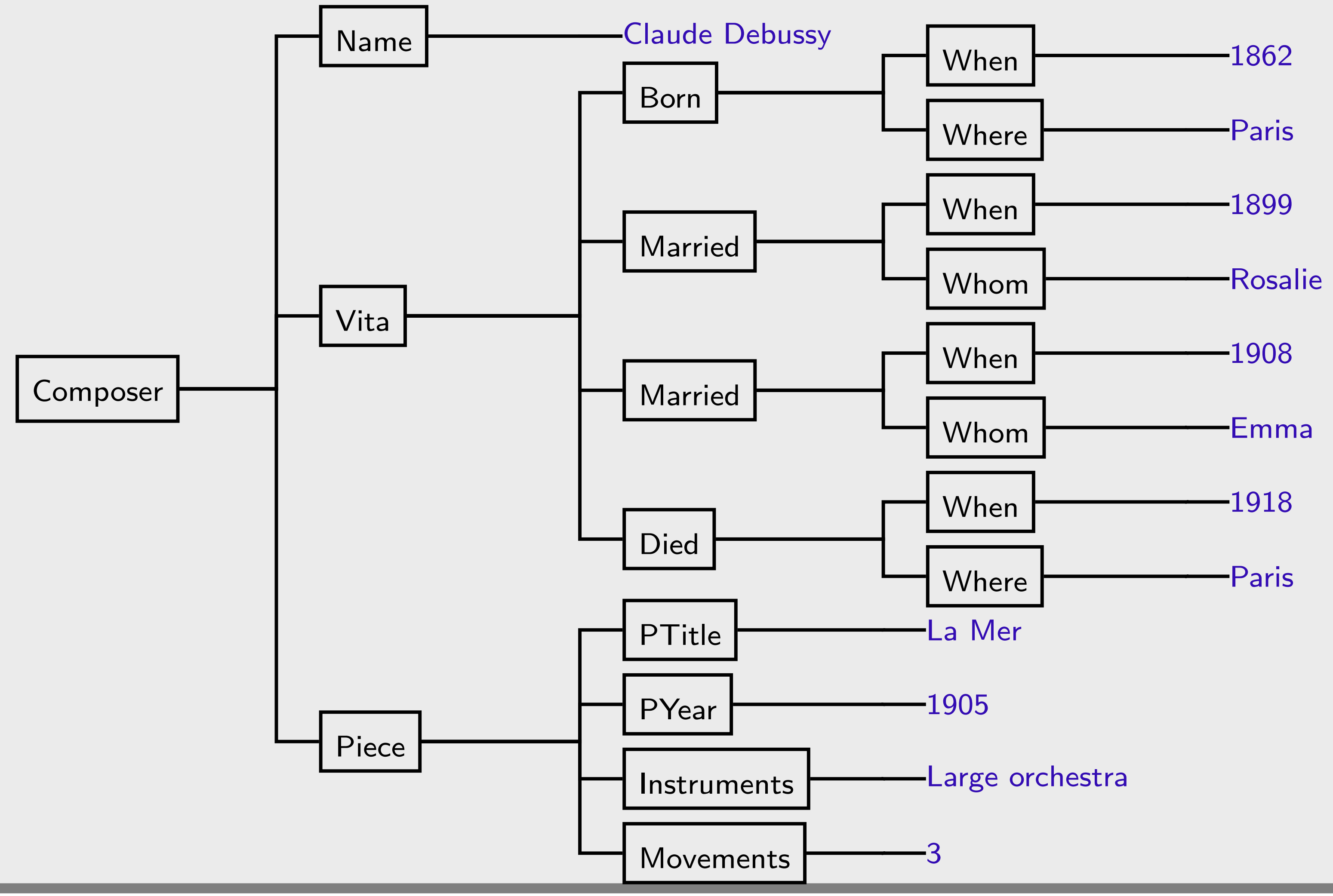
Model	Membership	Non-emptiness	Containment
Det. top-down TA	**LOGSPACE**	**PTIME**	**PTIME**
Det. bottom-up TA	**LOGDCFL**	**PTIME**	**PTIME**
Nondet. bottom-up TA	**LOGCFL**	**PTIME**	**EXPTIME**
Nondet. top-down TA	**LOGCFL**	**PTIME**	**EXPTIME**
Det. TWA	**LOGSPACE**	**PTIME** ($*$)	**PTIME** ($*$)
Nondet. TWA	**NLOGSPACE**	**PTIME** ($*$)	**EXPTIME** ($*$)

($*$: upper bounds)

Contents

From Ranked to Unranked Trees

Example Tree



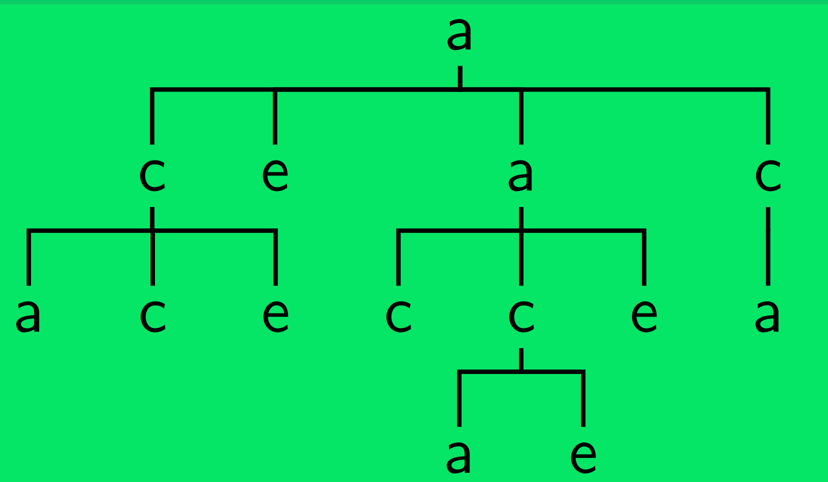
From Ranked to Unranked Trees

Agenda

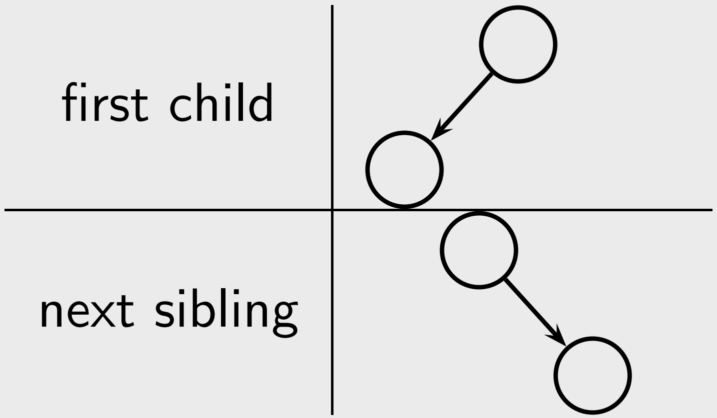
- Now we move from ranked to unranked trees
- There is a basic choice:
  - Either: we encode unranked trees as binary trees and go on with ranked automata
  - Or: we adapt the ranked automata models
- In both cases: not many surprises, most results remain

Encoding Unranked Trees as Binary Trees

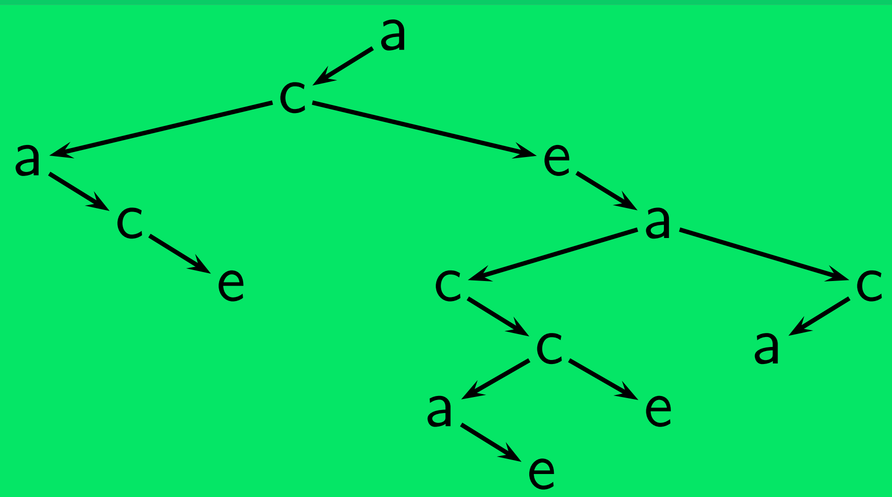
Example: Unranked Tree



Encoding via ...

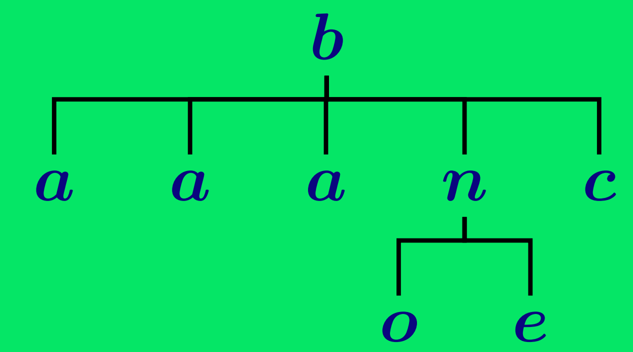


... as Binary Tree

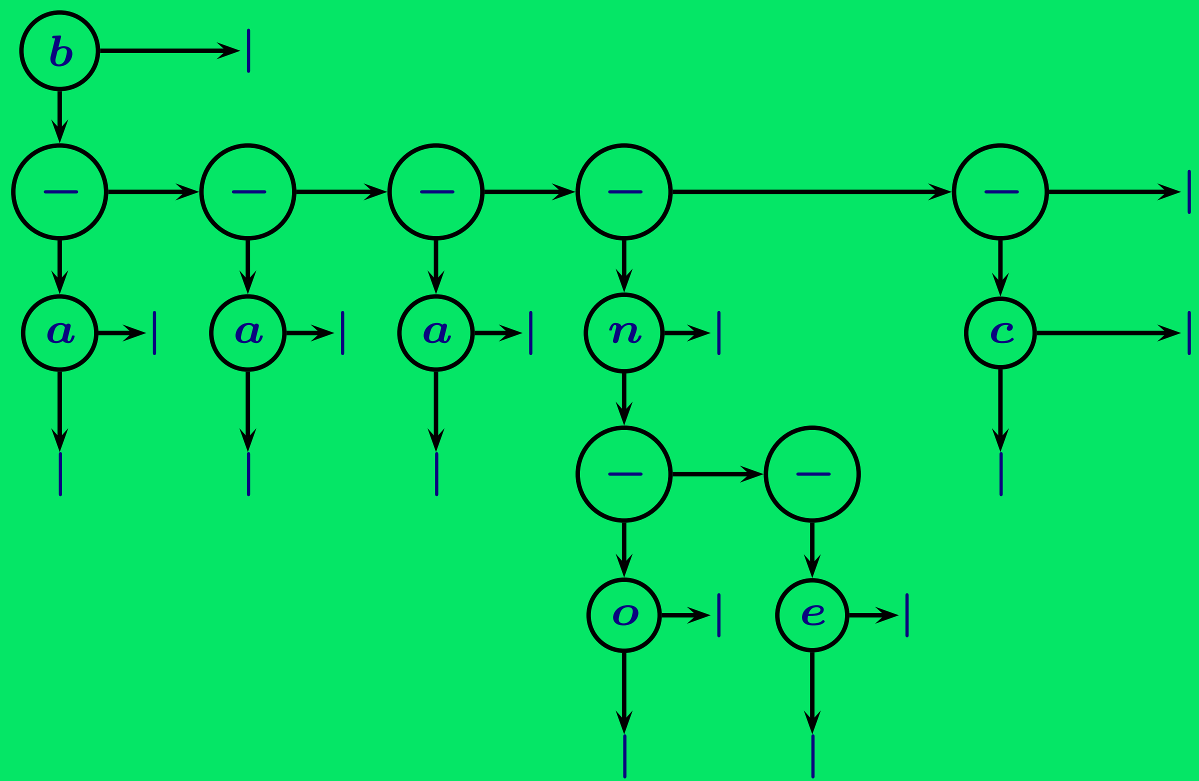


Encoding Unranked Trees as Binary Trees (cont.)

Example: Unranked Tree



... if path expressions matter (Milo,Suciu,Vianu 00)



Binary vs. unranked trees

Remark

- There are still other ways to encode unranked trees as binary trees

$\longrightarrow$ e.g., [Carme, Niehren, Tommasi 04]

- We consider automata for unranked trees next

Unranked Trees: Formal Definition

Definition

A **(finite) tree domain** V over $\mathbb{N}$ is a (finite) subset of $\mathbb{N}^*$, such that if $v \cdot i \in V$, where $v \in \mathbb{N}^*$ and $i \in \mathbb{N}$,

- then $v \in V$
- and $v \cdot (i-1) \in V$, if $i > 1$

Note

ε represents the root

Definition

A **labelled tree t** is a pair (V, λ), where V is a tree domain over $\mathbb{N}$, and λ is a function from V to the set Σ of labels.

Remark

XML tags can be captured by the set Σ of labels. But what about text?

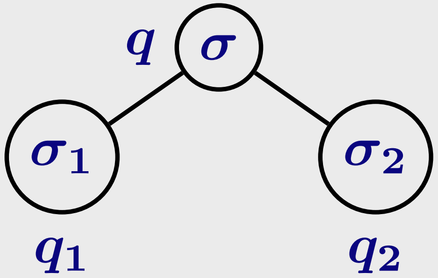
- This depends on the context
- E.g., for type checking, text is irrelevant.
- In many applications, the relevant information about text nodes can be represented by predicates, e.g., whether the `name` $=$ `'Debussy'`.

From Ranked to Unranked Tree Automata

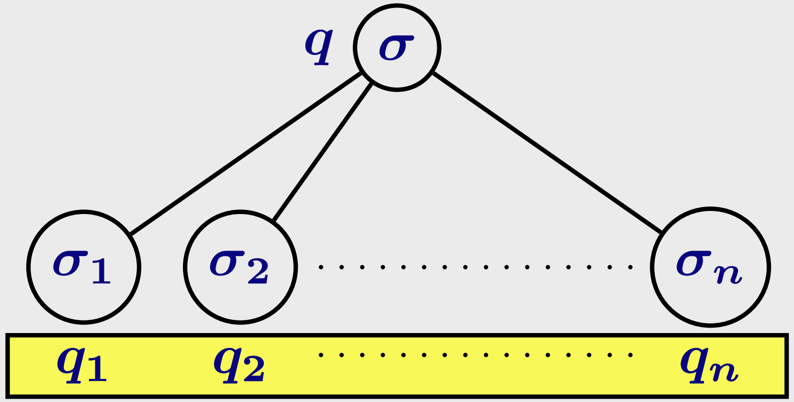
Ranked trees

Transitions are described by finite sets:

$\delta(\sigma, q) = \{(q_1, q_2), (q_3, q_4), \ldots\}$



Unranked trees



$\in \delta(\sigma, q)$?

