Logic and XML

München

May 2005

Thomas Schwentick

Contents

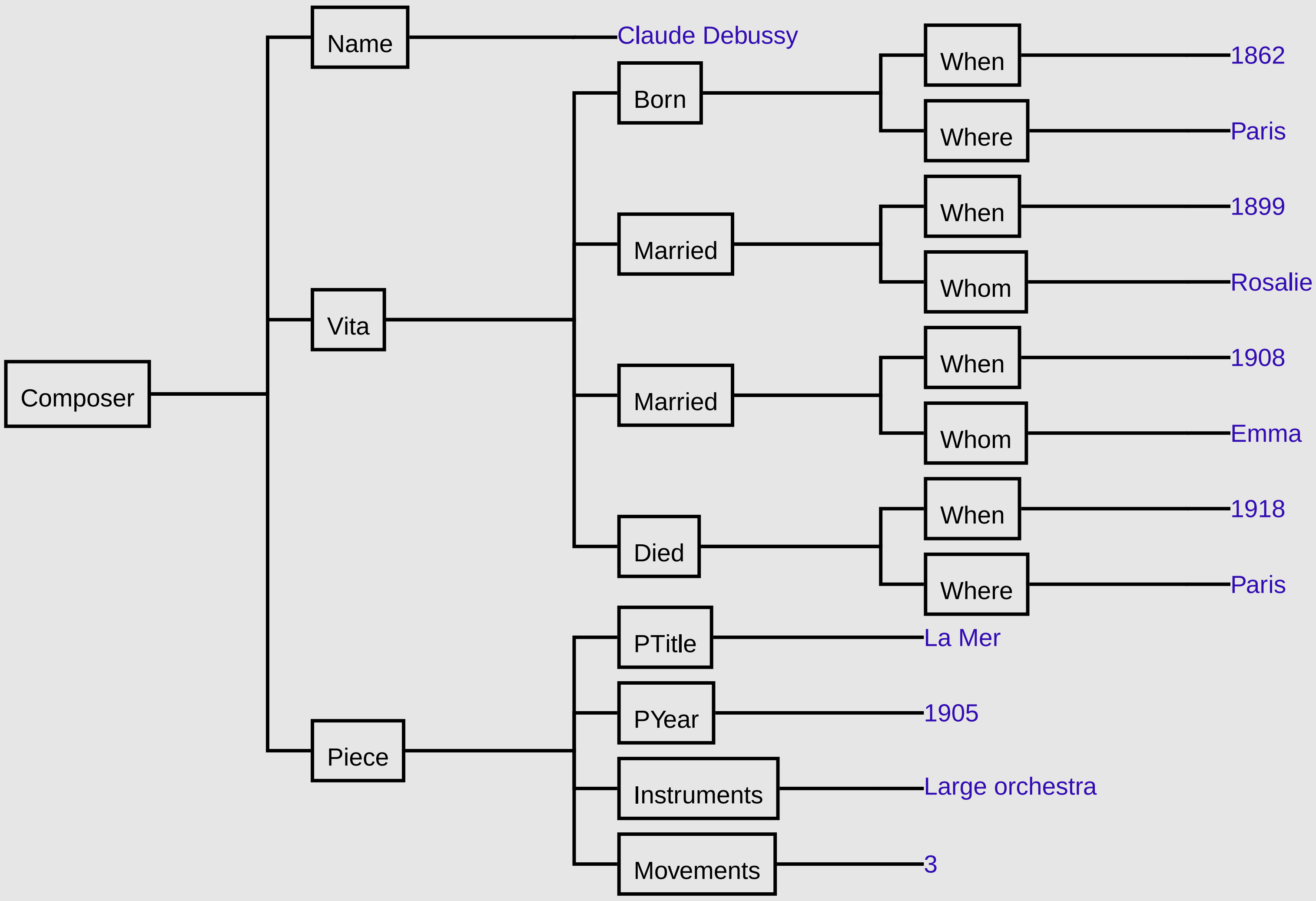
XML

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⟩
...
```

XML

Example Tree



XML Processing

Four important kinds of XML processing

Validation

Check whether an XML document is of a given type

Navigation

Select a set of positions in an XML document

Querying

Extract information from an XML document

Transformation

Construct a new XML document from a given one

XML Processing

Four important kinds of XML processing	and their languages
Validation Check whether an XML document is of a given type	`DTD`, `XML Schema`
Navigation Select a set of positions in an XML document	`XPath`
Querying Extract information from an XML document	`XQuery`
Transformation Construct a new XML document from a given one	`XSLT`

Validation: DTD

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⟩
        ...
```

Validation: DTD

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⟩
...

DTD

DTDs describe types of XML documents

Validation: DTD

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⟩
...

DTD

DTDs describe types of XML documents

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

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⟩
          ...
```

Navigation: XPath

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⟩
...

XPath

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

Navigation: XPath

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⟩
...

XPath

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

Example query

```
//Vita/Died/*
```

Navigation: XPath

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⟩
...

XPath

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

Example query

```
//Vita/Died/*
```

Navigation: XPath

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⟩
...

XPath

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

Example query

```
//Vita/Died/*
```

Remark

XPath expressions define sets of nodes:

node-selecting queries

Querying: `XQuery`

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⟩
        ...
```

Querying: `XQuery`

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⟩
        ...
```

`XQuery`

`XQuery` is a full-fledged XML query language

Querying: XQuery

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⟩
...

XQuery

XQuery is a full-fledged XML query language

Example query

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

Querying: XQuery

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⟩
...

XQuery

XQuery is a full-fledged XML query language

Example query

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

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⟩
...
```

Transformation: XSLT

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⟩
        ...

Transformation: XSLT

Example document	XSLT
⟨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⟩ ...	XSLT transforms documents by means of templates

Transformation: XSLT

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⟩
...

XSLT

XSLT transforms documents by means of templates

Example

⟨xsl:template match=”Composer[Vita//Where=’Paris’]”⟩
⟨ParisComposer⟩
⟨xsl:copy-of select=“Name”/⟩
⟨xsl:copy-of select=“Vita/Born”/⟩
⟨/ParisComposer⟩
⟨/xsl:template⟩