$\delta(\sigma, q)$

- For unranked trees, $\delta(\sigma, q)$ is a regular language
- $\delta(\sigma, q)$ can be specified by regular expression or finite string automaton [Brüggemann-Klein, Murata, Wood 2001]

Representation of $\delta(\sigma, q)$

Remark

- Representation of $\delta(\sigma, q)$ has influence on complexity
- Natural choice:
  - For nondeterministic tree automata:
    represent $\delta(\sigma, q)$ by NFAs or regular expressions
  - For deterministic tree automata:
    represent $\delta(\sigma, q)$ by DFAs

$\Rightarrow$ Same complexity results as for ranked trees

Regular sets of unranked trees

Theorem

The following are equivalent for a set L of unranked trees:

(a) L is accepted by a nondeterministic bottom-up automaton

(b) L is accepted by a deterministic bottom-up automaton

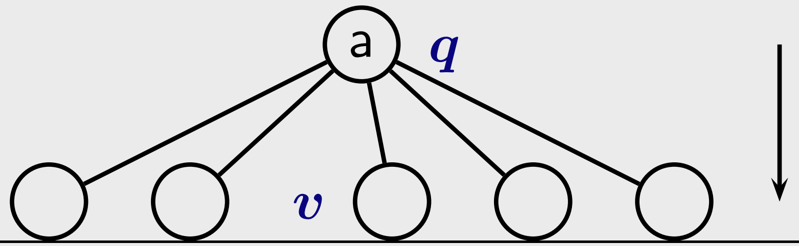
(c) L is accepted by a nondeterministic top-down automaton

(d) L is characterized by an MSO-formula

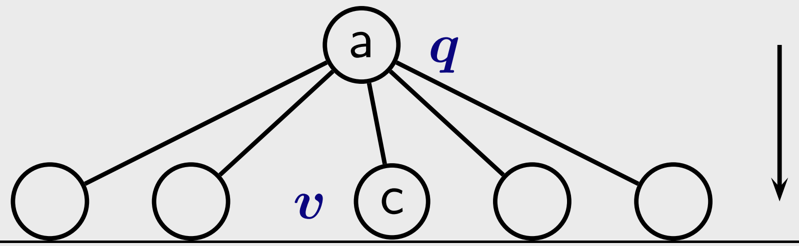
Deterministic Top-Down Automata

State at v might depend on ...

state and symbol of parent

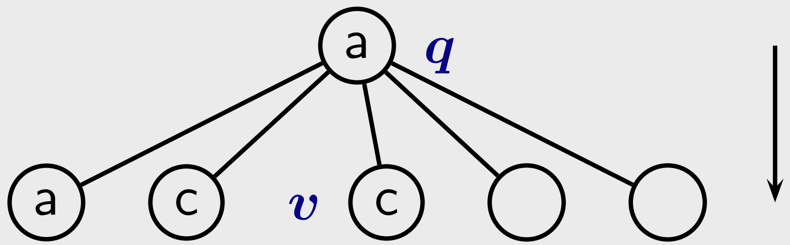


state and symbol of parent and symbol of v



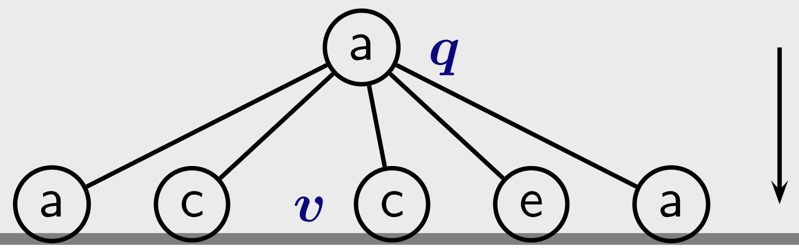
simple

state and symbol of parent and symbols at v and its left siblings



left-siblings aware

state and symbol of parent and symbols at v and its siblings



Checking Existence of Paths

Fact

A simple deterministic top-down automaton can check the existence of vertical paths with regular properties

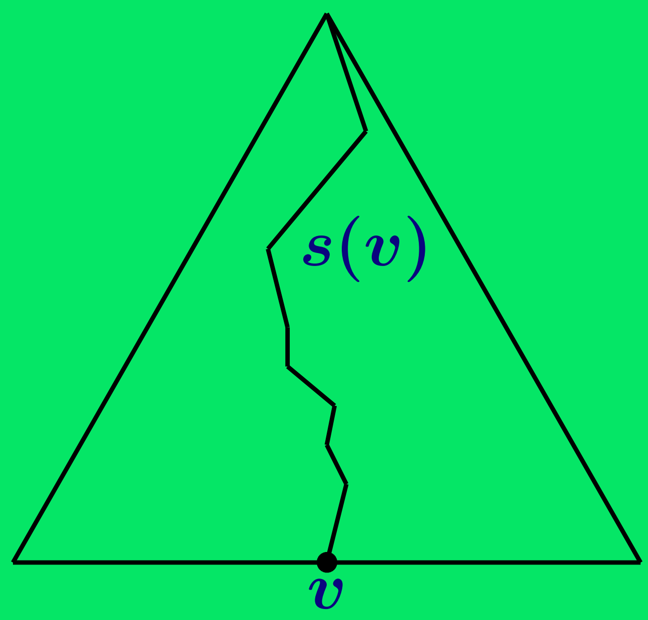
Construction

- For a node v let $\boxed{s(v)}$ denote the sequence of labels from the root to v
- Let $\mathcal{A}$ be a deterministic string automaton
- $\mathcal{A}' :=$ top-down automaton which takes at v state of $\mathcal{A}$ after reading $s(v)$

$\Rightarrow$ $\mathcal{A}'$ is deterministic

- There is a path from the root to a leaf v with $s(v) \in L(\mathcal{A})$ iff $\mathcal{A}'$ assumes at least one state from F at a leave

Illustration



Streaming XML

Similar construction used for `XPath` evaluation on streams [Green et al. 2003]

Contents

Sequential Automata on Unranked Trees

Generalization of Tree-Walk Automata

Allowed transitions: `Go up`

`Go to first child`

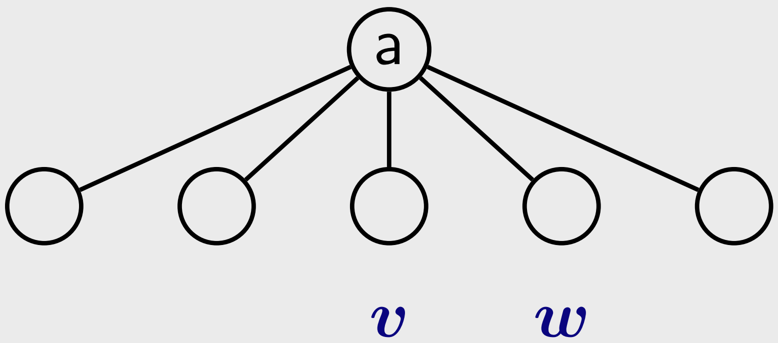
`Go to left sibling`

`Go to right sibling`

$\longrightarrow$ Caterpillar automata [Brüggemann-Klein, Wood 2000]

Basic design choice

Should a transition to a sibling be aware of the label of the parent?



Contents

Document Automata

A third kind of automata for XML

- **Document automata** are string automata reading XML documents as text
- Tags are represented by symbols from a given alphabet
- Variants:
  - Finite document automata
  - Pushdown document automata
- Useful especially in the context of streaming XML

Theorem [Segoufin, Vianu 02]

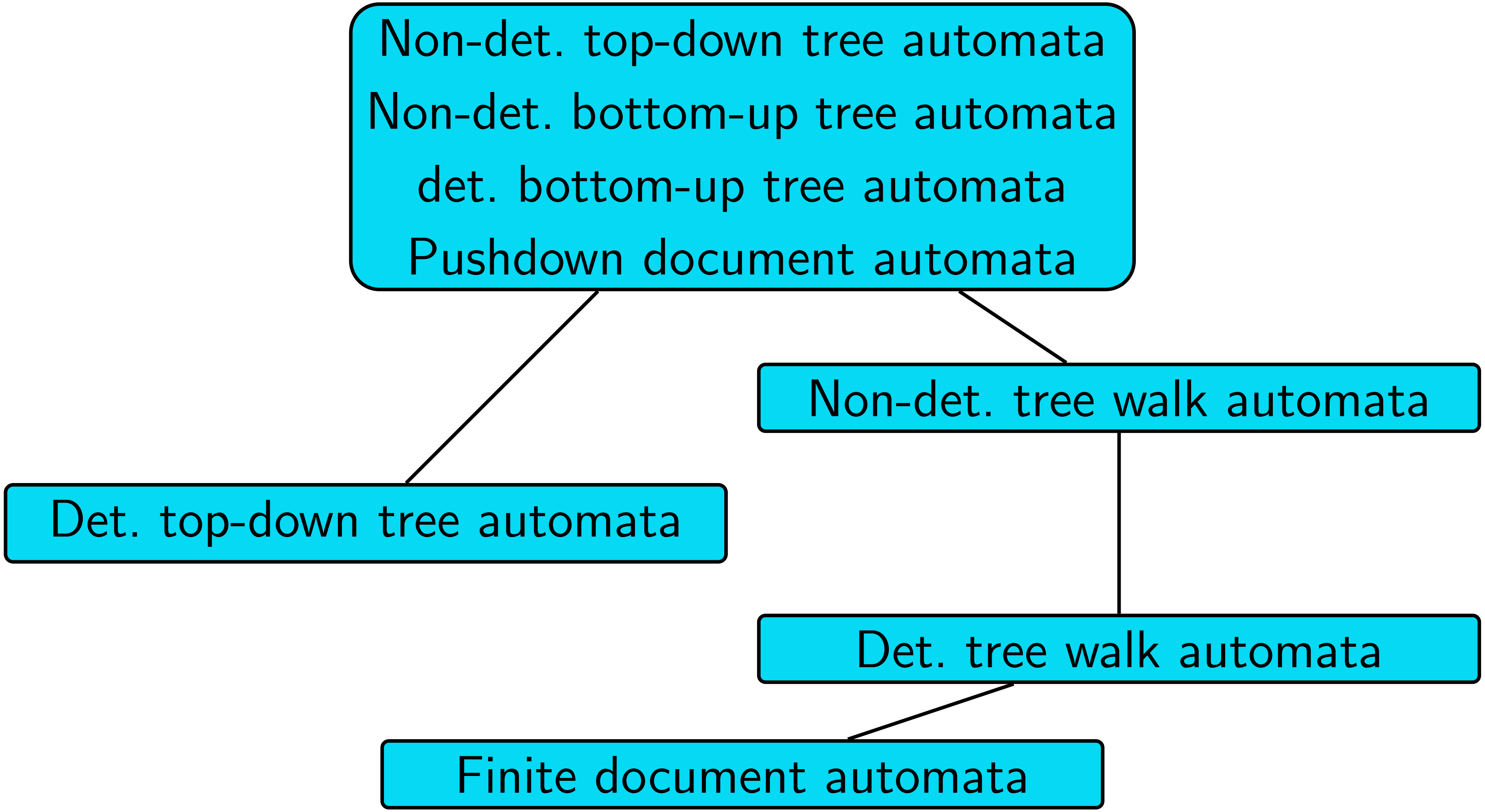
- Regular languages of XML-trees can be recognized by deterministic push-down document automata.
- Depth of push-down is bounded by depth of tree

Summary: Unranked Tree Automata

Summary

- Moving from ranked to unranked automata requires some adaptations
- Transitions can be defined with regular string languages $\delta(\sigma, q)$
- By and large, things work smoothly
- In particular:
  - there is an equally robust notion of regular tree languages
  - The complexities are the same as for ranked automata (if the sets $\delta(\sigma, q)$ are represented in a sensible way)

Refined Overview of Models



Contents

DTDs

Example DTD

```
<!DOCTYPE Composers [
    <!ELEMENT Composers (Composer*)>
    <!ELEMENT Composer (Name, Vita, Piece*)>
    <!ELEMENT Vita (Born, Married*, Died?)>
    <!ELEMENT Born (When, Where)>
    <!ELEMENT Married (When, Whom)>
    <!ELEMENT Died (When, Where)>
    <!ELEMENT Piece (PTitle, PYear,
         Instruments, Movements)>
    ]>
```

Some Facts

- DTDs $\equiv$ generalized context-free grammars

$\longrightarrow$ [Berstel,Boasson 00] provide characterizations

- Additional restriction: one-unambiguous

One-unambiguous Regular Expressions

Definition: One-unambiguous Regular Expression

- Let r be a regular expression
- $r \mapsto r'$: number the symbols of r from left to right
- $w \in L(r) \longleftrightarrow$ there is a numbered string $w' \in L(r')$
- r is **one-unambiguous** if

$$ux_iv \in L(r'),\ uy_jw \in L(r'),\ i \neq j \Rightarrow x \neq y$$

Example

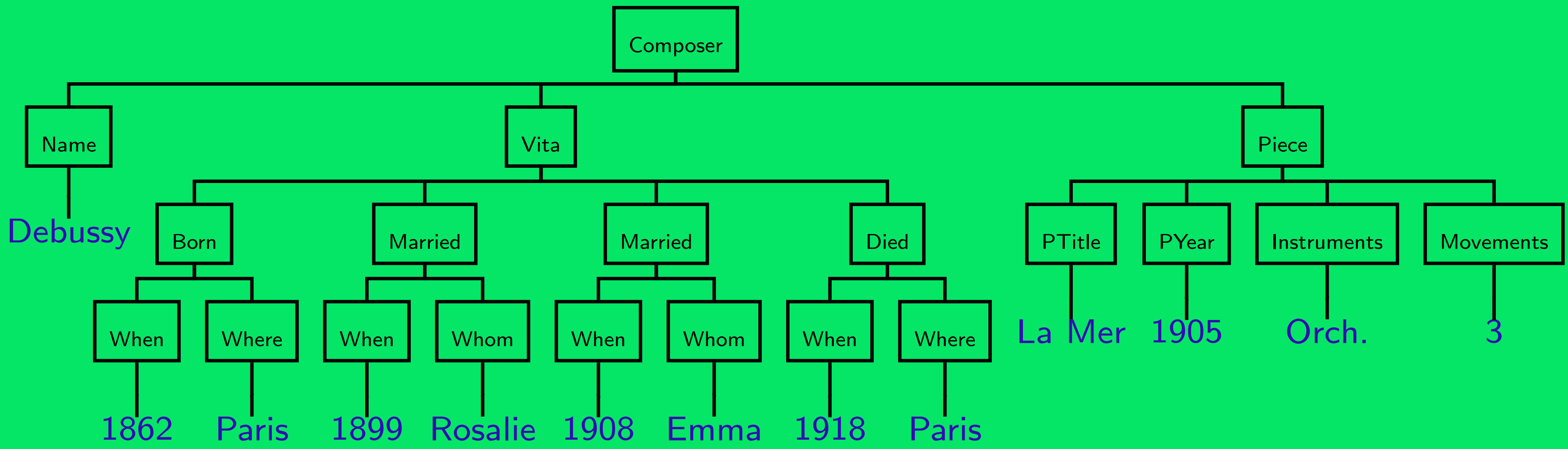
- $(a+b)^*ac+c \mapsto (a_1+b_2)^*a_3c_4+c_5$
- $babbac \in L(r)$ and $b_2a_1b_2b_2a_3c_4 \in L(r')$
- $(a+b)^*ac+c$ is not one-unambiguous because $b_2b_2\boxed{a_3}c_4 \in L(r')$ and $b_2b_2\boxed{a_1}a_3c_4 \in L(r')$
- $(b^*a)^*c$ is one-unambiguous

Restriction

- Expressions in DTDs have to be one-unambiguous
- Inherited from SGML

Validation wrt a DTD

Example Tree



Example DTD

```
<!DOCTYPE Composers [
    <!ELEMENT Composers (Composer*)>
    <!ELEMENT Composer (Name, Vita, Piece*)>
    <!ELEMENT Vita (Born, Married*, Died?)>
    <!ELEMENT Born (When, Where)>
    <!ELEMENT Married (When, Whom)>
    <!ELEMENT Died (When, Where)>
    <!ELEMENT Piece (PTitle, PYear,
         Instruments, Movements)>
    ]>
```

Validation Algorithm

For each node:

Check that the children are ok wrt the parent's rule

Validation wrt a DTD (cont.)

Observation

- Validation wrt DTDs is a very simple task
- Can be done by
  - Bottom-up automata
  - Deterministic top-down automata (if siblings contribute to new state)
  - Deterministic tree-walk automata: Just a depth-first left-to-right traversal
- In particular: Validation possible in linear time during one pass through the document (1-pass validation)

DTDs: Satisfiability

Example

$$\begin{array}{ccc} a & \longrightarrow & bc \\ b & \longrightarrow & cd \\ c & \longrightarrow & \epsilon \\ d & \longrightarrow & \epsilon \end{array}$$

Fact

Satisfiability for DTDs is complete for **PTIME**

Containment of DTDs

Lemma [Martens, Neven, Sch. 04]

Containment of DTDs with regular expressions from $\boldsymbol{R}$ is in C

$$\Longleftrightarrow$$

Containment of regular expressions from $\boldsymbol{R}$ is in C

Corollary

Containment of DTDs (with one-unambiguous regular expressions) is in **PTIME**

Proof sketch

- One-unambiguous regular expressions have deterministic automata of linear size

$\Rightarrow$ Containment of regular expressions $\boldsymbol{r_1, r_2}$ by product automaton of size $\boldsymbol{O(|r_1||r_2|)}$

Containment of DTDs (cont.)