Transformation: XSLT

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⟩
        ...
```

XSLT

XSLT transforms documents by means of templates

Example

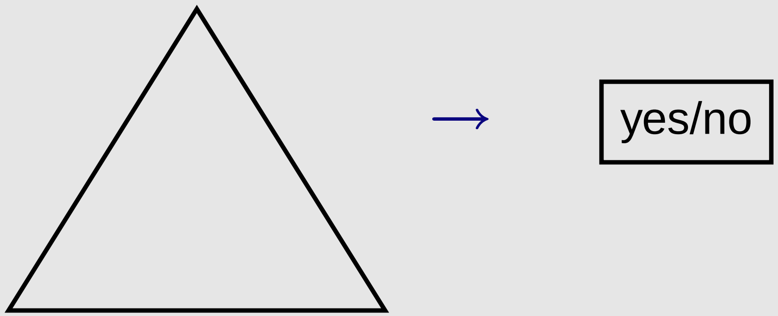
```
⟨xsl:template match=”Composer[Vita//Where=’Paris’]”⟩
  ⟨ParisComposer⟩
    ⟨xsl:copy-of select=“Name”/⟩
    ⟨xsl:copy-of select=“Vita/Born”/⟩
  ⟨/ParisComposer⟩
⟨/xsl:template⟩
```

Result

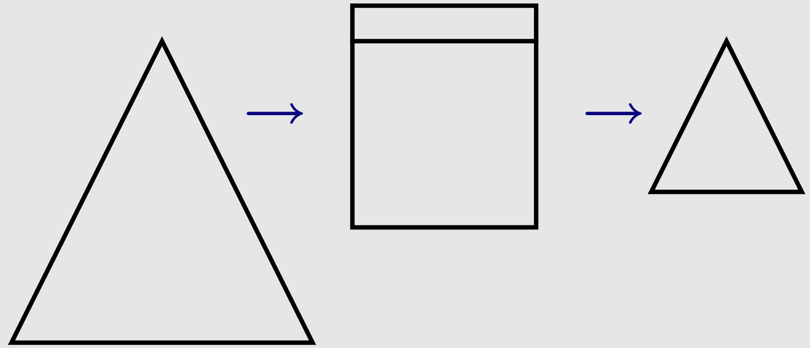
```
⟨ParisComposer⟩⟨Name⟩ Claude Debussy ⟨/Name⟩
  ⟨Born⟩ ⟨When⟩ August 22, 1862 ⟨/When⟩
    ⟨Where⟩ Paris ⟨/Where⟩
  ⟨/Born⟩⟨/ParisComposer⟩
⟨ParisComposer⟩ ⟨Name⟩ Frédéric Chopin ⟨/Name⟩
  ⟨Born⟩ ⟨When⟩ March 1, 1810 ⟨/When⟩
    ⟨Where⟩ Želazowa ⟨/Where⟩
  ⟨/Born⟩⟨/ParisComposer⟩
⟨ParisComposer⟩ ⟨Name⟩ Camille Saint-Saëns ⟨/Name⟩
  ⟨Born⟩ ⟨When⟩ October 9, 1835 ⟨/When⟩
  ⟨Where⟩ Paris ⟨/Where⟩
⟨/Born⟩ ⟨/ParisComposer⟩
```

A Schematic View

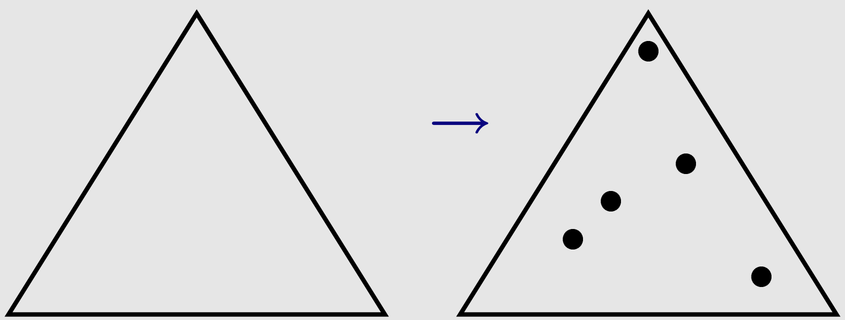
DTD/ XML Schema



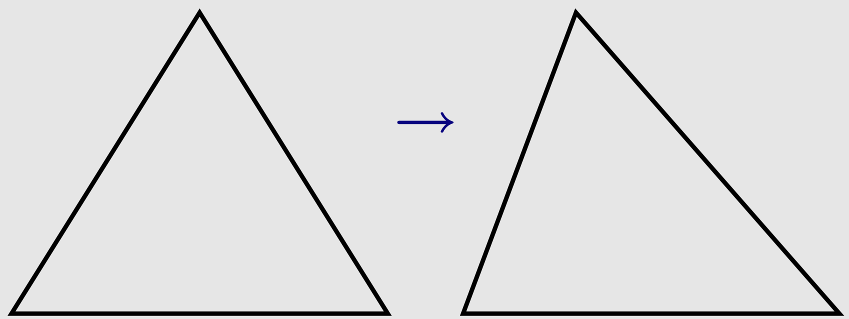
XQuery



XPath

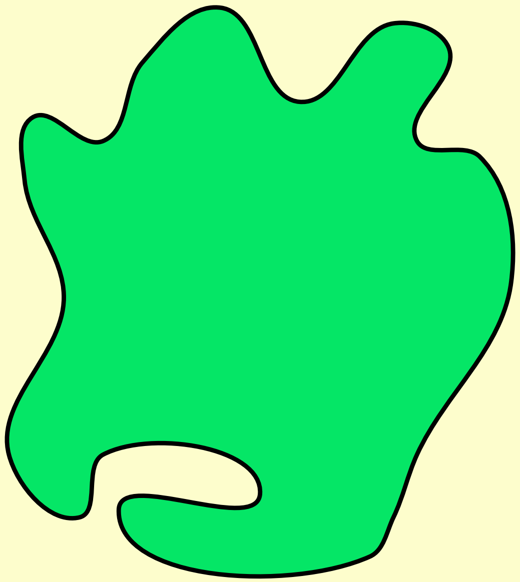


XSLT

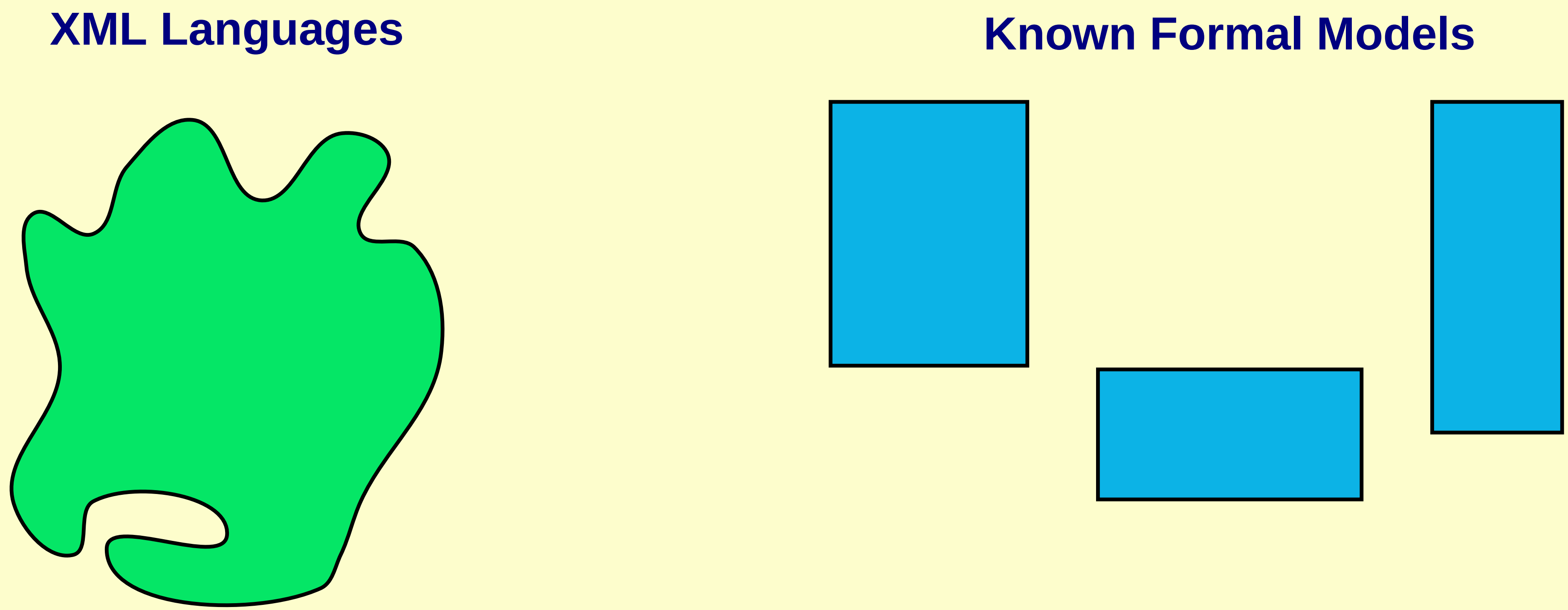


The Big Picture

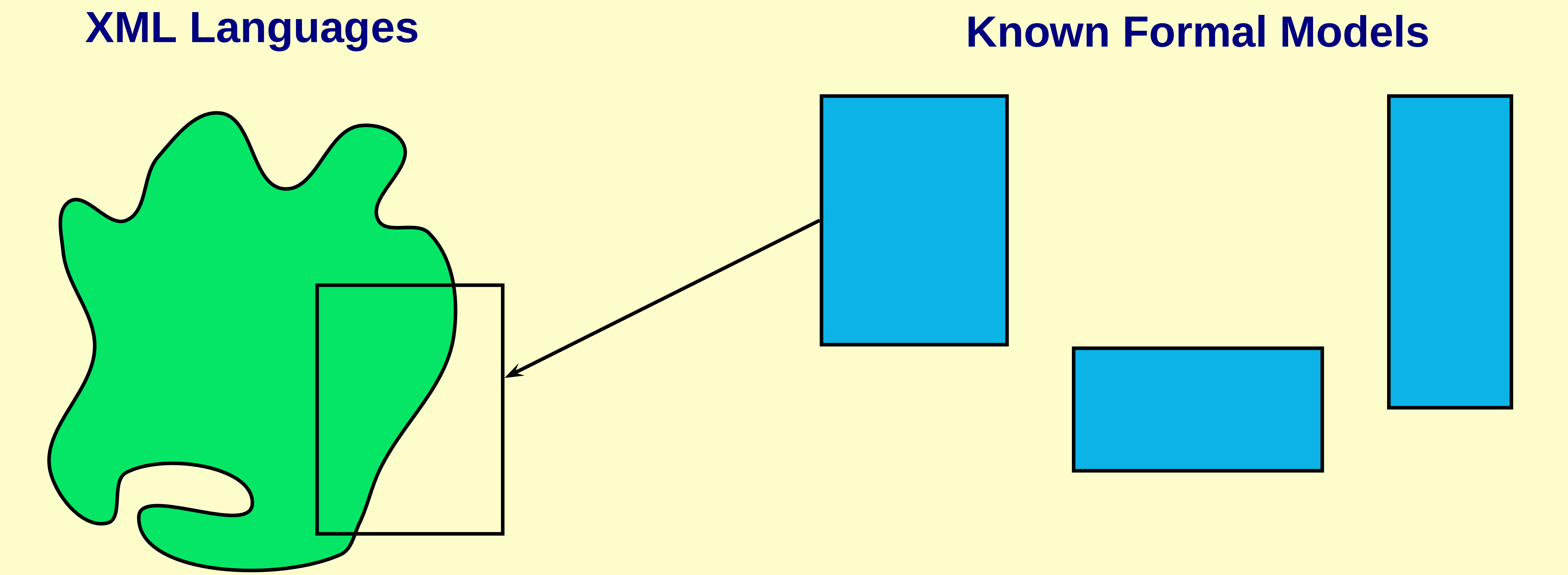
XML Languages



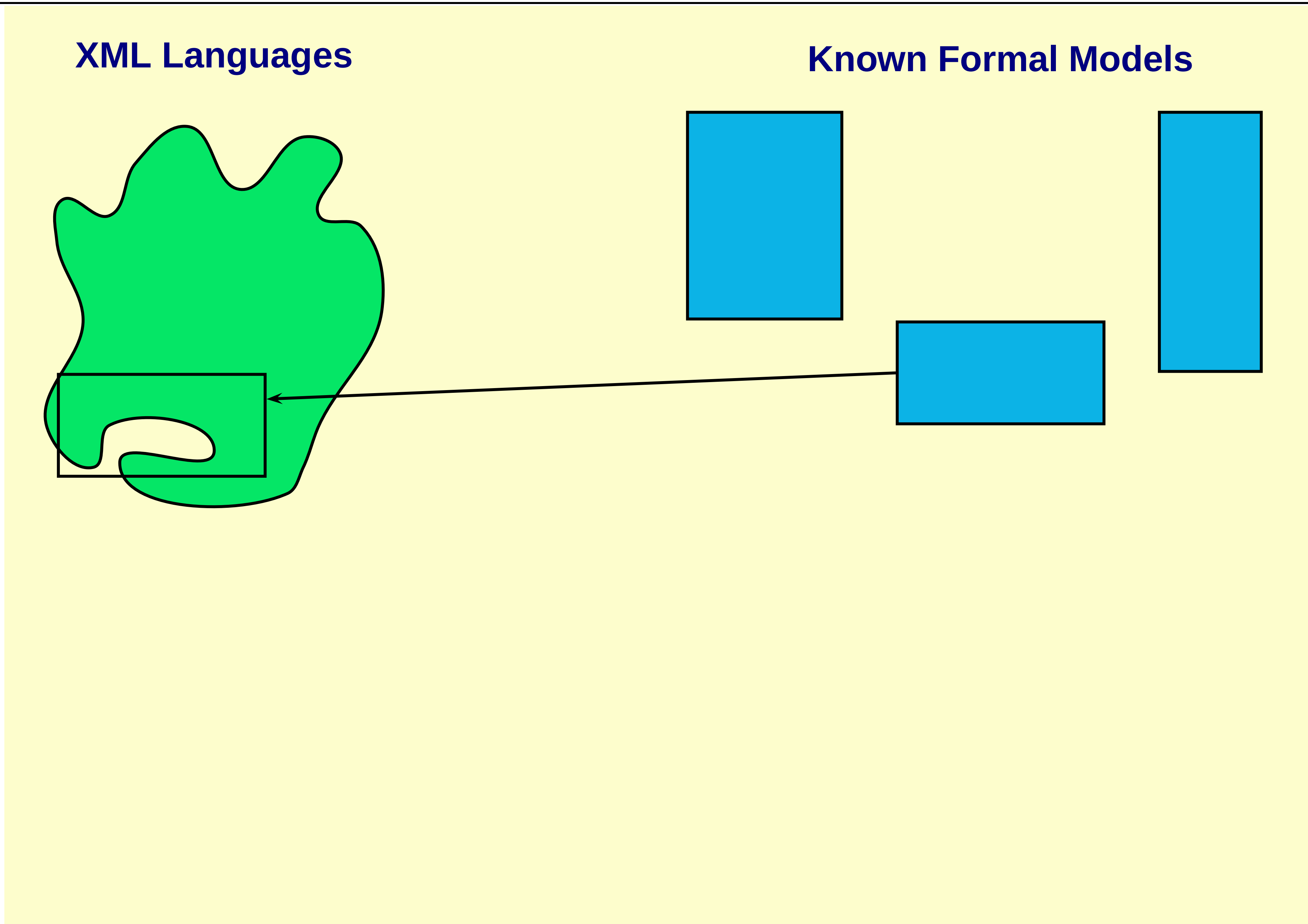
The Big Picture



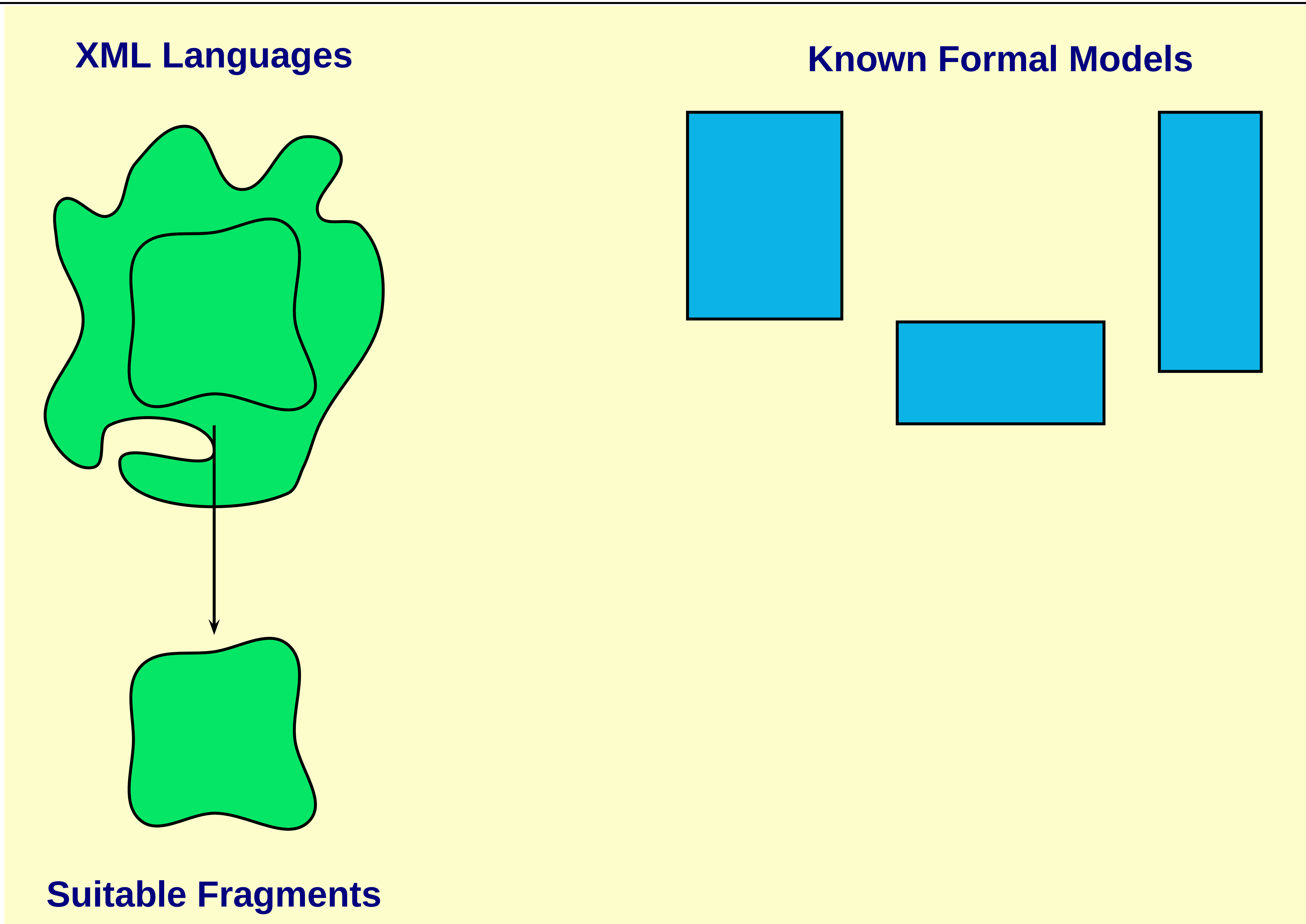
The Big Picture



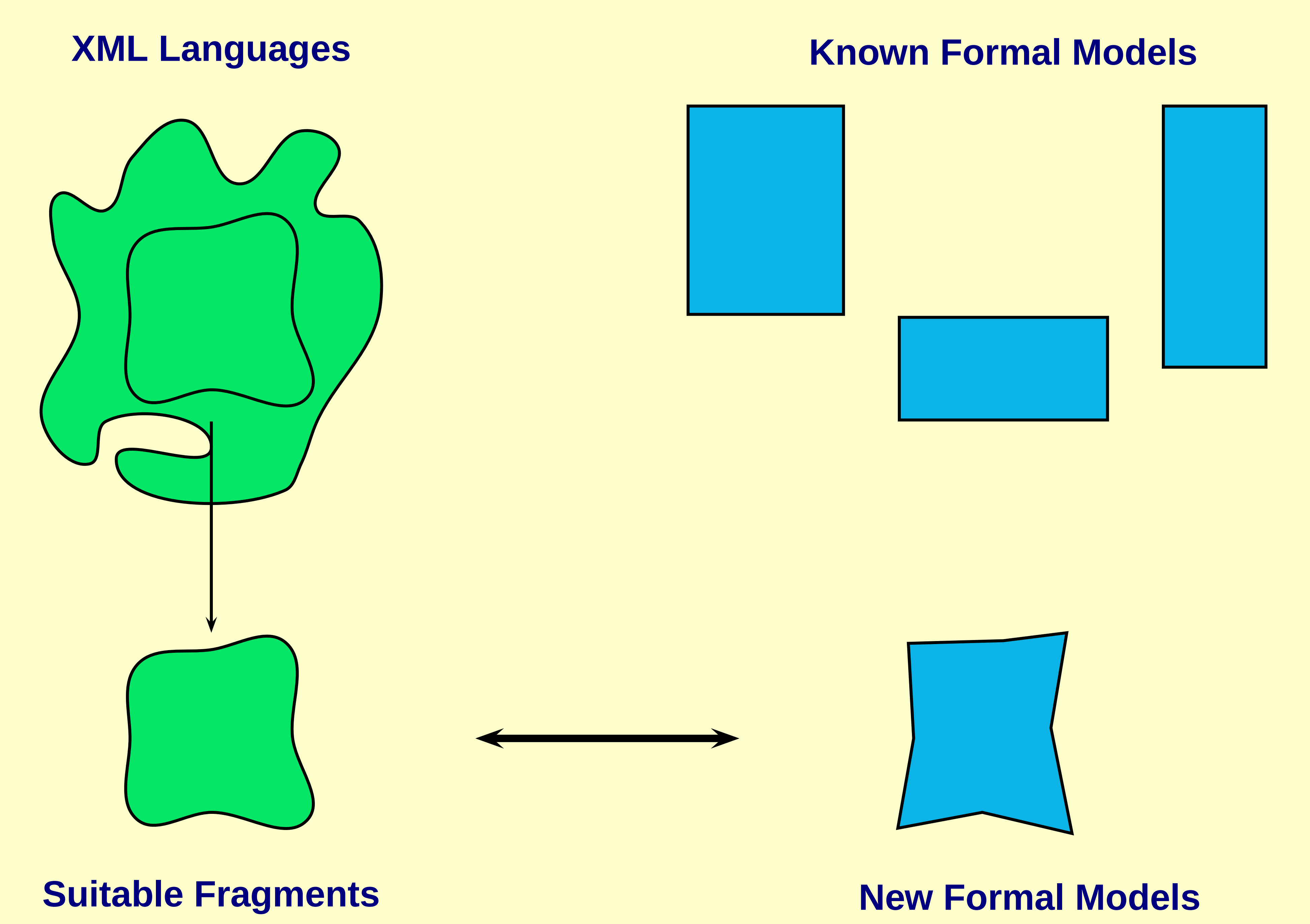
The Big Picture



The Big Picture



The Big Picture



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,

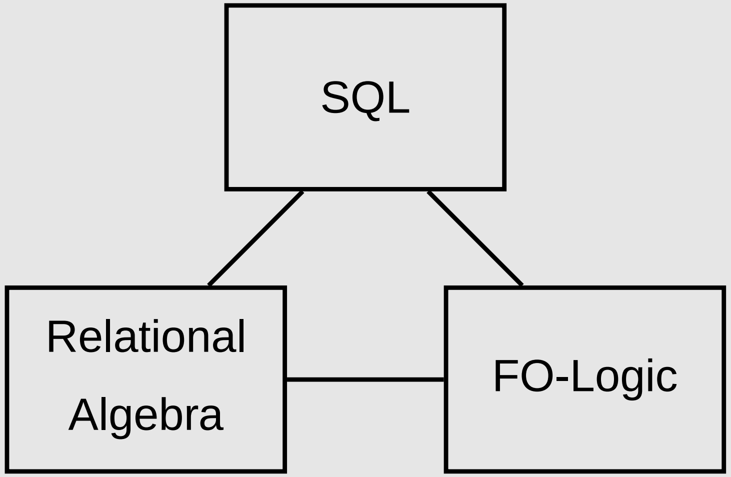
Transformation T

O: Type of

$\{T(t) \mid t \models d\}$

A Look Back

Relational Databases



Basic Properties of these Formalisms

SQL

- Declarative, easy to use
- Queries, data definition, updates

FO-logic

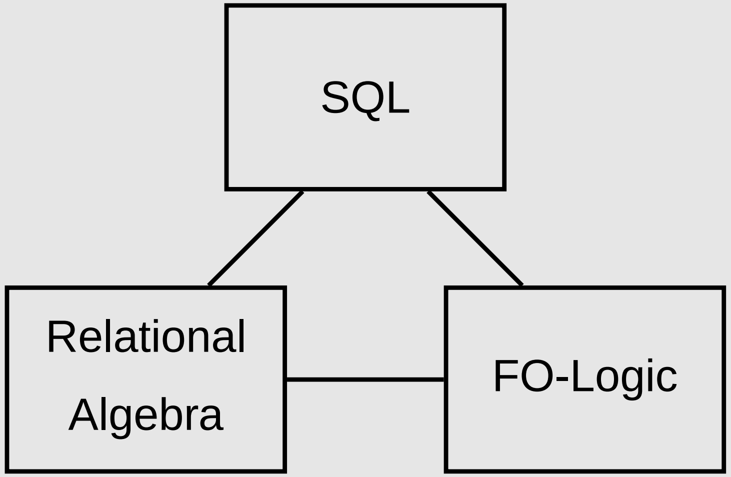
- Formal framework for investigations
- Clear Semantics
- Expressive power well understood

Relational Algebra

- Operational model
- Flexible, optimizable
- Automatic translation

A Look Back

Relational Databases



Basic Properties of these Formalisms

SQL

- Declarative, easy to use
- Queries, data definition, updates

FO-logic

- Formal framework for investigations
- Clear Semantics
- Expressive power well understood

Relational Algebra

- Operational model
- Flexible, optimizable
- Automatic translation

Further Properties

- Satisfiability undecidable

$\longrightarrow$ Fragments like conjunctive queries

- Evaluation for conjunctive queries **NP**-hard

$\longrightarrow$ but works well in practice

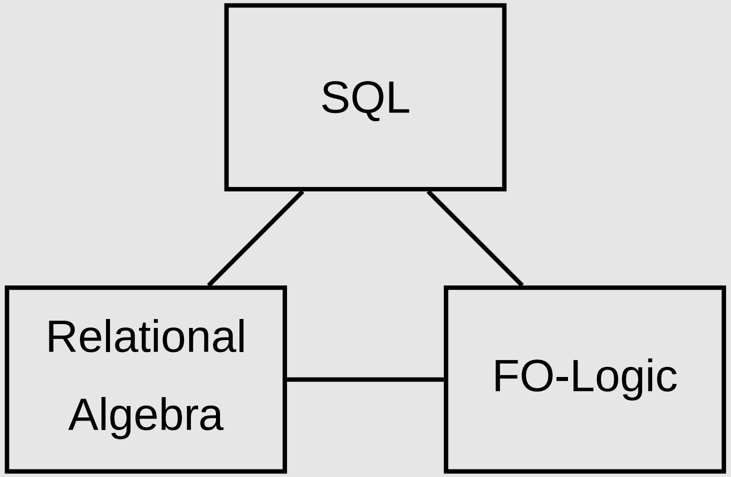
- SQL can count and group

$\longrightarrow$ Can be added to FO

[Hella et al. 01]

A Look Back

Relational Databases



Basic Properties of these Formalisms

SQL

- Declarative, easy to use
- Queries, data definition, updates

FO-logic

- Formal framework for investigations
- Clear Semantics
- Expressive power well understood

Relational Algebra

- Operational model
- Flexible, optimizable
- Automatic translation

Further Properties

- Satisfiability undecidable

$\longrightarrow$ Fragments like conjunctive queries

- Evaluation for conjunctive queries **NP**-hard

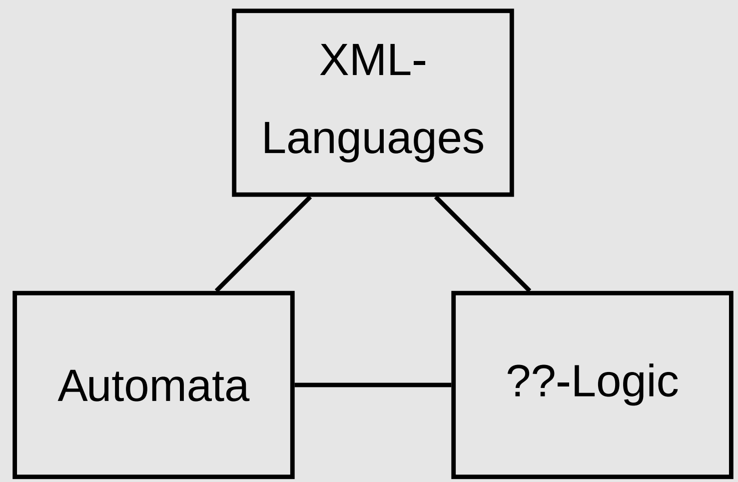
$\longrightarrow$ but works well in practice

- SQL can count and group

$\longrightarrow$ Can be added to FO

[Hella et al. 01]

Goal



Contents

XML as Tree

Example Document

⟨a⟩⟨b⟩

⟨d⟩12⟨/d⟩⟨e⟩22⟨/e⟩

⟨/b⟩

⟨b⟩⟨d⟩4⟨/d⟩⟨/b⟩

⟨b⟩⟨d⟩11⟨/d⟩

⟨e⟩8⟨/e⟩18⟨e⟩5⟨/e⟩

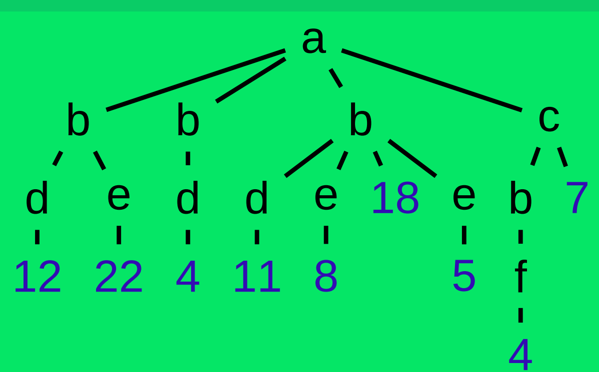
⟨/b⟩

⟨c⟩⟨b⟩⟨f⟩2⟨/f⟩

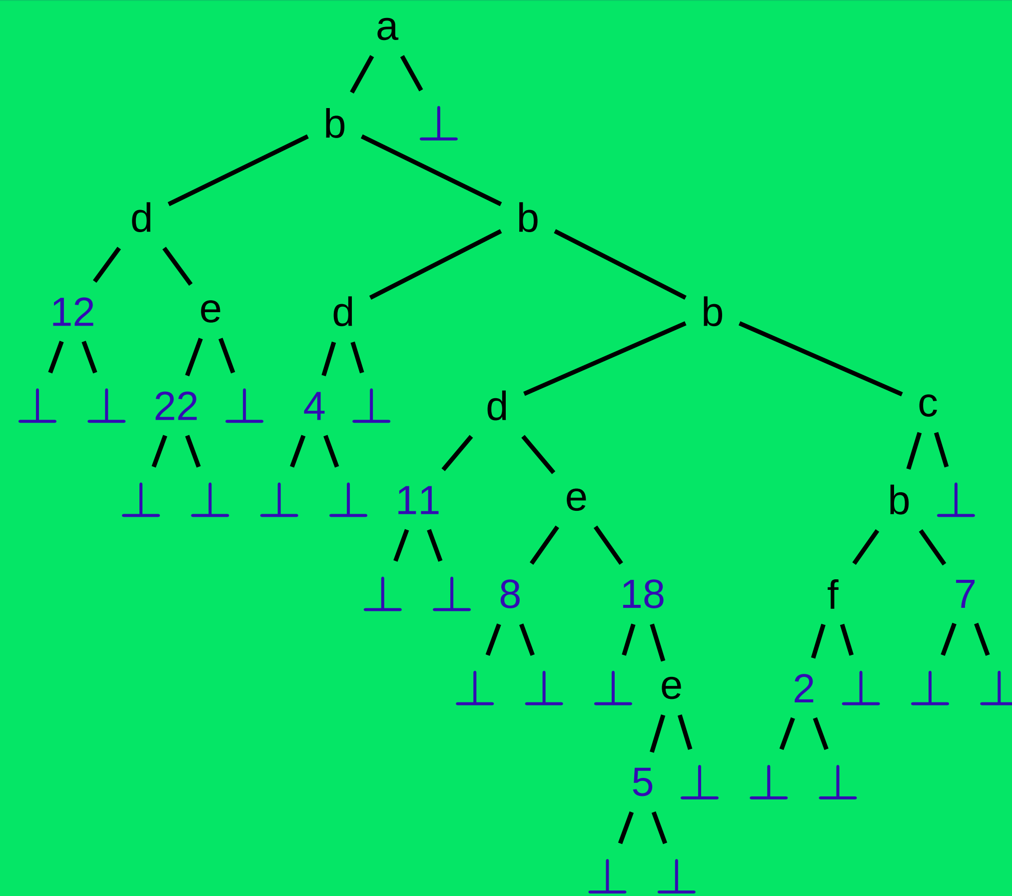
⟨/b⟩7⟨/c⟩

⟨/a⟩

...as Unranked Tree



...as Binary Tree

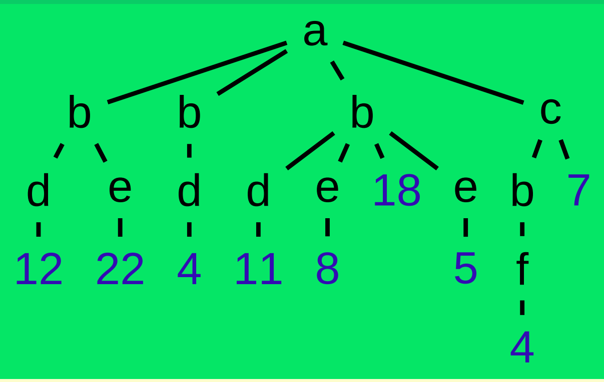


XML Trees as Finite Models

Data Values

For many investigations, data values can and have to be ignored

Example



Labels

- Usually, XML trees are modeled as labeled trees over a finite alphabet
- For schema languages this is ok
- For queries, the actual alphabet might depend on the query
- Unary predicates on data values can be also modeled this way

Signatures

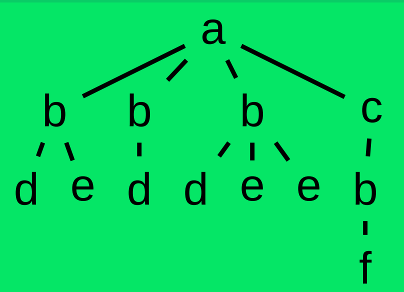
- There are various ways to represent unordered trees as finite relational models
- Possible relations
  - Local orders: $\longrightarrow$, $\downarrow$ $\longleftarrow$, $\uparrow$ $\swarrow$
  - Their transitive closures: $\longrightarrow^+$, $\downarrow^+$, $\longleftarrow^+$, $\uparrow^+$
  - Reflexive and transitive: $\longrightarrow^*$, $\downarrow^*$, $\longleftarrow^*$, $\uparrow^*$
  - Document order: $\uparrow^*\longrightarrow\downarrow^*$
- Frequent combinations:
  - $\longrightarrow$, $\downarrow$
  - $\longrightarrow^+$, $\downarrow^+$
  - $\longrightarrow$, $\downarrow$, $\longrightarrow^+$, $\downarrow^+$
  - $\longrightarrow$, $\swarrow$

XML Trees as Finite Models

Data Values

For many investigations, data values can and have to be ignored

Example



Labels

- Usually, XML trees are modeled as labeled trees over a finite alphabet
- For schema languages this is ok
- For queries, the actual alphabet might depend on the query
- Unary predicates on data values can be also modeled this way

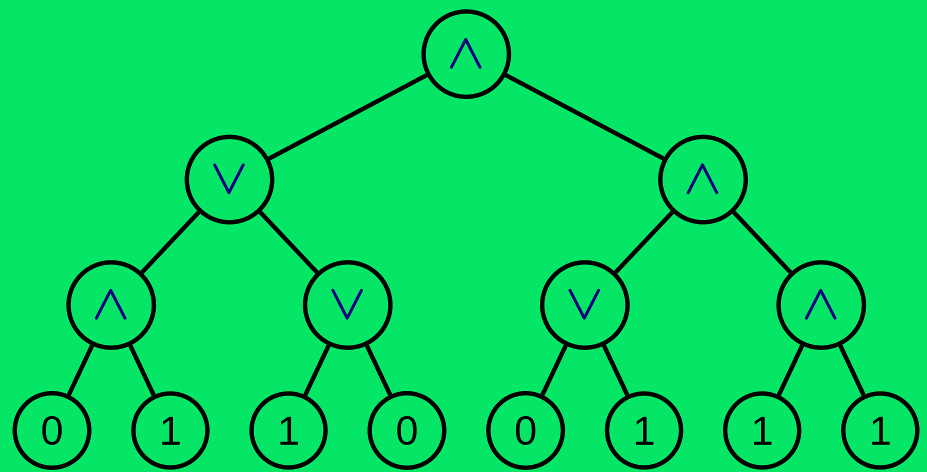
Signatures

- There are various ways to represent unordered trees as finite relational models
- Possible relations
  - Local orders: $\longrightarrow$, $\downarrow$ $\longleftarrow$, $\uparrow$ $\swarrow$
  - Their transitive closures: $\longrightarrow^+$, $\downarrow^+$, $\longleftarrow^+$, $\uparrow^+$
  - Reflexive and transitive: $\longrightarrow^*$, $\downarrow^*$, $\longleftarrow^*$, $\uparrow^*$