Question

What if the requirement of being one-unambiguous is dropped?

A classical result

Theorem [Stockmeyer, Meyer 71]

Containment and Equivalence for regular expressions on strings are complete for **PSPACE**

Corollary

Containment of DTDs (with unrestricted regular expressions) is **PSPACE**-complete

Theorem [Martens, Neven, Sch. 04]

Containment and Equivalence for regular expressions are

- **coNP**-complete for concatenations of a, b, c and a^*, b^*, c^*
- **coNP**-complete for concatenations of a, b, c and $a?, b?, c?$
- **PSPACE**-complete for concatenations of a, b, c and $(a^* + b^* + \cdots + c^*)$

Contents

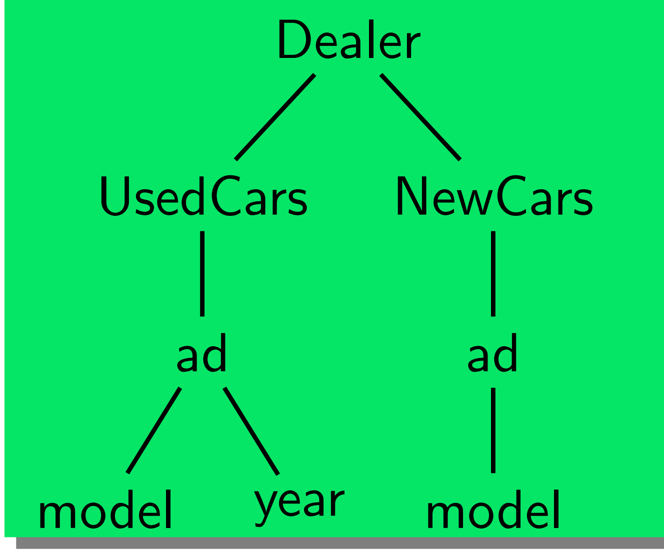
Weakness of DTDs

A classical example

```
<!DOCTYPE Dealer [
    <!ELEMENT Dealer (UsedCars NewCars)>
    <!ELEMENT UsedCars (ad*)>
    <!ELEMENT NewCars (ad*)>
    <!ELEMENT ad ((model, year) | model)> ]>
```

Intention

Intention:



Observation

- Elements with the same name may have different structure in different contexts

$\longrightarrow$ It would be nice to have types for elements

$\longrightarrow$ **Specialized DTDs**

Specialized DTDs

Definition: [Papakonstantinou, Vianu 2000]

A **specialized DTD** (SDTD) over alphabet Σ is a pair (d, μ), where

- d is a DTD over the alphabet Σ' of **types**
- $\mu : \Sigma' \longrightarrow \Sigma$ maps types to tag names

Note

Concerning the name:

"specialized" refers to types, not to DTDs

Example

Dealer $\longrightarrow$ UsedCars NewCars	μ(Dealer) $=$ Dealer
UsedCars $\longrightarrow$ adUsed*	μ(UsedCars) $=$ UsedCars
NewCars $\longrightarrow$ adNew*	μ(NewCars) $=$ NewCars
adUsed $\longrightarrow$ model year	μ(adUsed) $=$ ad
adNew $\longrightarrow$ model	μ(adNew) $=$ ad

A Further Example

Example: SDTD for Boolean circuit trees

1-AND	$\longrightarrow$	(1-OR | 1-AND | 1-leaf)*
1-OR	$\longrightarrow$	.* (1-OR | 1-AND | 1-leaf) .*
0-AND	$\longrightarrow$	.* (0-OR | 0-AND | 0-leaf) .*
0-OR	$\longrightarrow$	(0-OR | 0-AND | 0-leaf)*
1-leaf	$\longrightarrow$	ϵ
0-leaf	$\longrightarrow$	ϵ

Tag	μ(Tag)
1-AND	AND
0-AND	AND
1-OR	OR
0-OR	OR
1-leaf	1
0-leaf	0

Specialized DTDs (cont.)

Observation

- A naive validation by exhaustively trying all possible functions μ requires exponential time
- But help comes from automata...
- A tree conforms to a specialized DTD (d, μ) if there is a labeling of its nodes by types which is valid wrt. d
- This reminds us of something...

Theorem

Specialized DTDs capture exactly the regular tree languages

Validation and Typing

Definition: Validation

Given: Specialized DTD d, tree t

Qeustion: Is t valid wrt d?

Definition: Typing

Given: Specialized DTD d, tree t

Output: Consistent type assignment for the nodes of t

Facts

- Specialized DTDs $\equiv$ regular tree languages

$\longrightarrow$ Validation in linear time by deterministic push-down automata

- Typing in linear time (Bottom-up automaton)
- Satisfiability $\equiv$ Non-emptiness of tree automata: **PTIME**

Restrictions of Schemas

Restricted Schemas

(Murata, Lee, Mani 2001) introduced* restrictions on specialized DTDs to ensure efficient validation (∗: in a slightly different framework)

- Two types b, b' **compete** if $\mu(b) = \mu(b')$
- A specialized DTD is **single-type** if no competing types occur in the same rule (e.g., $a \rightarrow bcb'$ is not single-type)
- A specialized DTD is **restrained-competition** if no rule allows strings wbv, $wb'v'$ with competing types b, b'
  (e.g., $a \rightarrow c(b + d^*b')$ is not restrained-competition)
- The authors argue that XML-Schema roughly corresponds to single-type SDTDs

Schema Containment

Schema Containment

Given: Schemas d_1, d_2

Question: Is $L(d_1) \subseteq L(d_2)$?

Observations

- Important, e.g., for data integration
- Recall: Specialized DTDs are essentially non-deterministic tree automata

$\Rightarrow$ Containment of specialized DTDs is in **EXPTIME**

- But the restricted forms have lower complexity
- Complexity of containment depends on the allowed regular expressions

Schema Containment: Complexity

Results (partly from [Martens, Neven, Sch. 04])

Schema type	unrestricted	deterministic expressions
DTDs	**PSPACE**	**PTIME**
single-type SDTDs	**PSPACE**	**PTIME**
restrained-competition SDTDs	**PSPACE**	**PTIME**
unrestricted SDTDs	**EXPTIME**	**EXPTIME**

Observations

- For unrestricted SDTDs the complexity is dominated by tree automata containment
- For the others it is dominated by the sub-task of checking containment for regular expressions

Schema Containment: Complexity

Observations (cont.)

- ... for the others it is dominated by the sub-task of checking containment for regular expressions
- Actually, this observation can be made more precise

Theorem [Martens, Neven, Sch. 04]

For a class $\mathcal{R}$ of regular expressions and a complexity class $\mathcal{C}$, the following are equivalent

(a) The containment problem for $\mathcal{R}$ expressions is in $\mathcal{C}$.

(b) The containment problem for DTDs with regular expressions from $\mathcal{R}$ is in $\mathcal{C}$.

(c) The containment problem for single-type SDTDs with regular expressions from $\mathcal{R}$ is in $\mathcal{C}$.

Contents

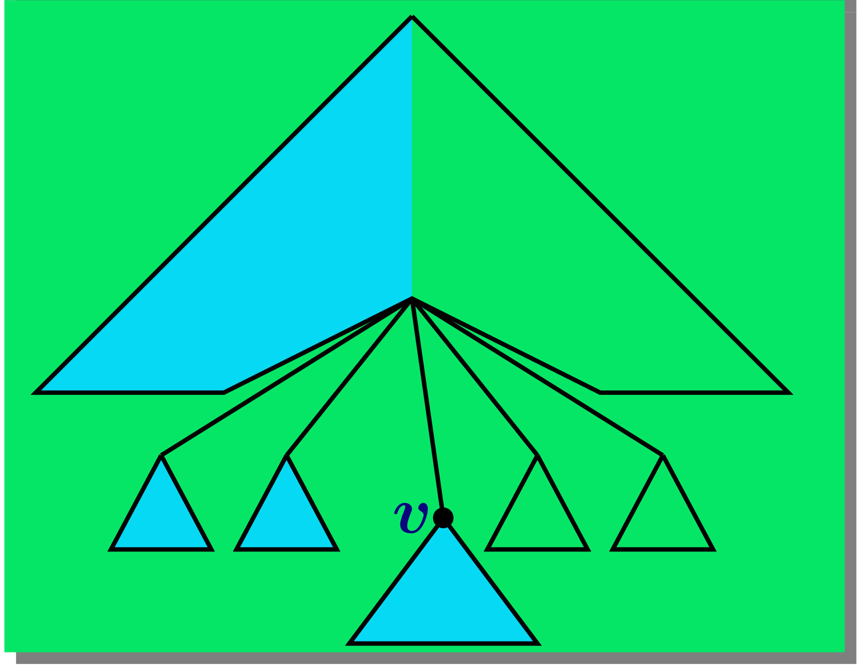
Typing (cont.)

Observations

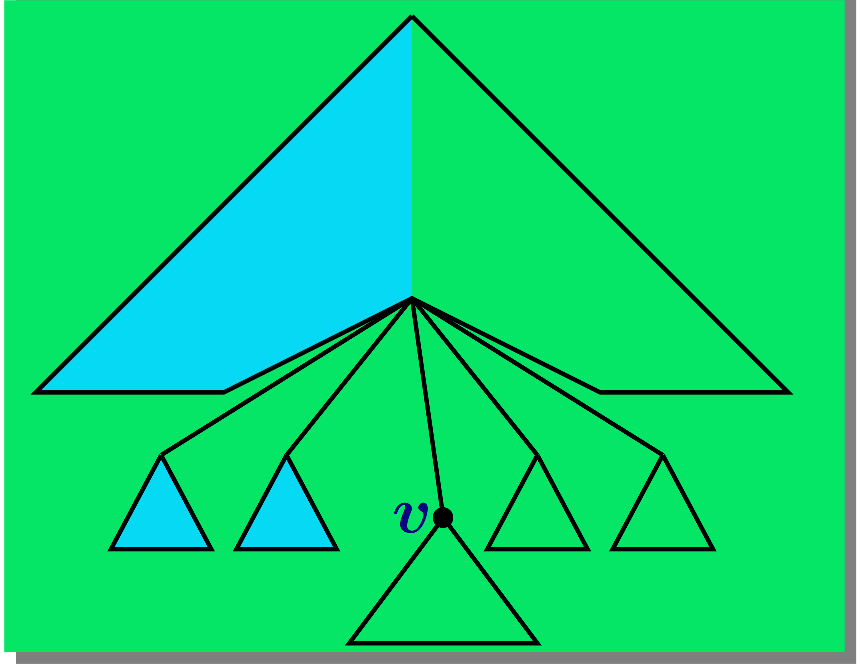
- Type of a node $\equiv$ state of deterministic bottom-up automaton
- Deterministic push-down automaton can assign types during 1 pass
- But the type of a node v is determined after visiting its subtree
- 1-pass preorder typing:

  determine type of v before visiting the subtree of v

...after visiting subtree



...before visiting subtree

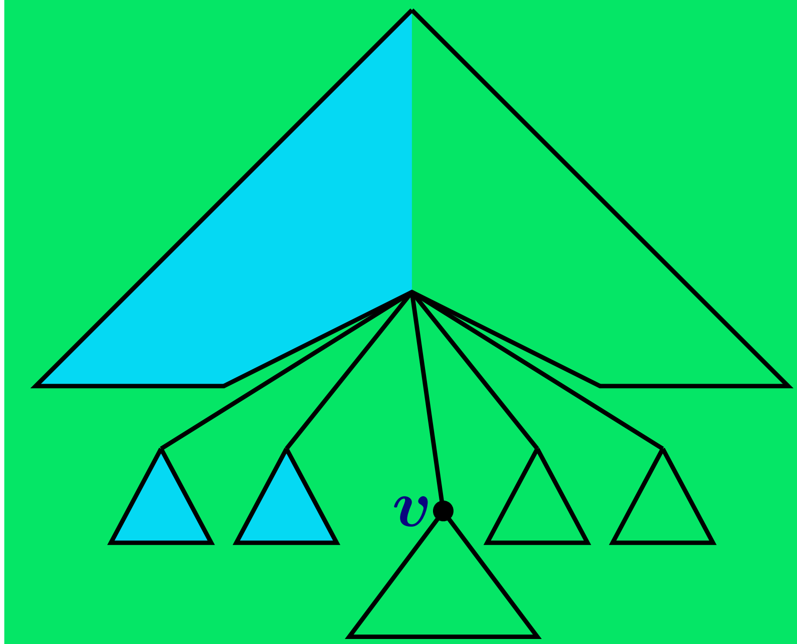


1-Pass Preorder Typing

Question

When would it be important to know the type of v before visiting the subtree of v?

...before visiting subtree



Answer

Whenever the processing proceeds in document order, e.g.:

- Streaming XML: Typing as the first operator in a pipeline
- SAX-based processing

Our next goal

Find out which schemas admit 1-pass preorder typing

1-Pass Preorder Typing (cont.)

Remarks

- The definition of "1-pass preorder typing" does not yet restrict the efficiency of determining the type of a node
- Typing could be 1-pass preorder but very time consuming
- It turns out that essentially this never happens
- Clearly, restrained competition is sufficient for 1-pass preorder typing
- Is it also necessary?

Theorem [Martens, Neven, Sch. 2004]

For a regular tree language L the following are equivalent

(a) L can be described by a 1-pass preorder typable SDTD

(b) L can be described by a restrained-competition SDTD

(c) L has linear time 1-pass pre-order typing

(d) L can be preorder-typed by a deterministic pushdown document automaton

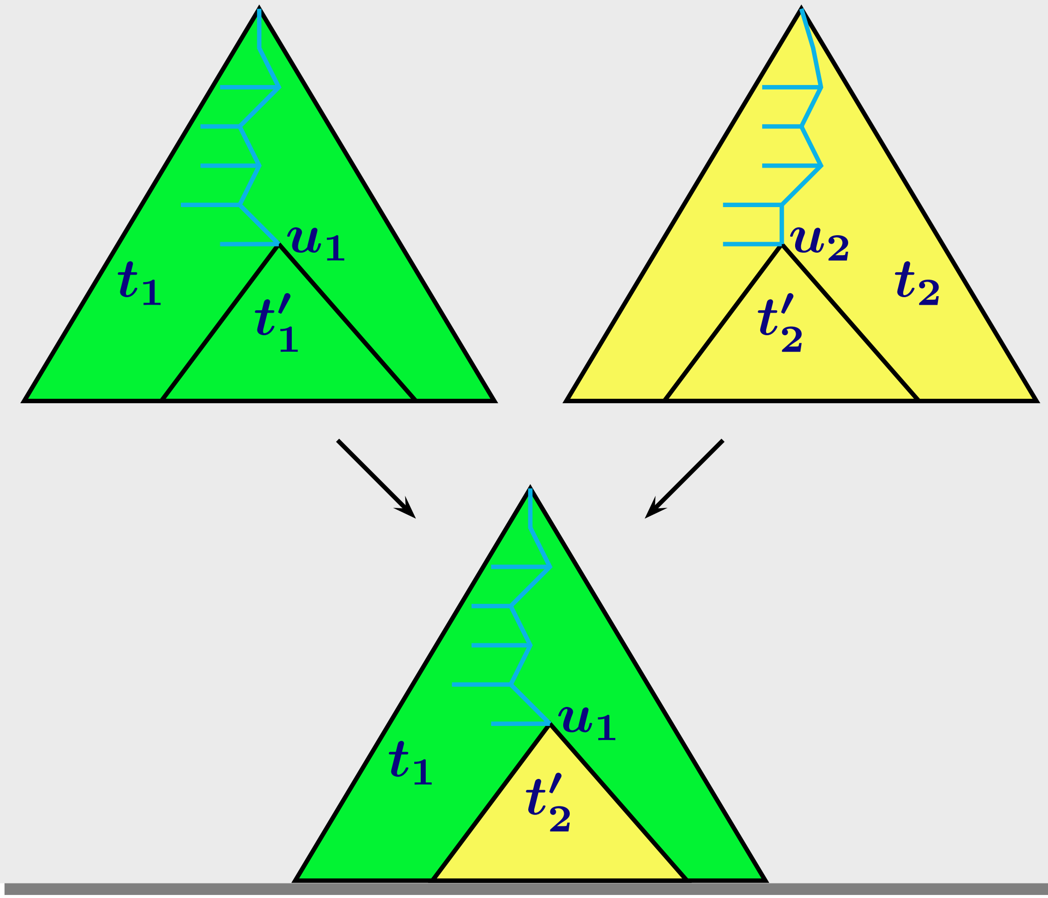
(e) Types for trees in L can be computed by a left-siblings-aware top-down deterministic tree automaton

A Very Robust Class

Further characterizations

- This class has further interesting characterizations
- E.g., by closure under ancestor-sibling-guarded subtree exchange

Illustration



A Related Result

Theorem [Martens, Neven, Sch. 2004]

For a regular tree language L the following are equivalent

(a) L can be described by a single-type SDTD

(b) Types for trees in L can be computed by a simple top-down deterministic tree automaton

(c) L is closed under ancestor-guarded subtree exchange

Summary: Schema Languages

Summary

Expressive power

- Regular tree languages offer a nice framework ($\equiv$ MSO logic!)
- Restrained competition $\equiv$ Deterministic top-down automata

Validation Linear time

Typing

- Linear time
- Efficient 1-pass preorder typing for restrained competition SDTDs

Satisfiability

- DTDs: **PTIME**
- SDTDs: **PTIME**

Containment

- General SDTDs: **EXPTIME**
- Restrained competition SDTDs: **PTIME**

Contents

Node-Selecting Queries

Example document

⟨Composer⟩
⟨Name⟩ Claude Debussy ⟨/Name⟩
⟨Vita⟩
⟨Born⟩ ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
⟨Died⟩ ⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩ ⟨/Died⟩
⟨/Vita⟩
⟨Piece⟩
⟨PTitle⟩ La Mer ⟨/PTitle⟩
⟨PYear⟩ 1905 ⟨/PYear⟩
⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
⟨Movements⟩ 3 ⟨/Movements⟩
...
⟨/Piece⟩
...
⟨/Composer⟩
...