  - Document order: $\uparrow^*\longrightarrow\downarrow^*$
- Frequent combinations:
  - $\longrightarrow$, $\downarrow$
  - $\longrightarrow^+$, $\downarrow^+$
  - $\longrightarrow$, $\downarrow$, $\longrightarrow^+$, $\downarrow^+$
  - $\longrightarrow$, $\swarrow$

Contents

Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits



Idea

Two states: q_0, q_1

$q_1 \equiv$ subtree evaluates to 1

Transitions

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

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

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

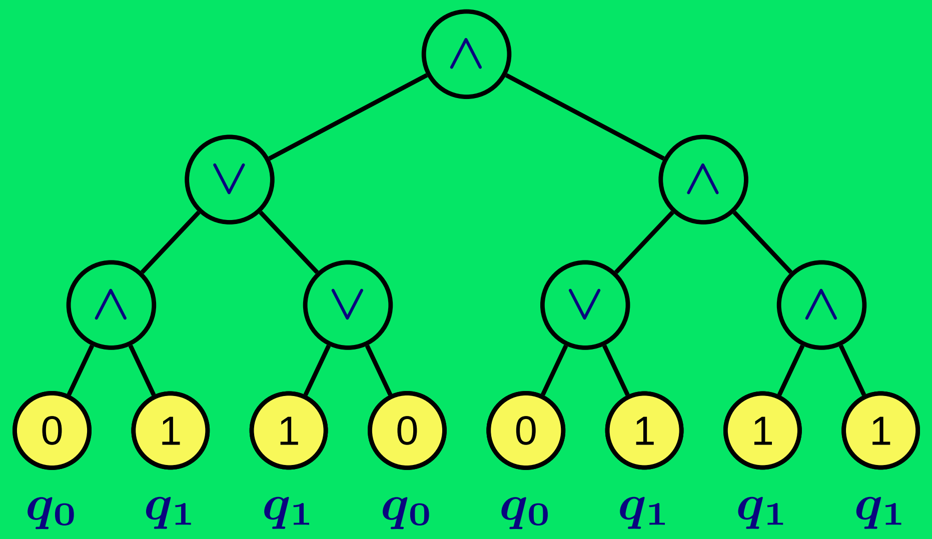
$\delta(\vee, q_0) = \{(q_0, q_0)\}$

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

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

Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits



Idea

Two states: q_0, q_1

$q_1 \equiv$ subtree evaluates to 1

Transitions

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

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

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

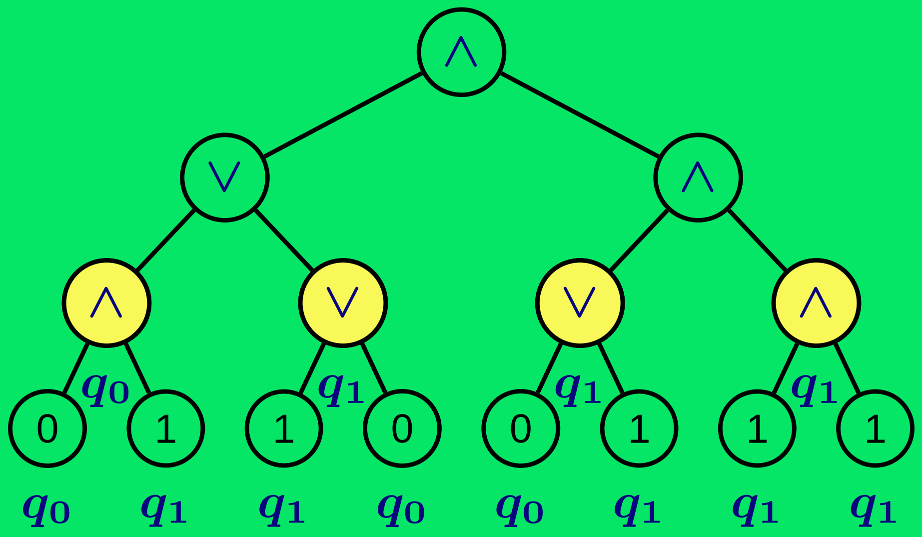
$\delta(\vee, q_0) = \{(q_0, q_0)\}$

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

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

Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits



Idea

Two states: q_0, q_1

$q_1 \equiv$ subtree evaluates to 1

Transitions

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

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

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

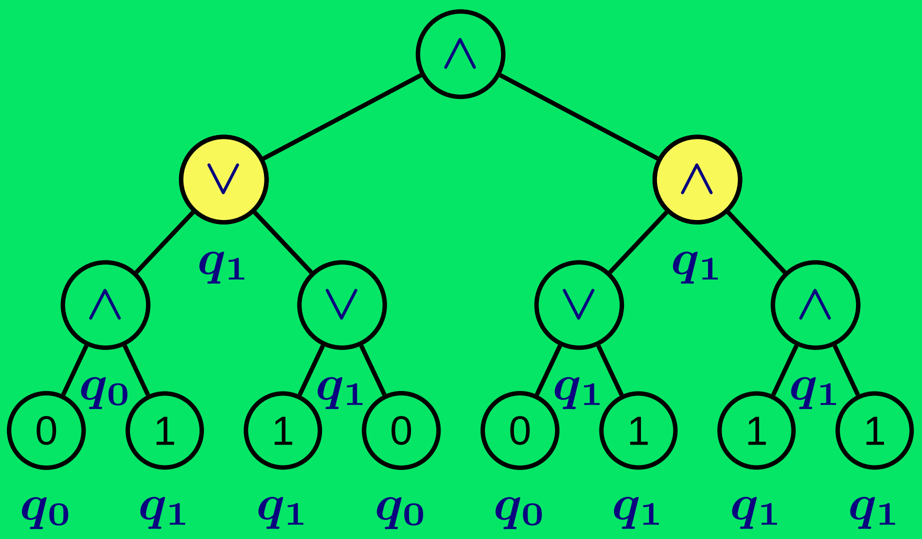
$\delta(\vee, q_0) = \{(q_0, q_0)\}$

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

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

Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits



Idea

Two states: q_0, q_1

$q_1 \equiv$ subtree evaluates to 1

Transitions

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

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

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

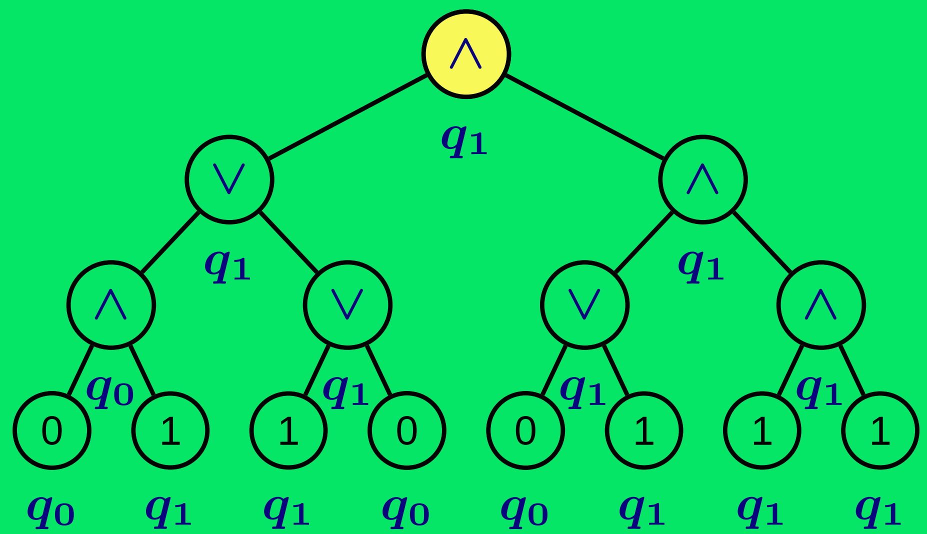
$\delta(\vee, q_0) = \{(q_0, q_0)\}$

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

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

Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits



Idea

Two states: q_0, q_1

$q_1 \equiv$ subtree evaluates to 1

Transitions

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

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

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

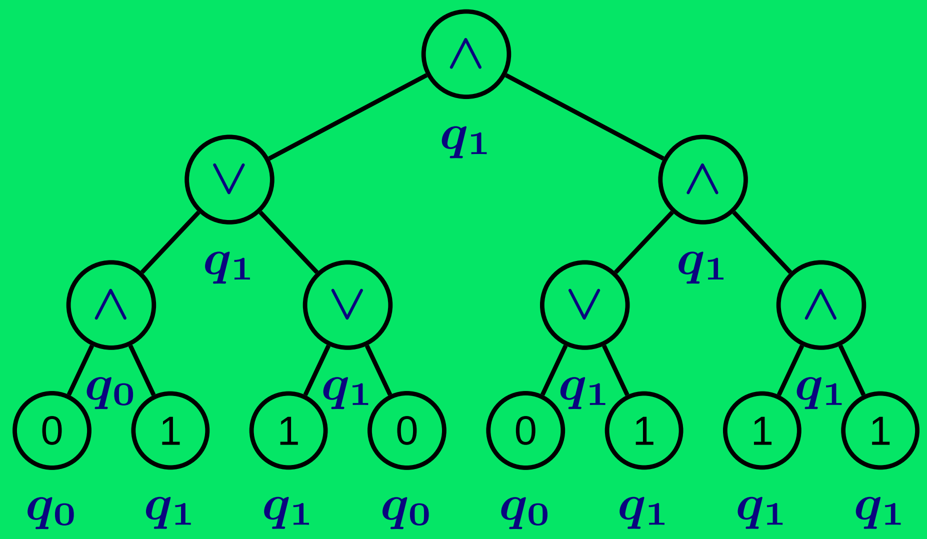
$\delta(\vee, q_0) = \{(q_0, q_0)\}$

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

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

Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits



Idea

Two states: q_0, q_1

$q_1 \equiv$ subtree evaluates to 1

Transitions

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

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

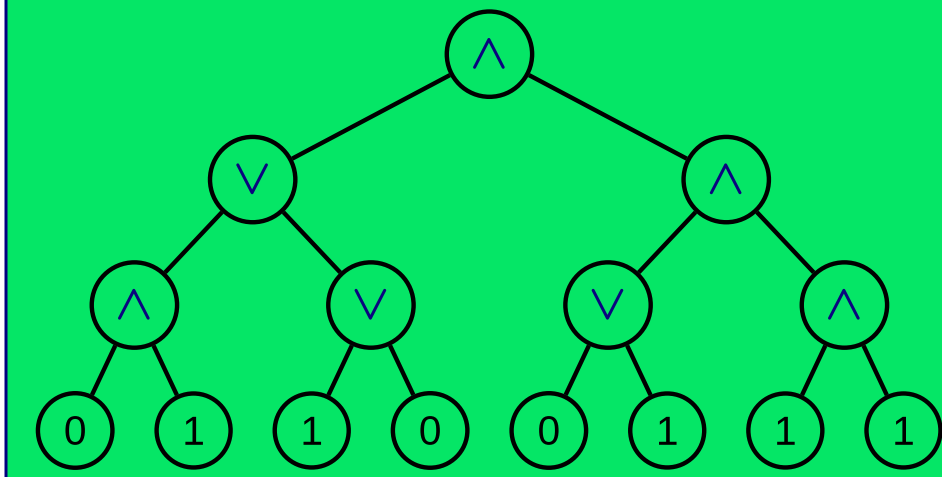
$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

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

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

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

Top-down Automaton for Boolean Circuits



Idea

Three states: q_0, q_1, acc

$q_1 \equiv$ subtree will be 1

Transitions

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

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

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

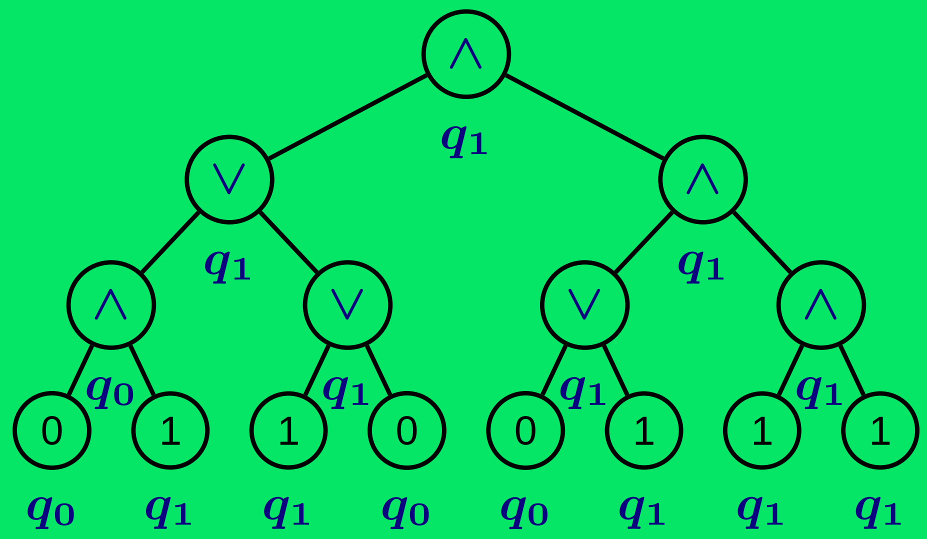
$\delta(\vee, q_0) = \{(q_0, q_0)\}$

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

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

Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits



Idea

Two states: q_0, q_1

$q_1 \equiv$ subtree evaluates to 1

Transitions

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

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

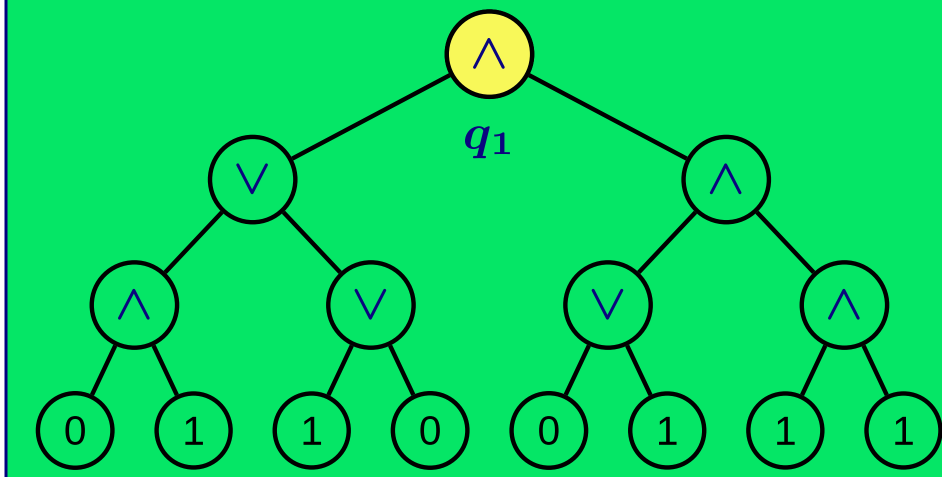
$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

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

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

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

Top-down Automaton for Boolean Circuits



Idea

Three states: q_0, q_1, acc

$q_1 \equiv$ subtree will be 1

Transitions

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

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

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

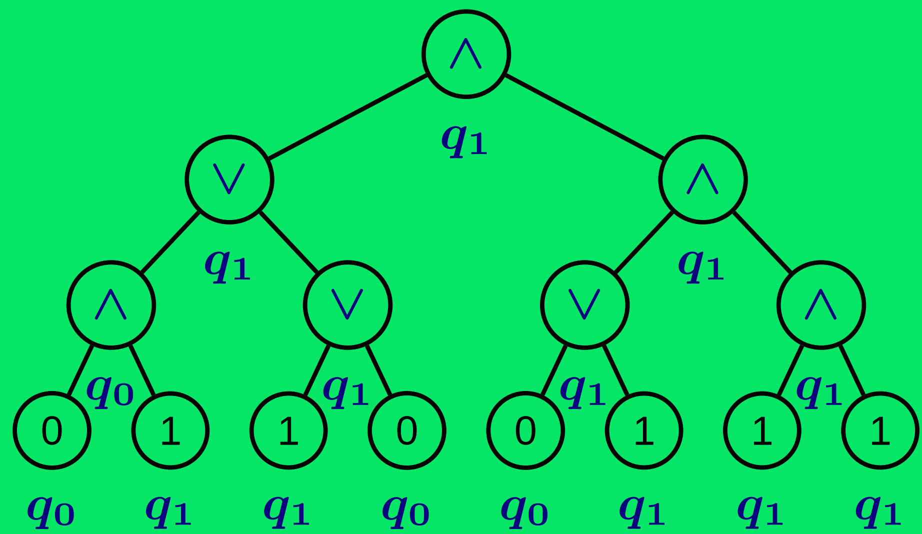
$\delta(\vee, q_0) = \{(q_0, q_0)\}$

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

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

Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits



Idea

Two states: q_0, q_1

$q_1 \equiv$ subtree evaluates to 1

Transitions

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

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

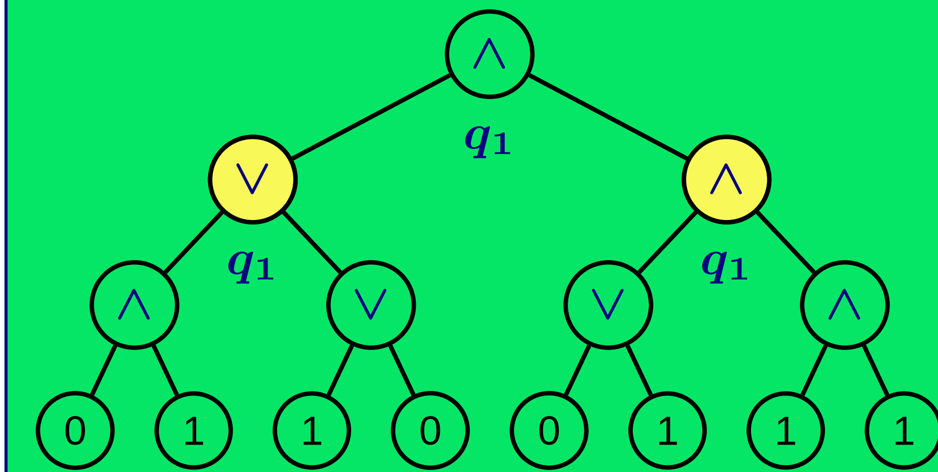
$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

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

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

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

Top-down Automaton for Boolean Circuits



Idea

Three states: q_0, q_1, acc

$q_1 \equiv$ subtree will be 1

Transitions

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