Example query

`//Vita/Died/*`

Observation

`XPath` expressions define sets of nodes → **node-selecting queries**

Node-Selecting Queries (cont.)

Question

Is there a class of node-selecting queries, as robust as the regular tree languages?

Observation

- There is a simple way to define node selecting queries by monadic second-order formulas:
- Simply use one free variable: $\varphi(x)$
- Is there a corresponding automaton model?
- It is relatively easy to add node selection to nondeterministic bottom-up automata

Definition: (Nondetermistic bottom-up node-selecting automata)

- Nondeterministic bottom-up automata plus select function:

$$s : Q \times \Sigma \rightarrow \{0, 1\}$$

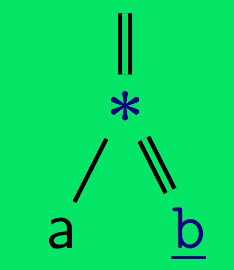
- Node v is in result set for tree t $:\Longleftrightarrow$ there is an accepting computation on t in which v gets a state q such that $s(q, \lambda(v)) = 1$

Example Automaton

Example query

//∗[a]//b

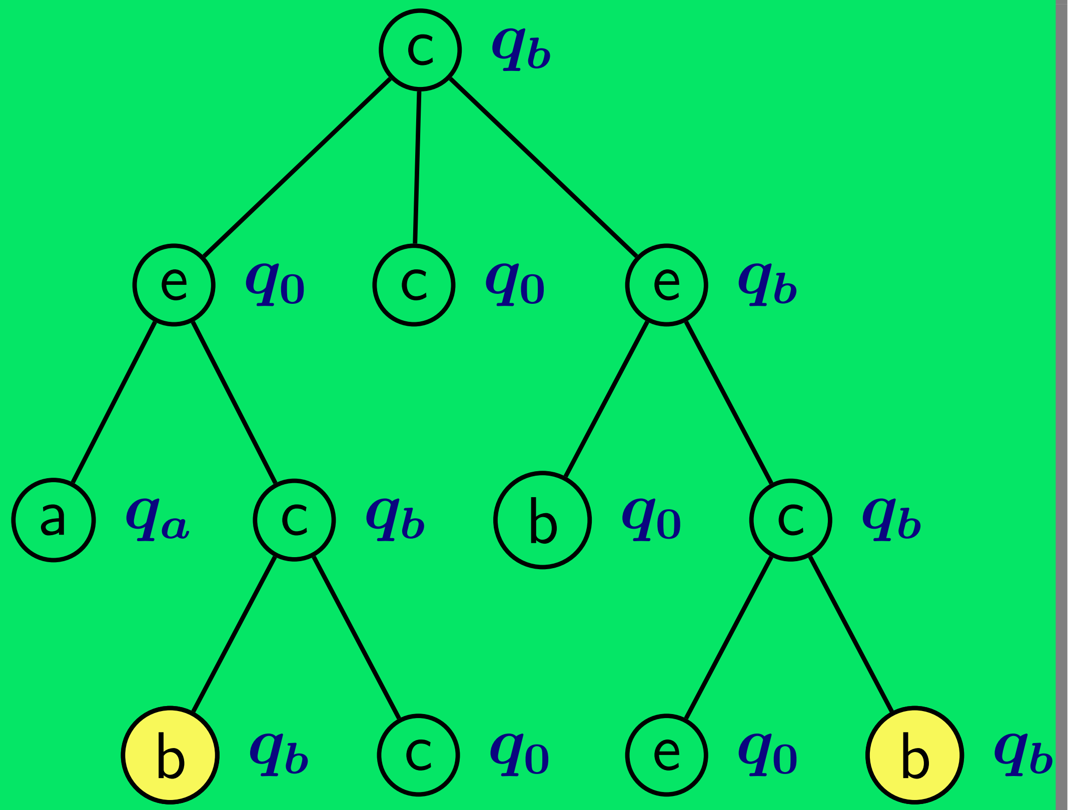
Query tree



Example automaton

- $Q = \{q_0, q_a, q_b\}$
- $L(q_a, a) = Q^*$
- $L(q_b, \sigma) = Q^*$
- $L(q_0, \sigma) =$
  $\epsilon + q_0^* + Q^* q_a Q^*$
- all other sets empty
- $s(q_b, b) = 1$
- all others: 0
- Accepting: q_0

Example tree: Run 2



Node-Selecting Automata

Fact

- Existential semantics: a node is in the result if there exists an accepting run which selects it
- Universal semantics: a node is in the result if every accepting run selects it
- Both semantics define the same class of queries

Result

A node selecting query is MSO-definable iff it is expressible by a nondeterministic bottom-up node selecting automaton

Node-Selecting Automata (cont.)

Result

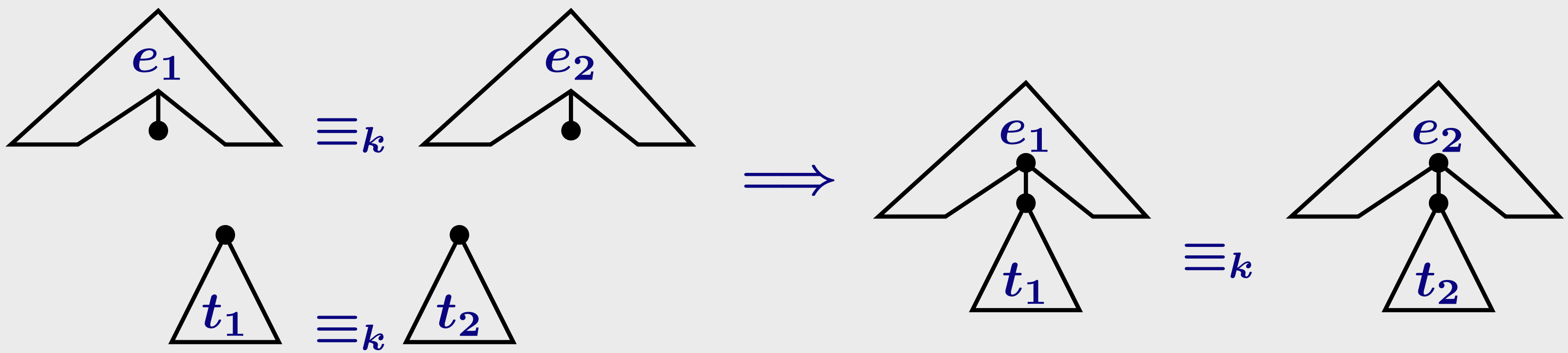
A node selecting query is MSO-definable iff it is expressible by a nondeterministic bottom-up node selecting automaton

Proof idea

- Given formula $\varphi(x)$ of quantifier-depth k and tree t, for each node v the automaton does the following:
  - Compute k-type of subtree at v
  - Guess k-type of "envelope tree" at v
  - Conclude whether v is in the output
  - Check consistency upwards towards the root

$\Rightarrow$ one unique accepting run

Crucial fact



Equivalent Models

More query models

- Unfortunately, the translation from formula to automaton can be prohibitively expensive: number of states $\sim \left.2^{2^{\cdot^{\cdot^{\cdot^{2^{2^{|\varphi|}}}}}}}\right\}|\varphi|$
- Actually: If $\mathbf{P} \neq \mathbf{NP}$ there is no elementary f, such that MSO-formulas can be evaluated in time $f(|\text{formula}| \times p(|\text{tree}|))$ with polynomial p [Frick, Grohe 2002]

$\longrightarrow$ query languages with better complexity properties needed

- Good candidate: Monadic Datalog [Gottlob, Koch 2002] and its restricted dialects like TMNF
- Further models:
  - Attributed Grammars [Neven, Van den Bussche 1998]
  - μ-formulas [Neumann 1998]
  - Context Grammars [Neumann 1999]
  - Deterministic Node-Selecting Automata [Neven, Sch. 1999]

Node-selecting Queries: Evaluation Complexity

Some facts about query evaluation

- MSO node-selecting queries can be evaluated in two passes through the tree
  - first pass, bottom-up: essentially computes the types of the subtrees
  - second pass, top-down: essentially computes the types of the envelopes and combines it with the subtree information
- This can be implemented by a 2-pass pushdown document automaton which in its first pass attaches information to each node [Neumann, Seidl 1998; Koch 2003]
- In particular: queries can be evaluated in linear time

Node-selecting Queries: Static Analysis

Facts

- Satisfiability: Non-emptiness of node-selecting automata is **PTIME**-complete
- Satisfiability of MSO-queries is non-elementary
- Containment of node-selecting automata is **EXPTIME**-complete

Summary: Node-Selecting Queries

Summary

- There is a natural notion of **regular node-selecting queries** generalizing regular tree languages
- Probably for most practical purposes too strong
- But it offers a useful framework for the study of other classes of queries
- A robust but weaker class of queries is captured by pebble automata

Contents

XPath Fragments

XPath and Important Fragments

- Many fragments of `XPath` have been defined
- The main fragments we consider are:
  - **Full `XPath`**: `XPath` 1.0
    (besides the namespace related functions)
  - **Navigational `XPath`** [Gottlob, Koch, Pichler 03, Benedikt, Fan, Kuper 03]:
    Location paths along all axes plus Boolean operations
    (no attributes, no relational operators)
  - **Forward `XPath`**: Navigational `XPath` restricted to
    `child`, `descendant`, `self`, `descendant-or-self`

Main Ingredients of Navigational XPath

- **Location Step**:

$$p = \texttt{Axis :: Node-Test Predicate}^*$$

- **Predicate**: $[\texttt{Expression}]$
- **Location Path**:

$$\pi = \texttt{Location Step / Location Path}$$

  More explicitly: $\pi = p_1 / \cdots / p_k$

- **Expression**: basically a Boolean combination of location steps

Example

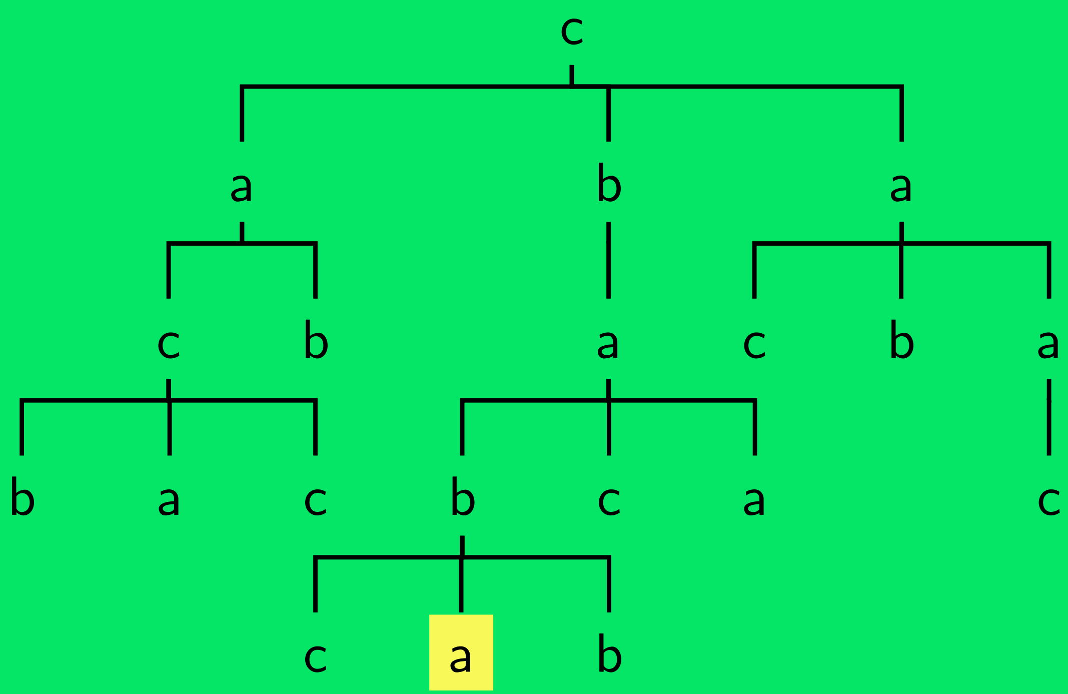
```
/descendant::a/
      child::*[descendant::c and not following-sibling::b]/
            descendant::a
```

XPath Example

Example XPath Expression

`/desc::a/child::*[desc::c and not foll-sib::b]/desc::a`

Example Tree



`XPath` Semantics

`XPath` Semantics

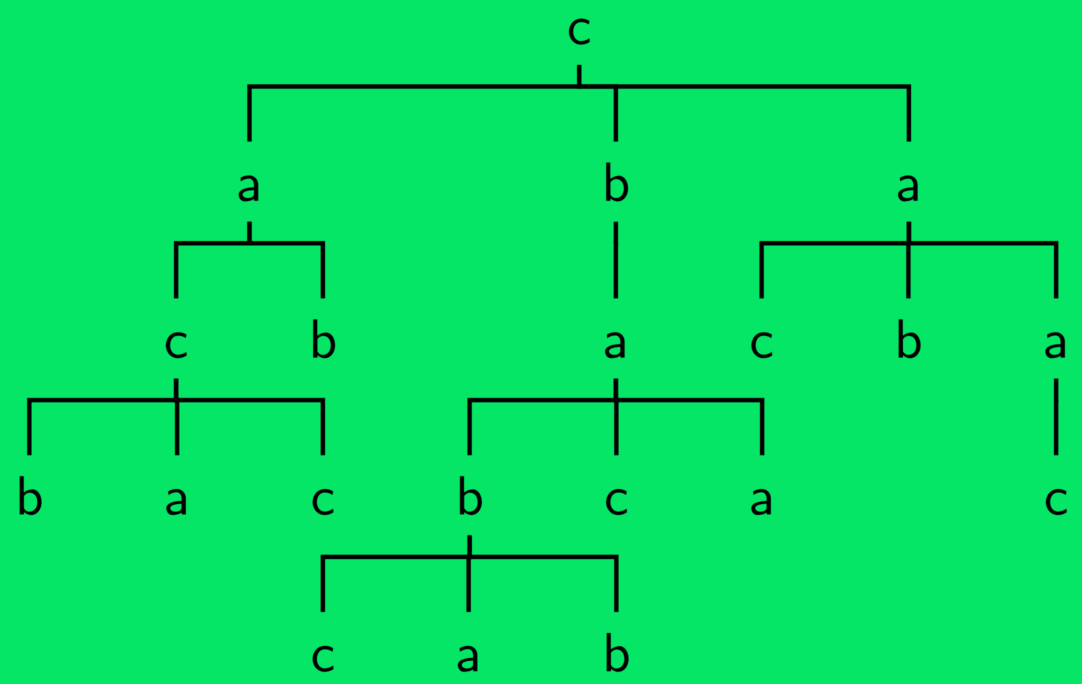
- Result of an expression is a node set or a single value (Boolean, number or string)
- Expressions are evaluated relative to a **context**, in particular relative to a **context node**
- Location step: $\boldsymbol{p} = (\mathtt{a} \ \mathtt{::} \ \mathtt{n} \ \mathtt{q})$ relative to context node $\boldsymbol{u}$ yields the set $[\![\boldsymbol{p}]\!](\boldsymbol{u})$ of nodes $\boldsymbol{v}$ such that
  - $(\boldsymbol{u}, \boldsymbol{v})$ are in a-relation
  - $\boldsymbol{v}$ is labeled according to $\boldsymbol{n}$ (arbitrary, if $\boldsymbol{n} = *$)
  - all predicates of $\boldsymbol{q}$ hold at $\boldsymbol{v}$
- Extended to sets $\boldsymbol{S}$ of nodes: $[\![\boldsymbol{p}]\!](\boldsymbol{S}) = \cup_{\boldsymbol{u} \in \boldsymbol{S}} [\![\boldsymbol{p}]\!](\boldsymbol{u})$
- Location path: $[\![\boldsymbol{p}/\boldsymbol{\pi}]\!](\boldsymbol{S}) = [\![\boldsymbol{\pi}]\!]([\![\boldsymbol{p}]\!](\boldsymbol{S}))$