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

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

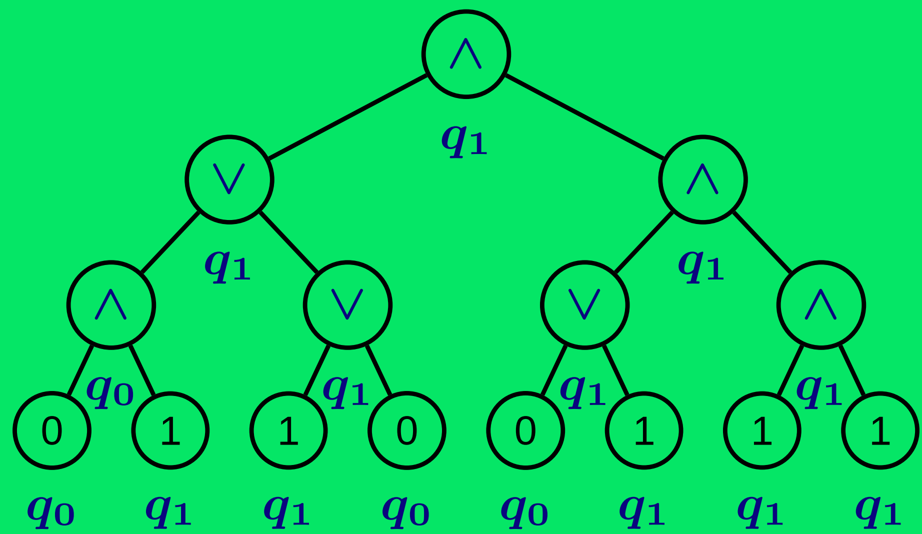
$\delta(\vee, q_0) = \{(q_0, q_0)\}$

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

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

Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits



Idea

Two states: q_0, q_1

$q_1 \equiv$ subtree evaluates to 1

Transitions

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

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

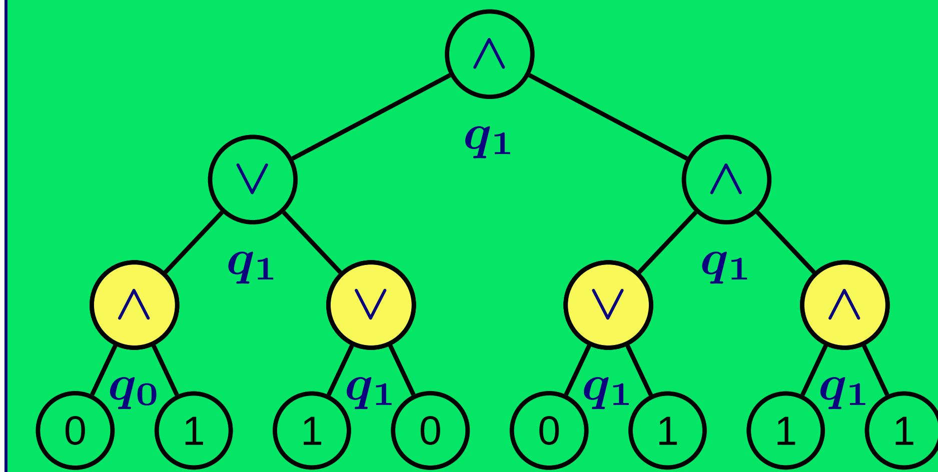
$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

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

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

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

Top-down Automaton for Boolean Circuits



Idea

Three states: q_0, q_1, acc

$q_1 \equiv$ subtree will be 1

Transitions

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

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

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

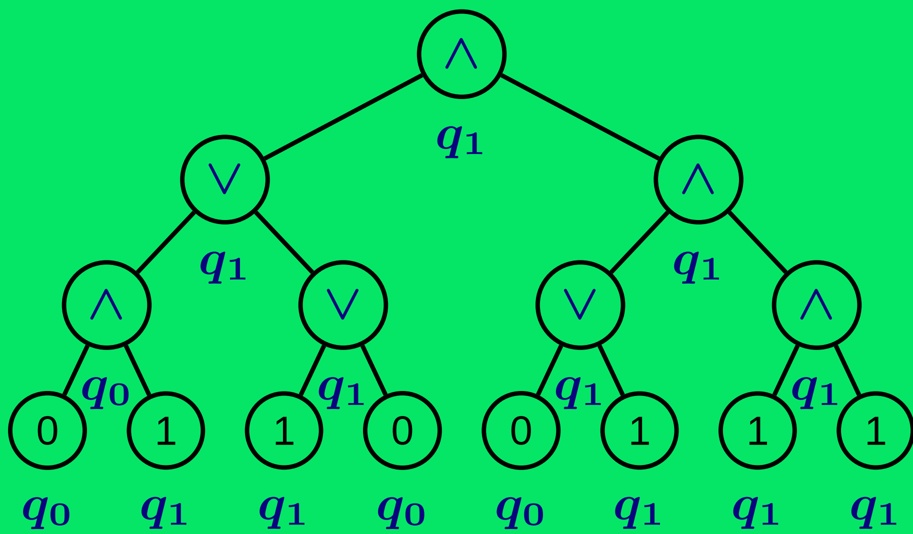
$\delta(\vee, q_0) = \{(q_0, q_0)\}$

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

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

Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits



Idea

Two states: q_0, q_1

$q_1 \equiv$ subtree evaluates to 1

Transitions

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

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

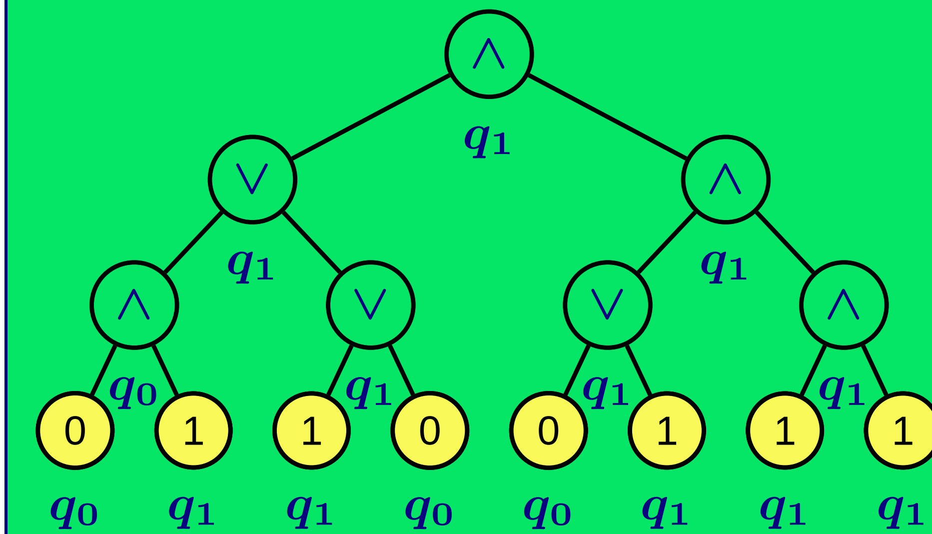
$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

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

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

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

Top-down Automaton for Boolean Circuits



Idea

Three states: q_0, q_1, acc

$q_1 \equiv$ subtree will be 1

Transitions

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

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

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

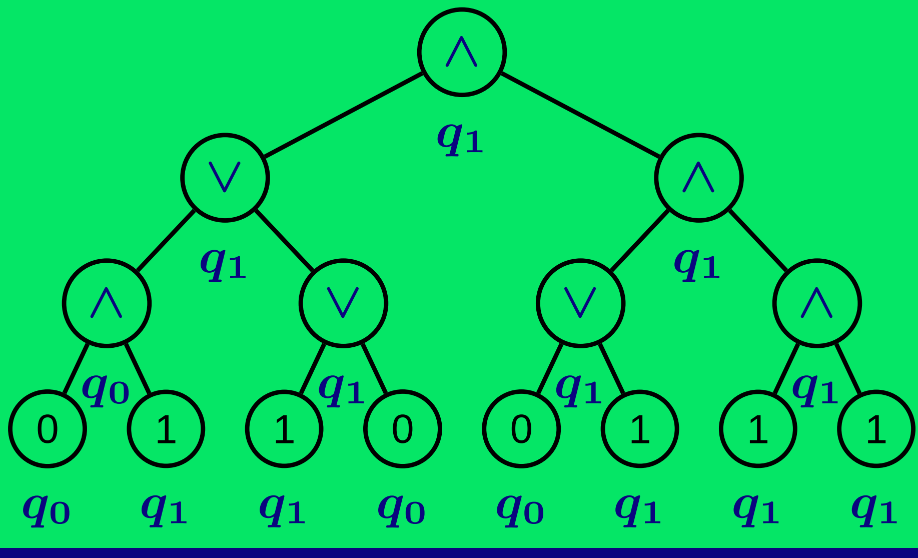
$\delta(\vee, q_0) = \{(q_0, q_0)\}$

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

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

Automata for Ranked Trees

Bottom-up Automaton for (Tree-) Boolean Circuits



Idea

Two states: q_0, q_1

$q_1 \equiv$ subtree evaluates to 1

Transitions

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

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

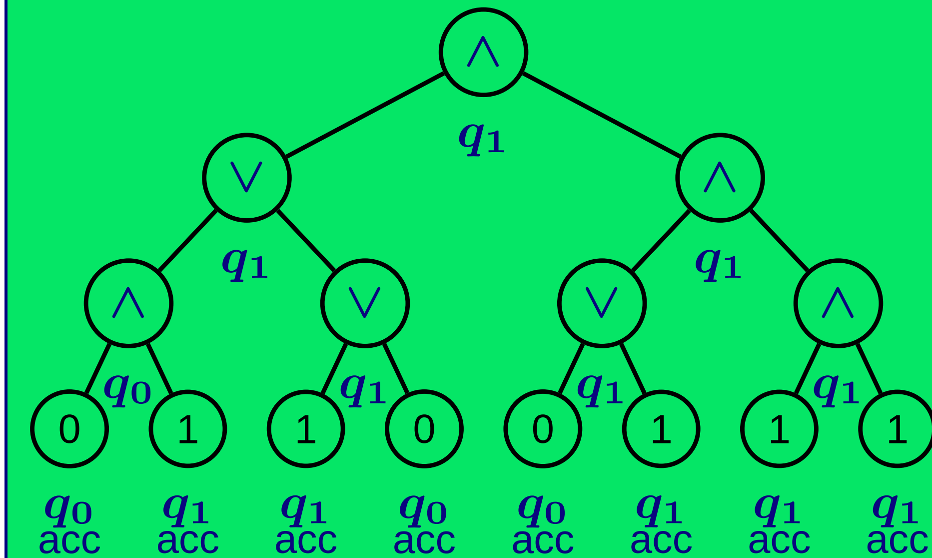
$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

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