Example Revisited

Example XPath Expression

`/desc::a/child::*[desc::c and not foll-sib::b]/desc::a`

Example Tree

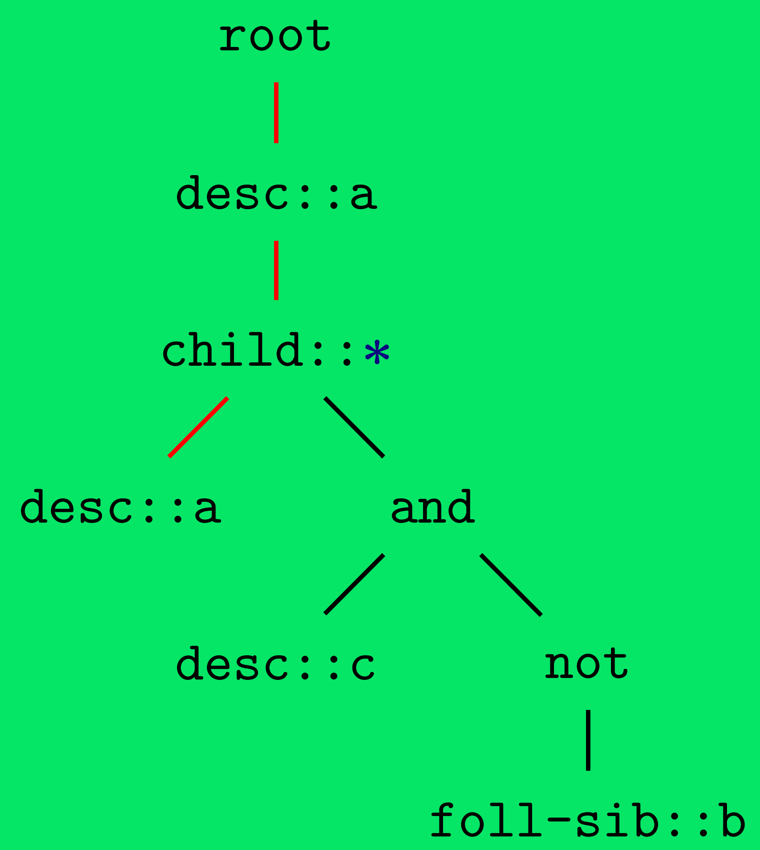


XPath Expression as Query Tree

Example XPath Expression

`/desc::a/child::*[desc::c and not foll-sib::b]/desc::a`

Example Query Tree



A simplified Notation [Benedikt, Fan, Kuper 03]

Notation

- $\downarrow$, $\uparrow$, $\longrightarrow$, $\longleftarrow$, $\circlearrowleft$:
  `child`, `parent`, next-sibling, previous-sibling, `self`
- $\downarrow^+$, $\uparrow^+$, $\longrightarrow^+$, $\longleftarrow^+$:
  `descendant`, `ancestor`, `following-sibling`, `preceding-sibling`
- $\downarrow^*$, $\uparrow^*$, $\longrightarrow^*$, $\longleftarrow^*$:
  `descendant-or-self`, `ancestor-or-self`, following-sibling-or-self, preceding-sibling-or-self

Example

- `child::a/descendant::c/following-sibling::*/parent::b` can be expressed as $\downarrow/\mathsf{a}/\downarrow^+/\mathsf{c}/\longrightarrow/\uparrow/\mathsf{b}$
- The `following`-axis can be expressed via $\uparrow^*/\longrightarrow^+/\downarrow^*$

Contents

Xpath and Existential First-order Logic

Characterizations of `XPath` [Benedikt, Fan, Kuper 03]

- Navigational `XPath` (without `not` and `and`) corresponds to positive existential first-order logic
- Different `XPath` axes correspond to different signatures

Proof idea

- Basic idea:
  For each node v of the query tree: guess a node $h(u)$ in the document tree and check that h is a "homomorphism"
- Main difficulty in proof:
  Deal with conjunctions of conditions

Further Results on

- closure properties
- axiomatizations of equivalence

Backward vs. Forward Axes

Elimination of Backward Axes [Olteanu et al. 02]

- In absolute `XPath` expressions, all backward axes can be eliminated
- Two sets of rewrite rules:
  - with intersection, linear time (and size)
  - without intersection, possibly exponential size

Xpath and First-Order Logic

Reminder

Navigational `XPath` without negation corresponds to positive existential first-order logic

Question: What is needed to capture full first-order logic?

Conditional axes

Conditional axes:
Expressions of the kind P^+, where P is an expression

Example

$(\texttt{child} :: a[\texttt{desc} :: b \text{ or } \texttt{child} :: c])^+$
holds between u and v if

- v is a descendant of u and
- all intermediate nodes
  - are labelled with a and
  - have a c-child or a b-descendant

Xpath and First-Order Logic (cont.)

Theorem [Marx 04]

Navigational `XPath` with conditional axes corresponds exactly to first-order logic (wrt node-selecting queries)

Proof idea

The proof uses a decomposition technique similar to the proof that LTL corresponds to first-order logic over linear structures [Gabbay et al. 80]

Pebble Automata and `XPath`

Definition: Pebble Automata

- Extension of tree-walk automata by fixed number of pebbles
- Only pebble with highest number (current pebble) can move, depending on state, number of pebble symbols under pebbles and incidence of pebbles
- Possible pebble movements:
  - stay, go to left sibling, go to right sibling, go to parent
  - lift current pebble or place new pebble at current position
- Nondeterminism possible

Fact

Deterministic pebble automata capture navigational `XPath` queries

Proof idea

For each node of the query tree:

cycle through all possible nodes of the document tree

Automata and Logic

Some observations

- On strings, MSO logic and (unary) transitive closure logic (TC-logic) coincide
- On trees
  - MSO $\equiv$ parallel automata
  - TC-logic $\equiv$ pebble automata (i.e., strongest sequential automata)
- Whether on trees MSO $\equiv$ TC-logic is open
- The relationship between logics and automata models between FO and TC-logic is largely unexplored:
  - Tree-walk automata,
  - FO-logic $+$ regular expressions
  - Conditional `XPath` $+$ arbitrary star operator
  - ...

Contents

`XPath` Query Evaluation

Naive Evaluation

Procedure Eval($p_1/\cdots/p_n$,v)

$S := [\![p_1]\!]v$

IF $n = 1$ RETURN S ELSE $S' := \emptyset$

FOR $u \in S$ DO $S' := S' \cup$ Eval($p_2/\cdots/p_n$,u)

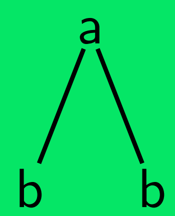
RETURN S'

Complexity

- $T(p_1/\cdots/p_n,t) = O(\text{size of } t) \times T(p_2/\cdots/p_n,t)$
- Could be exponential
- Experiments (reported in [Gottlob, Koch, Pichler 02]) show that available `XPath` processors had exponential complexity

Example

`/descendant::a(/child::b/parent::a)`^{n} on document

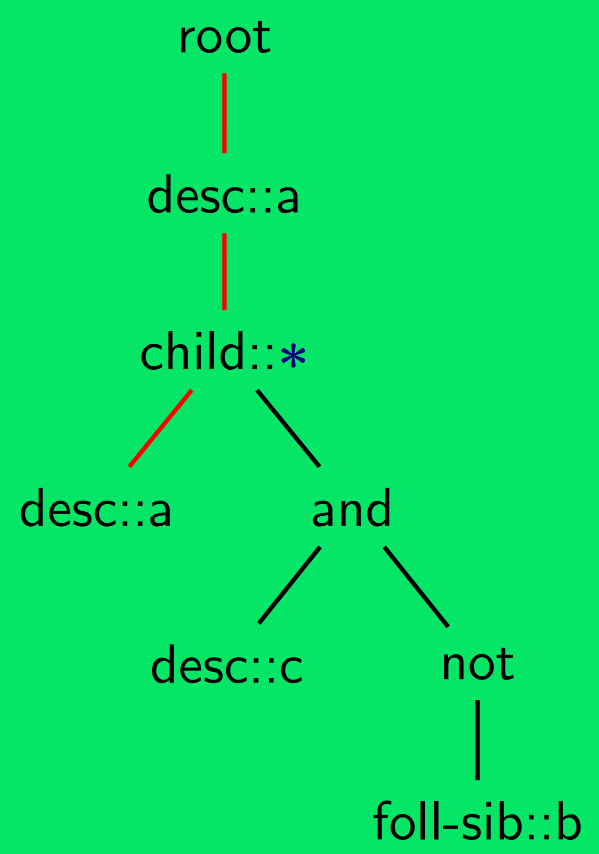


Evaluation of Navigational `XPath`

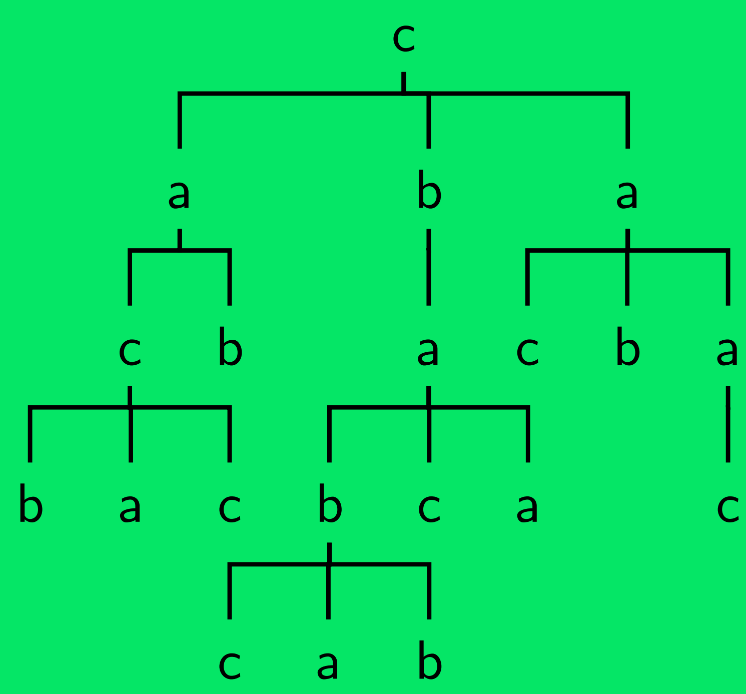
Basic Idea

Combine top-down evaluation of the "main path" with bottom-up evaluation of predicates [Gottlob, Koch, Pichler 02]

Example Query Tree



Example Document



Evaluation of Navigational XPath (cont.)

Evaluation Algorithm for Navigational XPath

Procedure NEval($p_1 / \cdots / p_n$,v)

$S' := \{v\}$

FOR $i := 1$ TO n

(* $p_i = a_i$::n_i q_i *)

$S' := \{u \mid v \in S', (v, u)$ in a_i-relation, u matches $n_i\}$

Compute $S'' := \{u \mid [\![q_i]\!](u) \neq \emptyset\}$ bottom-up

$S' := S' \cap S''$

RETURN S'

Complexity

- For each node of the query tree: $O(|t|)$ steps
- Overall: $O(\text{query size} \times |t|)$

Beyond Navigational XPath

Example expression

`/desc::a/child::*[desc::c[position()` $>$ `1]]/desc::a`

Observations

- In general, a subexpression does not only depend on a context node but also on
  - context position (`position()`)
  - context size (`last()`)

$\longrightarrow$ predicates can no longer be evaluated in a bottom-up fashion

- Basic idea of [Gottlob, Koch, Pichler 02]: Compute the value of each subexpression for each triple (v, i, l) of
  - a node v
  - a position i
  - a size l

Two Algorithms for XPath Evaluation

Results from [Gottlob, Koch, Pichler 02/03]

- The basic idea can be turned into different algorithms:
  - a bottom-up algorithm:
    * Computing the value for each $e, (v, i, l)$ in a dynamic programming fashion
    * Time bound: $O((\text{tree size})^5 \times (\text{query size})^2)$
  - a (mixed) top-down algorithm:
    * Compute as much information as possible in top-down fashion to evaluate subexpressions only for relevant triples (v, i, l)
    * Time bound: $O((\text{tree size})^4 \times (\text{query size})^2)$
- Further time bound for the "extended Wadler fragment": $O((\text{tree size})^2 \times (\text{query size})^2)$

Structural Complexity of XPath Evaluation

Further Results

- In [Gottlob, Koch, Pichler 03] the complexity of XPath evaluation is considered
- Data Complexity:
  - Navigational XPath: **LOGSPACE**-complete (e.g., via pebble automata)
  - Full XPath: also **LOGSPACE** (?)
- Combined Complexity:
  - Navigational XPath: **PTIME**-complete
  - Positive Navigational XPath: **LOGCFL**-complete
  - An even much larger fragment (pXPath) is in **LOGCFL**

Contents

XPath satisfiability

Observation

Not all `XPath` expressions are satisfiable, e.g.:

`child::a/child::b/following-sibling::c/parent::d`

Question

What is the complexity of checking satisfiability of an `XPath` expression for different fragments?

Theorem [Hidders 03]

- Satisfiability for positive navigational `XPath` expressions is in **NP**
- Even for expressions without Boolean operators it is **NP**-hard
- For relative expressions without Boolean operators it is in **P**

Remark

As navigational `XPath` can express star-free regular expressions along a path: Satisfiability of navigational `XPath` is non-elementary
(Note: this depends on the exact notion of *Navigational XPath*)

`XPath` Satisfiability: Upper Bound

Theorem [Hidders 03]

Satisfiability for positive navigational `XPath` expressions is in **NP**

Proof idea

- If an expression e without $\cup$ is satisfiable it has a model of size $\leq |e|$
- For an arbitrary (negation-free) expression guess a disjunct of the disjunctive normal form

XPath Satisfiability: Lower Bound

Theorem [Hidders 03]

Satisfiability for positive navigational XPath expressions without Boolean operators is **NP**-hard

Proof idea

- Reduction from *Bounded Multiple String Matching (BMS)*:
  - Given: Pattern strings $p_1, \ldots, p_n$ over $\{0, 1, *\}$
  - Question: Is there a string over $\{0, 1\}$ of length $|p_1|$ which matches all patterns?
- Example: $*$0$**$1, 00$*$1, $*$111 has solution 00111
- As XPath expression:
  $/\downarrow/\downarrow/0/\downarrow/\downarrow/\downarrow/1\ [\uparrow^*/1/\uparrow/\uparrow/0/\uparrow/0]\ [\uparrow^*/1/\uparrow/1/\uparrow/1]$

Contents

Forward XPath

Example query

`//Vita/Died/*`

Example document

⟨Composer⟩
⟨Name⟩ Claude Debussy ⟨/Name⟩
⟨Vita⟩
⟨Born⟩ ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
⟨Died⟩ ⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩ ⟨/Died⟩
⟨/Vita⟩
⟨Piece⟩
⟨PTitle⟩ La Mer ⟨/PTitle⟩
⟨PYear⟩ 1905 ⟨/PYear⟩
⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
⟨Movements⟩ 3 ⟨/Movements⟩
...
⟨/Piece⟩
...
⟨/Composer⟩
...

Abbreviated Syntax for Forward XPath

Another example query

`(/*[Name]//When) | (//Where)`

Example do…

⟨Composer⟩
  ⟨Name⟩ Claude Debussy ⟨/Name⟩
  ⟨Vita⟩
    ⟨Born⟩ ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩ ⟨/Born⟩
    ⟨Married⟩ ⟨When⟩ October 1899 ⟨/When⟩ ⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
    ⟨Married⟩ ⟨When⟩ January 1908 ⟨/When⟩ ⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
    ⟨Died⟩ ⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris…
  ⟨/Vita⟩
  ⟨Piece⟩
    ⟨PTitle⟩ La Mer ⟨/PTitle⟩
    ⟨PYear⟩ 1905 ⟨/PYear⟩
    ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
    ⟨Movements⟩ 3 ⟨/Movements⟩
    ...
  ⟨/Piece⟩
  ...
⟨/Composer⟩
...

More XPath operators

Operator	Meaning
p/q	child
$p//q$	descendant
$p[q]$	filter
$*$	wildcard
$p \mid q$	disjunction

XPath containment

Question

Does `//Vita/Died/*` always select a subset of positions of `(/*[Name]//When) | (//Where)`?

Answer

No!

Counter-example

⟨Vita⟩
  ⟨Died⟩
    ⟨How⟩ Heart disease ⟨/How⟩
  ⟨/Died⟩
⟨/Vita⟩

Further question

But what if the type of documents is constrained?

XPath Containment (cont.)

Fact

For all XML documents of type

```
<!DOCTYPE Composers [
    <!ELEMENT Composers (Composer*)>
    <!ELEMENT Composer (Name, Vita, Piece*)>
    <!ELEMENT Vita (Born, Married*, Died?)>
    <!ELEMENT Born (When, Where)>
    <!ELEMENT Married (When, Whom)>
    <!ELEMENT Died (When, Where)>
    <!ELEMENT Piece (PTitle, PYear,
         Instruments, Movements)>
    ]>
```

the pattern `//Vita/Died/*` always selects a subset of positions of

`(/*[Name]//When) | (//Where)`

XPath Containment: Definition

Definition: Containment for XPath(S)

Let S be a set of XPath-operators. The containment problem for XPath(S) is:

Given: XPath(S)-expression p, q

Question: Is $p(t) \subseteq q(t)$ for all documents t?

Definition: Containment for XPath (S) with DTD

Let S be a set of XPath-operators. The containment problem for XPath(S) in the presence of DTDs is:

Given: XPath(S)-expression p, q, DTD d

Question: Is $p(t) \subseteq q(t)$ for all documents t satisfying $t \models d$?

Observation

These problems are crucial for static analysis and query optimization

Question

For which fragments S are these problems

- decidable?
- efficiently solvable?

Results

General remarks

- The XPath containment problem has been considered for various sets of operators
- Focus on Forward XPath
- Results vary from **PTIME** to "undecidable"
- Various methods have been used:
  - Canonical model technique
  - Homomorphism technique
  - Chase technique
- More about this in [Miklau, Suciu 2002; Deutsch, Tanen 2001; Sch. 2004]
- We will consider automata based techniques

The Automata Technique

Definition: (Relative Containment for XPath (S) wrt DTD)

Let S be a set of XPath-operators. The containment problem for XPath(S) relative to a DTD is:

Given: XPath(S)-expression p, q, DTD d

Question: Is $p(D) \subseteq q(D)$ for all documents D satisfying $D \models d$?

A vague plan

- Construct an automaton $\mathcal{A}_p$ for p
- Construct an automaton $\mathcal{A}_q$ for q
- Construct an automaton $\mathcal{A}_d$ for d
- Combine these automata suitably to get an automaton which accepts all counter-example documents

A Simplification

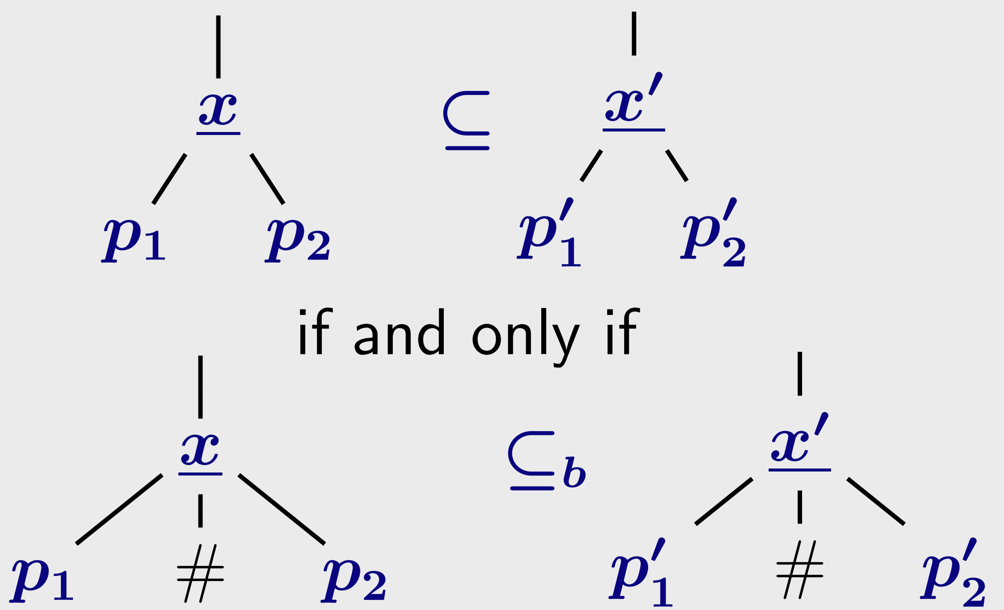
Definition: (Boolean containment)

$p \subseteq_b q$ $:\Longleftrightarrow$ whenever p selects *some* node in a tree t then q also selects some node in t.

Useful observation [Miklau, Suciu 2002]

In the presence of [ ], Boolean containment has the same complexity as containment.

Crucial idea



XPath Containment: 2 Examples

Result 1 [Neven, Sch. 2003]

The Boolean containment problem for XPath$(/, //)$ in the presence of DTDs is in **PTIME**

Result 2 [Neven, Sch. 2003]

The Boolean containment problem for XPath$(/, //, [\,], *, |)$ in the presence of DTDs is in **EXPTIME**

Note

Both results are optimal wrt complexity: the problems are complete for these classes

Containment for XPath(/, //) and DTDs

Result 1 [Neven, Sch. 2003]

The Boolean containment problem for XPath(/, //) in the presence of DTDs is in **PTIME**

Proof idea

- XPath(/, //)-expressions can only describe vertical paths in a tree
- Each expression is basically of the form $p_1//p_2//\cdots//p_k$, where each p_i is of the form $l_{i1}/\cdots/l_{im_i}$
- On strings this is a sequence of string matchings corresponding to a regular language L

$\Rightarrow$ Deterministic string automaton of linear size

- Recall: there is a deterministic top-down automaton which checks whether a p-path exists

$\Rightarrow$ Deterministic top-down automaton $\mathcal{A}_p$

$\Rightarrow$ Deterministic top-down automaton $\mathcal{A}_{\overline{q}}$ checking that no q-path exists

Containment for XPath(/, //) and DTDs

Result 1 [Neven, Sch. 2003]

The containment problem for XPath(/, //) in the presence of DTDs is in **PTIME**

Proof idea (cont.)

- Deterministic top-down automaton $\mathcal{A}_p$
- Deterministic top-down automaton $\mathcal{A}_{\overline{q}}$ checking that no q-path exists
- There is a deterministic top-down automaton $\mathcal{A}_d$ checking whether t conforms to d
- $p \subseteq_b q$ in the presence of $d \iff L(\mathcal{A}_p \times \mathcal{A}_{\overline{q}} \times \mathcal{A}_d) = \emptyset$
- The latter can be checked in polynomial time

Containment for XPath(/, //, [ ], ∗, |) and DTDs

Result 2 [Neven, Sch. 2003]

The containment problem for XPath(/, //, [ ], ∗, |) in the presence of DTDs is in **EXPTIME**

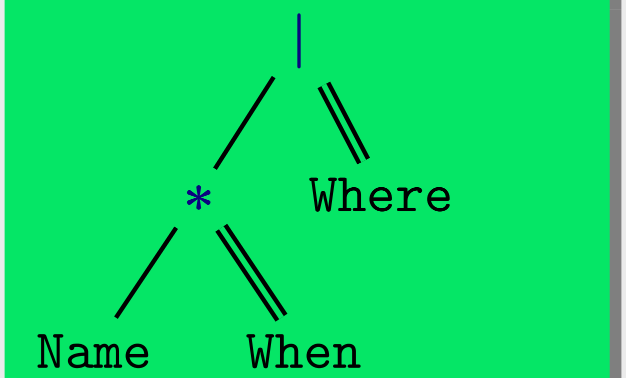
Proof idea

We again represent patterns like

(/*[Name]//When) | (//Where)

as query trees:

Example query tree



Lemma

For each XPath(/, //, [ ], ∗, |)-expression p there is a deterministic bottom-up automaton $\mathcal{A}_p$ of exponential size which checks whether in a tree p holds

Containment for $\texttt{XPath}(/, //, [\,], *, |)$ and DTDs

Lemma

For each $\texttt{XPath}(/, //, [\,], *, |)$-expression p there is a deterministic bottom-up automaton $\mathcal{A}_p$ of exponential size which checks whether in a tree p holds

Proof idea for Lemma

- States of $\mathcal{A}_p$ are of the form $(S_/, S_{//})$
- Both $S_/$ and $S_{//}$ are sets of positions of the query tree:
  - $S_/$: positions matching v
  - $S_{//}$: positions matching some node in the subtree of v

Containment for $\texttt{XPath}(/, //, [\,], *, |)$ and DTDs

Result 2 [Neven, Sch. 2003]

The containment problem for $\texttt{XPath}(/, //, [\,], *, |)$ in the presence of DTDs is in **EXPTIME**

Proof idea (cont.)

- Construct deterministic bottom-up automaton $\mathcal{A}_p$ of exponential size
- Construct deterministic bottom-up automaton $\mathcal{A}_{\overline{q}}$ of exponential size
- Construct deterministic bottom-up automaton $\mathcal{A}_d$ of exponential size
- $p \subseteq_b q$ in the presence of $d \iff L(\mathcal{A}_p \times \mathcal{A}_{\overline{q}} \times \mathcal{A}_d) = \emptyset$

$\Rightarrow$ exponential time

Corresponding Lower Bound

Theorem

The containment problem for $\texttt{XPath}(/, //, [\,], *, |)$ in the presence of DTDs is **EXPTIME**-hard

Proof sketch

Proof by reduction from *Two-player corridor tiling*

Example

Example:

Top row $T = \boxed{c}\boxed{a}\boxed{a}\boxed{c}$

Bottom row $B = \boxed{a}\boxed{c}\boxed{a}\boxed{c}$

Vertical and horizontal constraints:

$$V = \begin{array}{c}\boxed{c}\\\boxed{c}\end{array}, \begin{array}{c}\boxed{a}\\\boxed{c}\end{array}, \begin{array}{c}\boxed{c}\\\boxed{a}\end{array}$$

$$H = \boxed{a}\boxed{c}, \boxed{a}\boxed{a}, \boxed{c}\boxed{a}$$

Player I to move

Player II lost

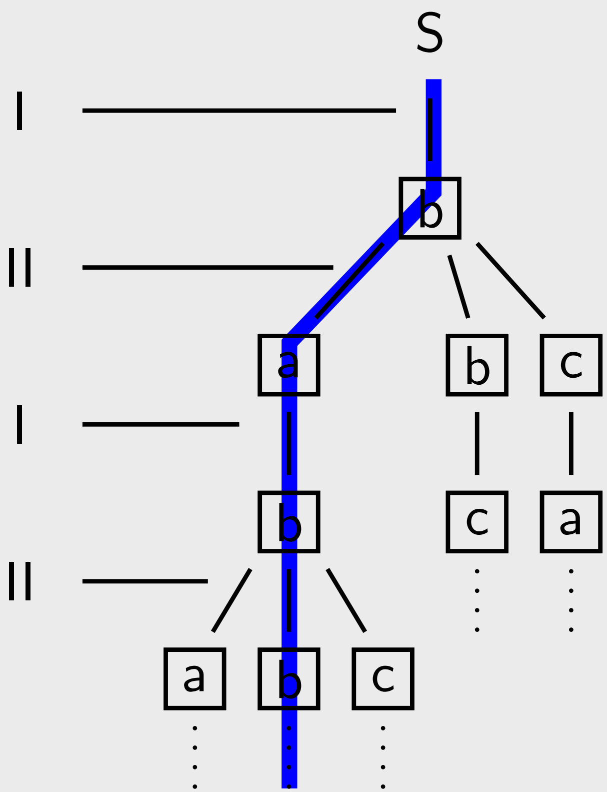
c	a	a	c
⋮	⋮	⋮	⋮
c	c		
c	a	c	a
a	c	a	c

Deciding whether player I has a winning strategy is **EXPTIME**-complete

Strategies as trees

Proof Sketch (cont.)

Tiles over $\{a, b, c\}$



This DTD describes all strategy trees:

$$S \rightarrow (a, \mathsf{I}) + (b, \mathsf{I}) + (c, \mathsf{I})$$

$$(\sigma, \mathsf{I}) \rightarrow (a, \mathsf{II})(b, \mathsf{II})(c, \mathsf{II}) + \# + \$^{\mathsf{II}}$$

$$(\sigma, \mathsf{II}) \rightarrow (a, \mathsf{I}) + (b, \mathsf{I}) + (c, \mathsf{I}) + \# + \$^{\mathsf{I}} + !$$

$$\$^{\mathsf{II}} \rightarrow (a, \mathsf{II})(b, \mathsf{II})(c, \mathsf{II})$$

$$\$^{\mathsf{I}} \rightarrow (a, \mathsf{I}) + (b, \mathsf{I}) + (c, \mathsf{I})$$

$\$ =$ line separator $\quad \# =$ terminal symbol

! indicates misplaced tile

One path corresponds to one game

Winning strategies and paths

Proof Sketch (cont.)

There are various kinds of paths in a game tree:

(a) Legal tilings $\Longrightarrow$ Player I wins

(b) Syntactically wrong: some row of wrong length

(c) II places a wrong tile $\Longrightarrow$ Player I wins

(d) I places a wrong tile $\Longrightarrow$ Player II wins

Player I has a winning strategy

$\Longleftrightarrow$

there is a tree in which all paths are of the form (a) or (c)

We want to construct q such that all paths of the form (b) or (d) are selected

Then: Player I wins iff $/S \not\subseteq q$ wrt DTD

Problem: if II places a wrong tile, I might be forced to place a wrong tile, too

$\Longrightarrow$ We let player I mark wrong tiles of II by $\boxed{!}$

$\Longrightarrow$ We have to check that I does this correctly

Path conditions

Proof Sketch (cont.)

Player I has winning strategy $\iff$ $/S \not\subseteq q$

q expresses that one of the following holds

- Player I violates a horizontal constraint:

  For each $(x,y) \notin H$: $//(x,II)/(y,I)$

- Player I violates a vertical constraint:

  For each $(x,y) \notin V$: $//(x,I)/*^{n+1}/(y,I)$

- Some row does not contain exactly n tiles

  $$\mathcal{D}^{n+1} \mid \bigcup_{i=0}^{n-1} (\$^{\mathsf{I}}|\$^{\mathsf{II}}|S)/\mathcal{D}^i/(\$^{\mathsf{I}}|\$^{\mathsf{II}}|\#)$$

- Player I wrongly claims a mistake of II:

  For each $(x,y) \in V, (x',y) \in H$:
  $//(x,II)/*^n/(x',I)/(y,II)/\boxed{!}$

- Some more conditions on B and T

$* =$ OR of all symbols, $\sigma^i = \sigma/\cdots/\sigma$ (i times)
$\mathcal{D} = (d_1,I)|\cdots|(d_m,I)|(d_1,II)|\cdots|(d_m,II)$

Related work on XPath containment

More Results

- Containment of XPath with $/$ and a subset of $\{//, [\,], *\}$ was studied in [Miklau and Suciu 2002]:
  - Containment of $\text{XPath}(//, [\,], *)$ is **coNP**-complete even if the number of $*$ or the number of $[\,]$ is bounded
  - If the number of $//$ is bounded then it is in polynomial time
- XPath containment in the presence of DTDs and simple integrity constraints was investigated in [Deutsch and Tanen 2001]:
  - In general (*unbounded* constraints): undecidable
- More complexity results between **coNP** and undecidable for other fragments and extensions in [Neven and S. 2003]

Some Open Questions

- What's the exact borderline between fragments of XPath with decidable and undecidable containment problem?
- To what extent can the presented result be extended to other axes (siblings, backward)?