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

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

Top-down Automaton for Boolean Circuits



Idea

Three states: q_0, q_1, acc

$q_1 \equiv$ subtree will be 1

Transitions

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

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

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

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

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

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

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

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

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

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

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

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

Definition

Such an L is called **regular**

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

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

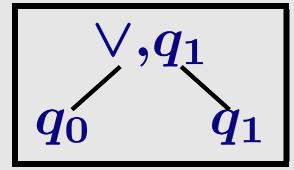
(a) $\Longleftrightarrow$ (c): Just the same thing, viewed in two different ways

Definition

Such an L is called **regular**

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

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

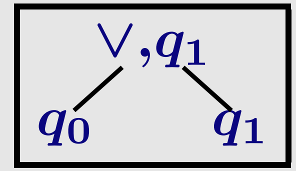
(a) $\Longrightarrow$ (b): Powerset construction

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

Definition

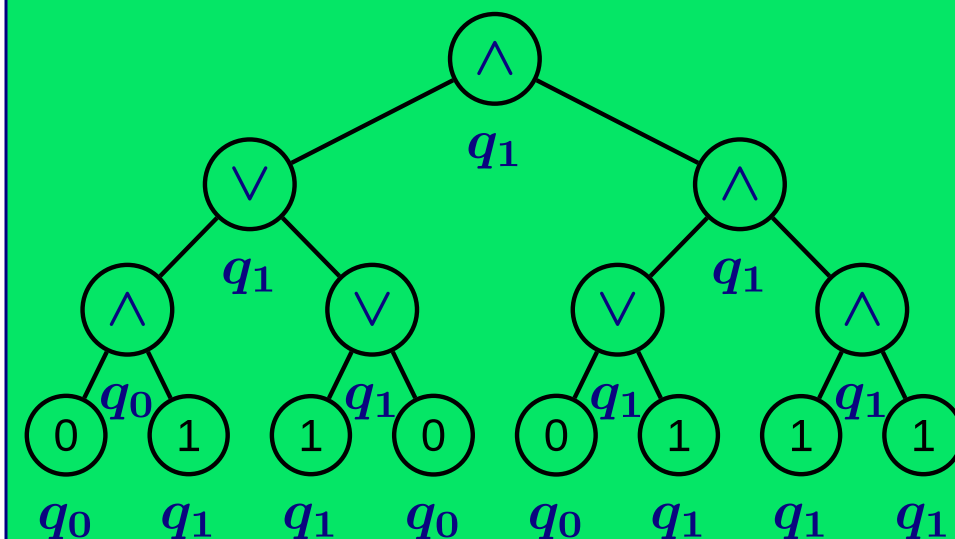
Such an L is called **regular**

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



MSO and Regular Tree Languages

Definition: MSO logic

- MSO-formulas talk about
  - node labels (Q_0,Q_1,$Q_\wedge$,$Q_\vee$)
  - Children and neighbors: $\longrightarrow$, $\downarrow$
  - the root of the tree (root)
- First-order-variables represent nodes
- Monadic second-order (MSO) variables represent sets of nodes

Remark

Exact signature does not matter

Example: Boolean Circuits

$\exists X \;\; X(\text{root}) \wedge \;\; \forall x$
$(Q_0(x) \rightarrow \neg X(x))\wedge$
$((Q_\wedge(x) \wedge X(x)) \rightarrow (\forall y[(x \downarrow y) \rightarrow X(y)]))\wedge$
$((Q_\vee(x) \wedge X(x)) \rightarrow (\exists y[(x \downarrow y) \wedge X(y)]))$

MSO and Regular Tree Languages

Definition: MSO logic

- MSO-formulas talk about
  - node labels (Q_0,Q_1,$Q_\wedge$,$Q_\vee$)
  - Children and neighbors: $\longrightarrow$, $\downarrow$
  - the root of the tree (root)
- First-order-variables represent nodes
- Monadic second-order (MSO) variables represent sets of nodes

Remark

Exact signature does not matter

Example: Boolean Circuits

$$\exists X\ \ X(\text{root}) \wedge\ \ \forall x$$