Summary: XPath

Summary

Expressive Power

Closely related to first-order logic

Evaluation

- In general: Polynomial time
- Large fragments in linear time
- Structural complexity between **LOGSPACE** and **PTIME**

Satisfiability

- Without negation: **PTIME** or **NP**
- With negation: non-elementary

Containment

- Varying from **PTIME** to undecidable
- Upper bound for positive navigational XPath?

Contents

XSLT Typechecking

Definition: Transformation typechecking

Given: DTDs d_1 and d_2 and a transformation T

Result: Is $T(t)$ valid wrt. d_2, for each document t valid wrt. d_1?

Question: Is XSLT typechecking decidable?

Question: What is the complexity?

Outline of the Following

- Provide an automata model for XSLT transformations
- Show that the behaviour of these automata can be captured by MSO logic
- Use manipulation of regular tree languages to solve type checking problem

$\longrightarrow$ This part is based on [Milo,Suciu,Vianu 01]

XSLT in more detail

How XSLT Roughly Works

Templates:

$\langle$`xsl:template name=TName match=pattern mode=MName`$\rangle$

Template application:

$\langle$`xsl:apply-templates select=Expression mode=MName`$\rangle$

XSLT Processing Whenever `xsl:apply-templates` is called at a node v the following happens:

- Compute set $S(v)$ of nodes, reachable from v via `Expression` (if `select` is not present, $S(v) =$ children of v)
- For each $w \in S(v)$ compute which templates that can be applied to w:
  - w has to match `pattern` of a template
  - the `mode` of the template has to be the same as the mode of `xsl:apply-templates`
- If no template matches, take the default template
- For each $w \in S(v)$ select the best template and apply it.

The process starts at the root of the tree

XSLT: Example

Example Transformation

Remove everything below a $\boldsymbol{c}$*. Translate* $\boldsymbol{a}$ *below* $\boldsymbol{b}$ *into* $\boldsymbol{d}$

Example `XSLT` (Abbreviated)

⟨... match="a"⟩ ⟨a⟩ ⟨xsl:apply-templates⟩ ⟨/a⟩ ⟨/...⟩

⟨... match="a" mode="below"⟩ ⟨d⟩ ⟨xsl:apply-templates⟩ ⟨/d⟩ ⟨/...⟩

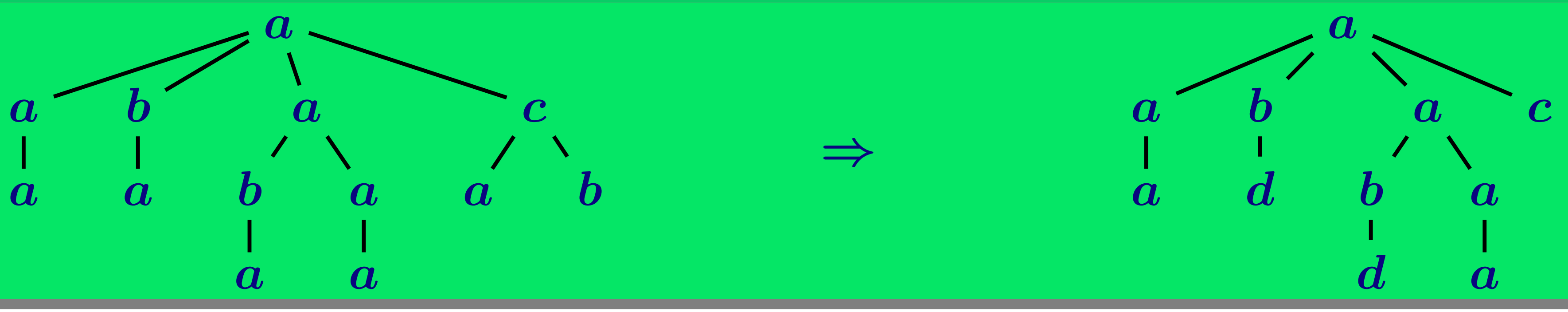
⟨... match="b"⟩ ⟨b⟩ ⟨xsl:apply-templates mode="below"⟩ ⟨/b⟩ ⟨/...⟩

⟨... match="b" mode="below"⟩ ⟨b⟩ ⟨xsl:apply-templates mode="below"⟩ ⟨/b⟩ ⟨/...⟩

⟨... match="c"⟩ ⟨c⟩ ⟨/c⟩ ⟨/...⟩

⟨... match="c" mode="below"⟩ ⟨c⟩ ⟨/c⟩ ⟨/...⟩

Example Trees



XSLT: More involved example

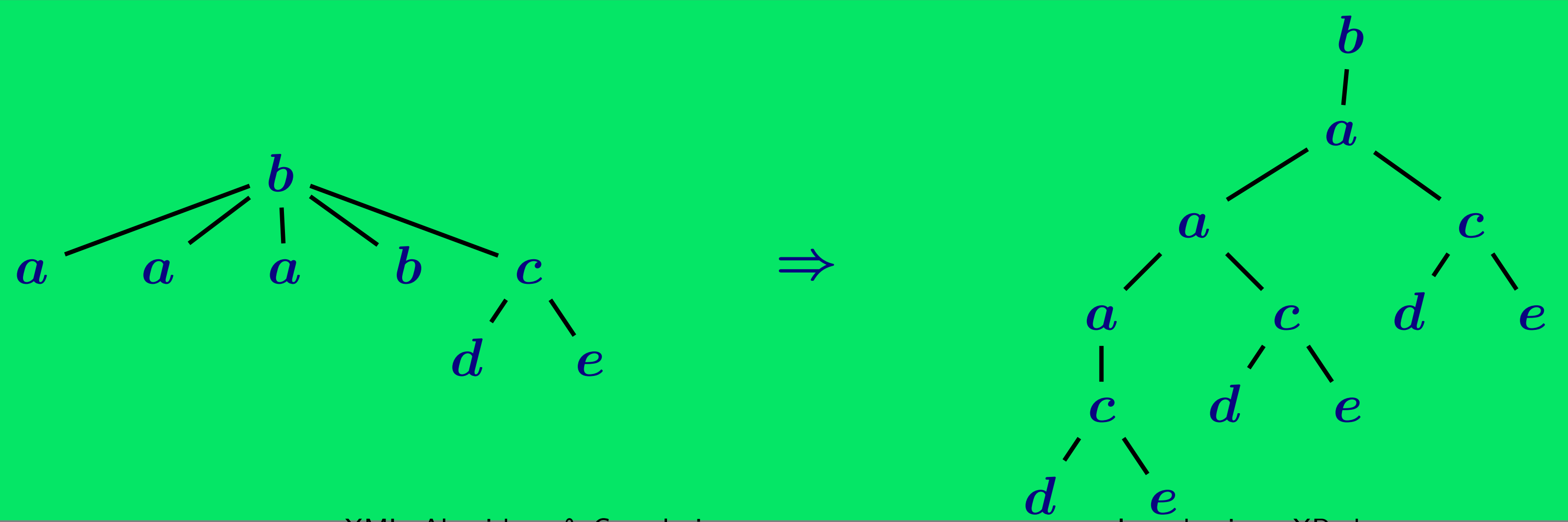
Remark

The previous example corresponds to top-down tree transducers

Example XSLT

```
⟨xsl:template match="/b"⟩
   ⟨b⟩ ⟨xsl:apply-templates select='child[1]' mode="acopy"⟩ ⟨/b⟩
⟨/xsl:template⟩
⟨xsl:template match="a" mode="acopy"⟩
   ⟨a⟩
   ⟨xsl:apply-templates select='child[1]' mode="acopy"⟩
   ⟨xsl:copy-of select='/child[last()]'⟩
   ⟨/a⟩
⟨/xsl:template⟩
```

Example Trees



An automaton model for XSLT

Definition: k-pebble Transducer

- Work on binary tree encodings of unranked trees
- Up to k pebbles can be placed on the tree
- Only pebble with highest number (*current pebble*) can move, depending on state, number of pebbles symbols under pebbles and incidence of pebbles
- possible pebble movements:
  - stay
  - go to left child, right child or parent
  - lift current pebble
  - place new pebble on the root
- Nondeterminism allowed
- If current pebble stays it is possible to produce output:
  - a node with two (forthcoming) subtrees; in this case two independent subcomputations (*branches*) are started, which construct the left subtree and right subtree, respectively
  - a leaf; in this case the computation branch stops

Computing XSLT transformations by k-pebble transducers

Fact

k-pebble transducers can evaluate most XPath expressions (and produce as output an encoded version of the result list) - even with other axes than the forward axis

Proof idea

- Whenever `xsl:apply-templates` is called at a node v the following happens:
  - Cycle through the set $S(v)$ of nodes, reachable from v via `Expression` (if `select` is not present, $S(v) =$ children of v)
  - For each $w \in S(v)$ check which templates can be applied to w:
    * w has to match `pattern` of a template
    * the `mode` of `xsl:apply-templates` is stored in the state of the automaton
  - For each $w \in S(v)$ select the best template and branch into
    * a subcomputation which handles the next node in $S(v)$ (via the right child)
    * a subcomputation which applies the template to the current node
- The computation starts at the root of the tree

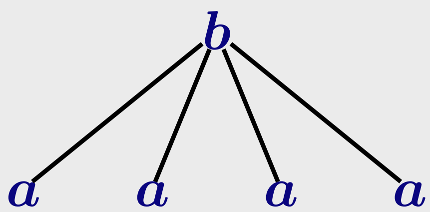
Back to the Typechecking Question

Question: Is XSLT typechecking decidable?

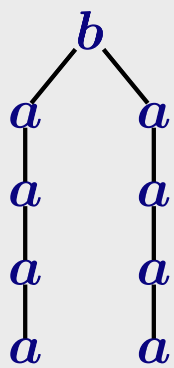
Proof idea

- How can we check that $T(t) \in L(d_2)$, for each $t \in L(d_1)$?
- Obvious approach:
  - Compute $T(L(d_1))$
  - Check that $T(L(d_1)) \subseteq L(d_2)$
- Problem: $T(L(d_1))$ does not need to be regular:

  Transform



  into



- Alternative approach:
  - Compute $T^{-1}(\overline{L(d_2)})$
  - Check $L(d_1) \cap T^{-1}(\overline{L(d_2)}) = \emptyset$

Pebble acceptors

Definition: k-pebble acceptors

- Basically the same as k-pebble transducers
- Instead of output producing steps:
  - *accept*
  - branch into two independent subcomputations
- A tree is accepted if all subcomputations accept

Main Steps of the Proof

(i) $T^{-1}(\overline{L(d_2)})$ is accepted by a k-pebble acceptor

(ii) k-pebble acceptors only accept regular tree languages

Step (i)

Lemma

$T^{-1}(\overline{L(d_2)})$ is accepted by a k-pebble acceptor

Proof

- Let B be a nondeterministic top-down tree automaton which accepts $\overline{L(d_2)}$
- Let T be a k-pebble tree transducer
- We construct k-pebble acceptor A for $T^{-1}(\overline{L(d_2)})$, i.e., an automaton which on input t decides whether there is a tree in $T(t)$ which is accepted by B:
  - Simulate T on t and B
  - Simulate at the same time the behaviour of B on the (virtual) output tree - this is possible as the output tree is produced top-down and can be instantly consumed by B
  - The simulation involves branching, whenever T branches, and produces two new subtrees

Step (ii)

Lemma

k-pebble acceptors only accept regular tree languages

Proof idea

Show that the language of a k-pebble acceptor can be expressed by an MSO-formula:

1. Reduce k-pebble automaton acceptance to AGAP (Alternating Graph Accessibility)
2. Show that AGAP can be expressed in MSO
3. Some adjustments necessary

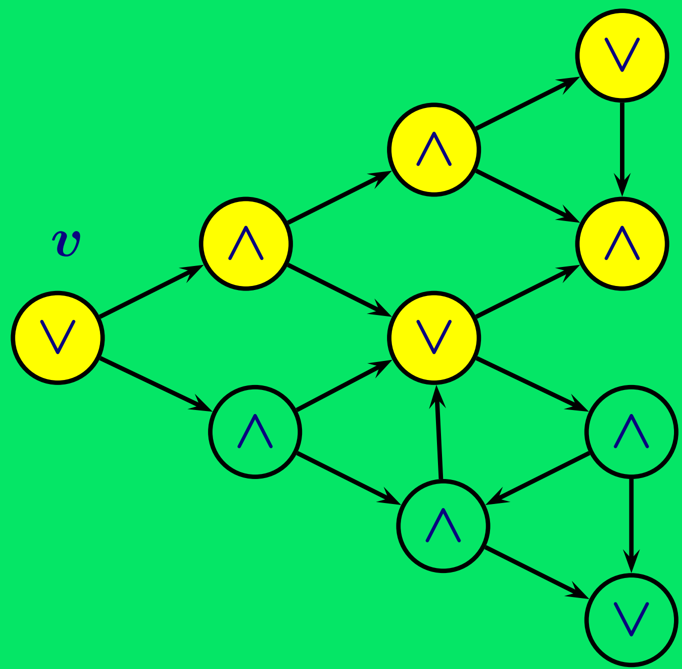
Alternating Graph Accessibility

Definition: Accessible Nodes

Let $G = (V, E)$, $V = V_\wedge \cup V_\vee$. A node w is **accesible** if

- $w \in V_\wedge$ and all successors of w are accessible, or
- $w \in V_\vee$ and at least one successor of w is accessible

Example



Definition: Alternating Graph Accessibility Problem (AGAP)

Given: Graph $G = (V, E)$, $V = V_\wedge \cup V_\vee$, and $v \in V$

Question: Is v accessible?

Alternating Graph Accessibility (cont.)

Construction of $G_{A,t}$ from Automaton A and Tree t

- Nodes in $V_\vee$ are the configurations of A on t:
  tuples $[i, q, \theta]$, where $\theta : \{1, \ldots, i\} \longrightarrow t$
- Nodes in $V_\wedge$ are ϵ and pairs (γ_1, γ_2) of configurations with "the same θ"
- Edges:
  - $(\gamma_1, \gamma_2) \longrightarrow \gamma_1$, $(\gamma_1, \gamma_2) \longrightarrow \gamma_2$
  - $\gamma \longrightarrow \gamma'$, if this is a step of A
  - $\gamma \longrightarrow \epsilon$, if A can get into the accept state from γ
  - $\gamma \longrightarrow (\gamma_1, \gamma_2)$ if this is a branching step of A

Fact

A k-pebble acceptor A accepts a tree $t \iff \gamma$ is accessible in $(G_{A,t})$

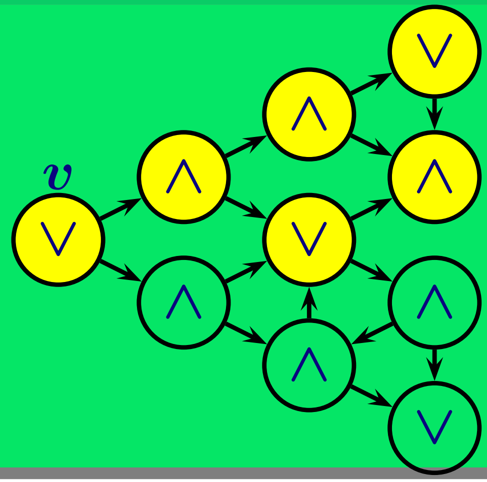
AGAP is MSO-expressible

Definition: Reverse-closed Sets of Nodes

A set S of nodes is **reverse-closed** if the following holds

- if v is in $V_\wedge$ and $w \in S$, for all nodes w with $(v, w) \in E$, then $v \in S$
- if v is in $V_\vee$ and $w \in S$, for some node w with $(v, w) \in E$, then $v \in S$

Example



Fact

Node v is accessible iff it is in every reverse-closed set of nodes.

...as MSO-Formula

v accessible $\equiv \forall S\ (\text{reverse-closed}(S) \longrightarrow S(v))$, where

$$\begin{aligned} \text{reverse-closed}(S) \quad \equiv \quad & \forall x\, ([V_\wedge(x) \wedge \forall y\, (E(x,y) \longrightarrow S(y))] \longrightarrow S(x)) \wedge \\ & ([V_\vee(x) \wedge \exists y\, (E(x,y) \wedge S(y))] \longrightarrow S(x)) \end{aligned}$$

k-pebble acceptors and MSO

Proof idea

Unfortunately, $G_{A,t}$ has too many nodes to use this directly:

- MSO can only quantify over sets of linear size in the given structure (i.e., t)
- $G_{A,t}$ has $\Omega(|t|^k)$ configurations
- But $G_{A,t}$ has a special structure:
  Nodes are only connected if their number of pebbles is the same ± 1 and they agree in all but at most the last pebble

k-pebble acceptors and MSO (cont.)

Proof (cont.)

- Wlog assume that each state of A is only used for a fixed number of pebbles: $Q = Q_1 \cup \cdots \cup Q_k$, where the states in Q_i are only used, when i pebbles are present
- Further assume that all sets Q_i are of equal size m: $Q_i = \{q_{i1}, \ldots, q_{im}\}$
- $k = 1$:
  - Use one relation S_i^1 for each state q_{1i}
  - Intended meaning of $v \in S_i^1$:
    there is an accepting subcomputation of A starting at v in state q_{1i}
  - $\varphi = \forall S_1^1 \cdots \forall S_m^1$ (reverse-closed $\longrightarrow S_1^1$(root))
  - reverse-closed is a conjunction of subformulas, induced by the transitions of A, e.g.:
    * if $(q_{1i}, a) \longrightarrow$ accept then $\forall x\, Q_a(x) \longrightarrow S_i^1(x)$
    * if $(q_{1i}, a) \longrightarrow (q_{1j},$ down-right$)$ then
      $\forall x\, \forall y\, (Q_a(x) \wedge E_r(x, y) \wedge S_j^1(y)) \longrightarrow S_i^1(x)$

k-pebble acceptors and MSO (cont.)

Proof (cont.)

$k = 2$:

- reverse-closed^1 and reverse-closed^2 describe reverse closure for configurations with one and two pebbles, respectively
- reverse-closed^2 expresses the same as reverse-closed before, but with the (immobile) pebble 1 represented by variable x_1
- reverse-closed^1 also refers to subcomputations with a second pebble
- Conjuncts corresponding to simple movements are essentially the same
- Conditions which check whether pebbles are at the same node have to be added
- The following conjuncts are added for pebble placement and lifting:
  - $(q_{2i}, a) \longrightarrow (q_{1j}, \text{lift})$ adds $\forall x_2\, (Q_a(x_2) \wedge S_j^1(x_1)) \longrightarrow S_i^2(x_2)$ to reverse-closed^2
  - $(q_{1i}, a) \longrightarrow (q_{2j}, \text{place})$ adds $\forall x_1\, (Q_a(x_1) \wedge \varphi^2) \longrightarrow S_i^1(x_1)$ to reverse-closed^1, where φ^2 is $\forall S_1^2 \cdots \forall S_m^2 (\text{reverse-closed}^2 \longrightarrow S_j^2(\text{root}))$

Summary of Proof

Proof (cont.)

- To solve the type checking problem, given d_1, d_2 and T, we can proceed as follows.
  1. Construct the k-pebble acceptor A for $T^{-1}(\overline{L(d_2)})$
  2. Transform A into an equivalent MSO formula Φ
  3. Φ holds for all trees t for which $T(t) \not\subseteq L(d_2)$
  4. Construct a nondeterministic bottom-up automaton A' equivalent to $\neg\Phi$
  5. Check that $L(d_1) \subseteq L(A')$
- Hence, the type-checking problem is decidable
- Steps (1) and (4) can be done in poly-time
- Step (2) is exponential in k, FO-quantifier depth of Φ is k, MSO-quantifier depth of Φ is $|Q|$
- Step (3) is non-elementary (exponentiation tower of height k)
- Hence,
  the algorithm for the type-checking problem has a very bad complexity

Summary: Typechecking

Related Work

- If transformations are allowed to compare data values in the input document type checking becomes undecidable very quickly, even for restricted types and transformations [Alon et al. 2001]
- Typechecking for deterministic top-down tree transducers is more tractable. Complexity depends on exact representation of DTDs and restrictions on the transducers: between **PTIME** and **EXPTIME** [Martens and Neven 2003]
- If **P** $\neq$ **NP** there is no elementary f, such that MSO-formulas can be evaluated in time $f(|\text{formula}|) \times p(|\text{tree}|)$ with polynomial p [Frick and Grohe 2002]

Open

- Find (more) transformations with a tractable typechecking problem
- In particular, with data values

Contents

Theory of `XQuery`

Remarks

- In general, the theoretical foundations of `XQuery` have to be developed
- Clearly: `XQuery` is Turing-complete and therefore static analysis is generally impossible
- What about important fragments with better properties?
- E.g., Tree pattern queries
- Here, we concentrate on:
  - Conjunctive queries for trees
  - Some questions related to automata for `XQuery`

Contents

Conjunctive Queries

Introduction

- Navigational `XPath` expressions (without `or` and `not`) can be written as **conjunctive queries**
- `/child::a/desc::*[child::c]/parent::d` corresponds to

$$Q(x) = \mathrm{root}(x_1) \wedge \mathtt{child}(x_1, x_2) \wedge L_a(x_2) \wedge \mathtt{desc}(x_2, x_3) \wedge \mathtt{child}(x_3, x_4) \wedge L_c(x_4) \wedge \mathtt{child}(x, x_3) \wedge L_d(x)$$

- Conjunctive Queries can express queries of higher arity:

$$Q(x, y) = \mathtt{child}(x, x_1) \wedge \mathtt{child}(x_1, y)$$

- What is the complexity of evaluating conjunctive queries on trees?
- Data complexity is in **PTIME** (even in **LOGSPACE**):

  Cycle through all valuations of the variables

- What about combined complexity?

A Generic Algorithm

Definition

- **Pre-valuation**: mapping from variables to non-empty sets of nodes
- For a conjunctive query Q a pre-valuation θ is **consistent** if:
  - for each atom $L_\sigma(x)$: $v \in \theta(x) \Rightarrow L_\sigma(v)$
  - for each atom $R(x,y)$:
    - $v \in \theta(x) \Rightarrow \exists u \in \theta(y) R(v,u)$
    - $v \in \theta(y) \Rightarrow \exists u \in \theta(x) R(u,v)$

Example Query

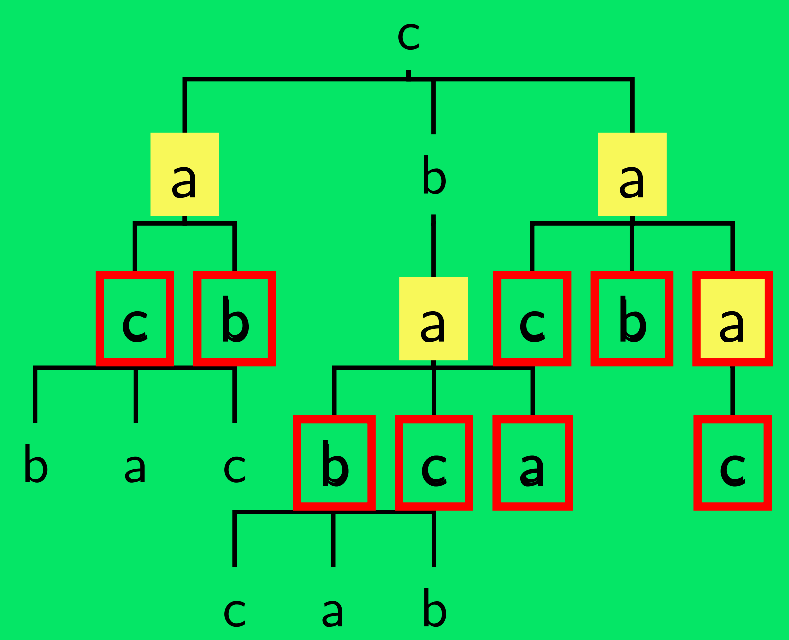
$$Q(x,y) = \texttt{child}(x,y) \wedge L_a(x)$$

Example Pre-Valuation

$\theta(x) = \ldots$

$\theta(y) = \boxed{\ldots}$

Example Document



A Generic Algorithm (cont.)

Fact

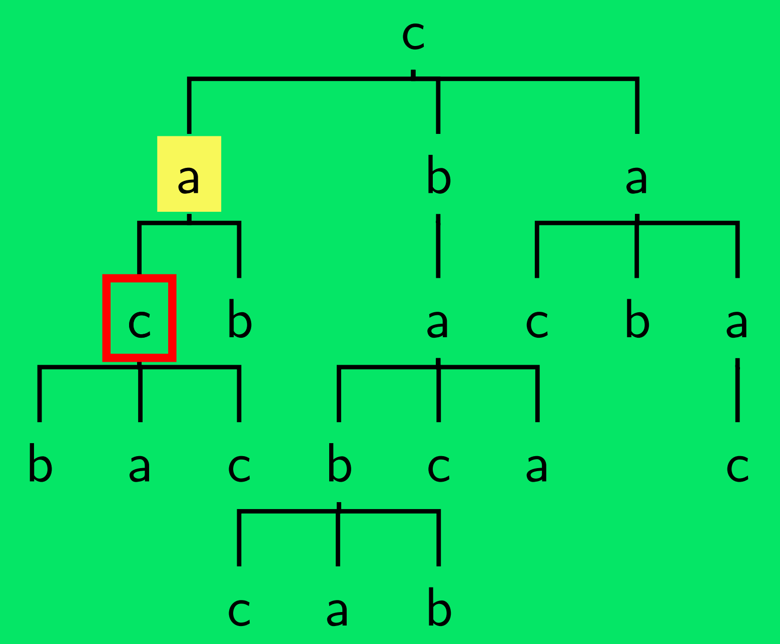
A maximal consistent pre-valuation θ can be computed in time

$$O(\text{query size} \times \text{tree size})$$

Algorithmic Idea

- Let $<$ be a total order on the nodes
- For query Q and tree t:
  Compute maximal consistent pre-valuation θ
  Define $<$-minimal valuation h via:
  For each variable x:
  $h(x) :=$ minimal node in $\theta(x)$ wrt $<$

Example Document



Let $<$ be the breadth-first left-to-right order

Question: Is h always a solution?

A Generic Algorithm (cont.)

Example Query

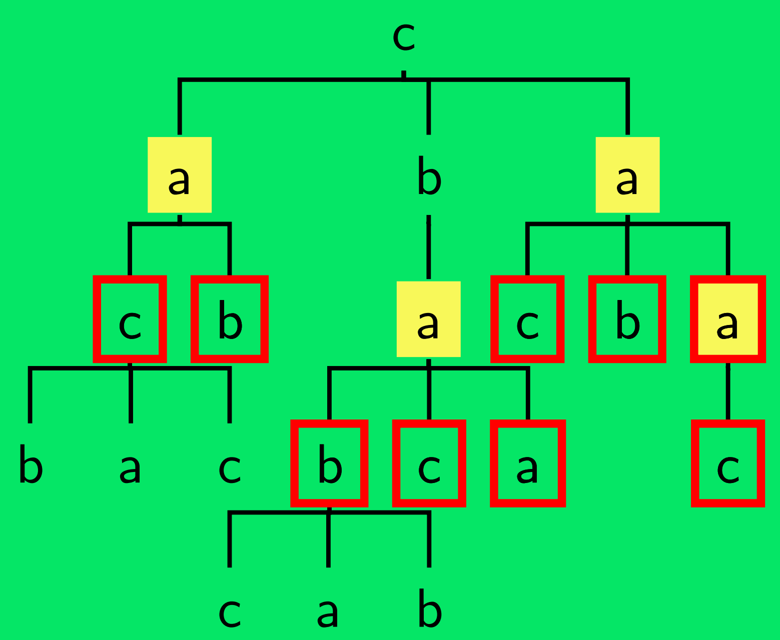
$$Q(x,y) = \mathtt{child}(x,y) \wedge L_a(x)$$

Example Pre-Valuation

$$\theta(x) = \dots$$

$$\theta(y) = \boxed{\dots}$$

Example Document



Question: Is h always a solution?

Observations

- Let $u = h(x), v = h(y)$
- As $u \in \theta(x)$ there is $v' \in \theta(y)$ such that $\mathtt{child}(u, v')$
- As $v \in \theta(y)$ there is $u' \in \theta(x)$ such that $\mathtt{child}(u', v)$
- As $u \leq u'$ and $v \leq v'$ we get $\mathtt{child}(u, v)$

A Generic Algorithm (cont.)

Definition

A binary relation R is **$<$-hemichordal** if for all u, u', v, v' with $u < u'$ and $u \leq v \leq v'$

- $R(u, v') \wedge R(u', v) \rightarrow R(u, v)$ and
- $R(v', u) \wedge R(v, u') \rightarrow R(v, u)$

Theorem [Gottlob, Koch, Schulz 04]

If the relations of a query Q are $<$-hemichordal and θ is a consistent pre-valuation for Q
then the $<$-minimal valuation for θ is a solution for Q

Corollary

If the axes used in a conjunctive query Q are $<$-hemichordal then Q can be evaluated in time $O(\text{query size} \times \text{tree size})$

Combined Complexity of Conjunctive Queries

Observation

It is sufficient to consider the axes `child`, $\texttt{child}^+$, $\texttt{child}^*$, NextSibling, NextSibling^+, NextSibling^*, `Following`

Theorem [Gottlob, Koch, Schulz 04]

- $\texttt{child}^+$ and $\texttt{child}^*$ are preorder-hemichordal
- `following` is postorder-hemichordal
- `child`, NextSibling, NextSibling^+, NextSibling^* are breadth-first-left-to-right-hemichordal

Corollary

For each of these sets of axes conjunctive queries can be evaluated in time $O(\text{query size} \times \text{tree size})$

Amazing Result

For sets of axes not contained in those, the combined complexity of conjunctive query evaluation is **NP**-complete

Contents

Automata and XQuery

So far...

- We have seen that automata are useful for
  - Validation, Typing
  - Navigation
  - Transformation
- What about more general queries?
  - results of higher arity?
  - joins, i.e., comparisons of data values
  - counting
- Are automata useful for `XQuery`?
- ... for tree pattern queries?

General Queries (cont.)

Higher arity

- Nonemptiness and containment questions can be handled by automata: tuples can be encoded by additional labels
- What about query evaluation for higher arity?

Data values

- When data values in XML documents are taken into account, things become more complicated, e.g.:
  - Even First-order logic becomes undecidable
  - Pebble automata become undecidable
  - Automata with data registers become undecidable when they are allowed to move up and down
- What is the right notion for regular (string) languages over infinite alphabets?
- What are sensible decidable restrictions of logics and automata in the context of data values?

General Queries (cont.)

Counting

- Automata can be equipped with counting facilities, e.g.:
  Presburger tree automata: $\delta(\sigma, q)$ is Boolean combination of
  - regular expressions and
  - quantifier-free Presburger formulas like
    "number of children in state q_1 $=$ number of children in state q_2"
- Nondet. Presburger automata:
  - $\equiv$ MSO logic
  - Whether automaton accepts all trees is undecidable
- Det. Presburger automata:
  - $\equiv$ Presburger μ-formulas
  - Membership test: $O(|\mathcal{A}||t|)$
  - Non-emptiness: **PSPACE**
  - Containment: **PSPACE**

[Seidl, Sch., Muscholl, Habermehl 2004]

Contents

Conclusion

Summary

- Schema languages and `XPath` are well understood
- There are some nice results on transformations
- Theory for `XQuery` still has to be developed

Finally...

Thanks for your patience