$$(Q_0(x) \rightarrow \neg X(x))\wedge$$
$$((Q_\wedge(x) \wedge X(x)) \rightarrow (\forall y[(x \downarrow y) \rightarrow X(y)]))\wedge$$
$$((Q_\vee(x) \wedge X(x)) \rightarrow (\exists y[(x \downarrow y) \wedge X(y)]))$$

Theorem (Doner 70; Thatcher, Wright 68)

On ranked trees:

MSO $\equiv$ Regular Tree Languages

Proof idea

Automata $\Rightarrow$ **MSO**:
Formula expresses that there exists a correct tiling

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

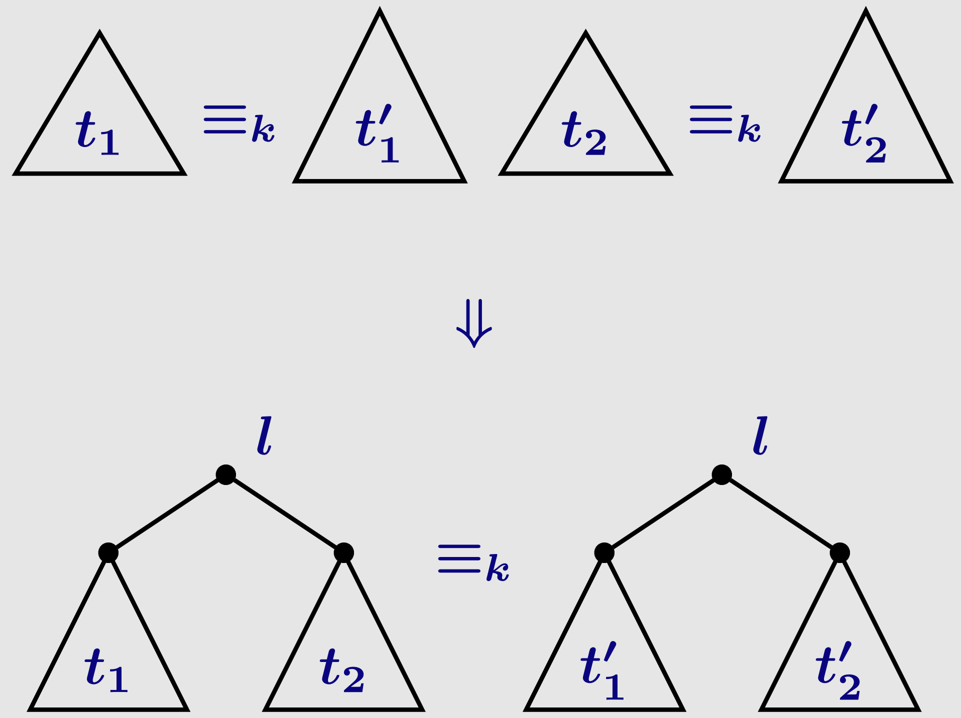
Basic idea:
Automaton computes for each node v the set of formulas which hold in the subtree rooted at v

MSO and Regular Tree Languages (cont.)

MSO $\Longrightarrow$ Automata: more precisely

- 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

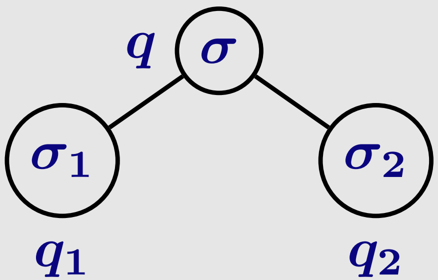


From Ranked to Unranked Trees

Ranked trees

On ranked trees, transitions are described by finite sets:

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

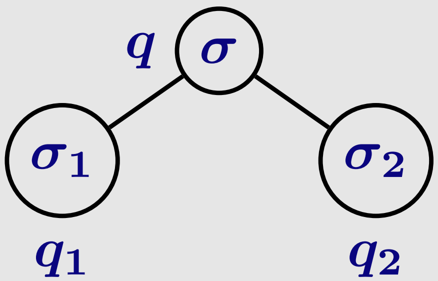


From Ranked to Unranked Trees

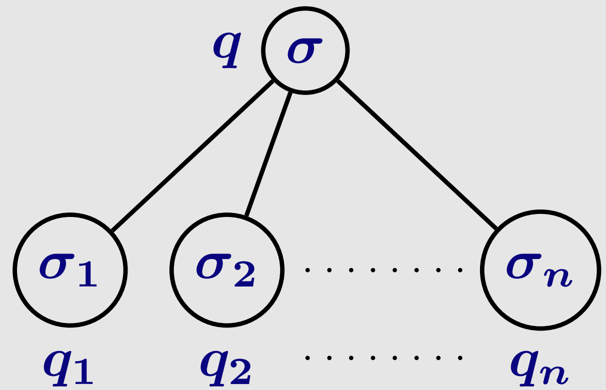
Ranked trees

On ranked trees, transitions are described by finite sets:

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



Unranked trees

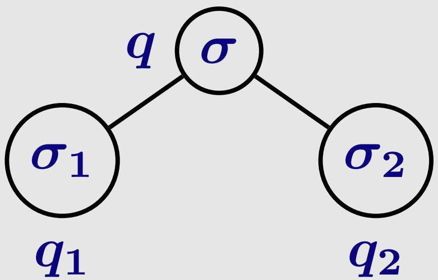


From Ranked to Unranked Trees

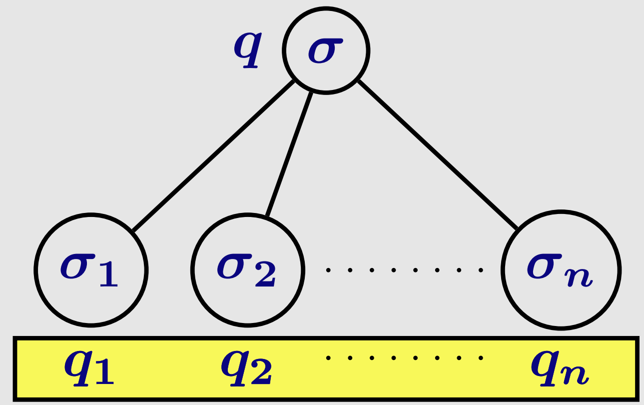
Ranked trees

On ranked trees, transitions are described by finite sets:

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



Unranked trees



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

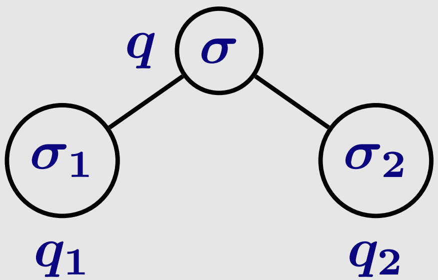
From Ranked to Unranked Trees

Ranked trees	Unranked trees

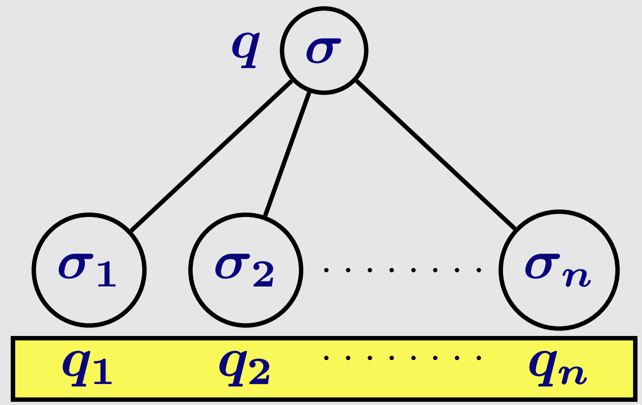
Ranked trees

On ranked trees, transitions are described by finite sets:

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



Unranked trees



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

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

Regular Tree Languages and Complexity

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

Definition

Again: such an L is called **regular**

Complexity issues for MSO on trees

Data Complexity:

Query evaluation is possible in time $O(|t|)$

Combined Complexity:

- Query evaluation is complete for **PSPACE**
- Query evaluation is possible in time $f(|\varphi|)|t|$,

  where f is $\sim \left.2^{2^{\cdot^{\cdot^{\cdot^{2^{2^{|\varphi|}}}}}}}\right\}|\varphi|$
- No elementary f possible unless **P** $=$ **NP** [Frick, Grohe 2002]
- Satisfiability: $f(|\varphi|)$ (same f)
- In practice much better: MONA [Klarlund et al.]

Contents

DTDs and their Weakness

DTDs

- DTDs are essentially generalized context-free grammars

→ [Berstel,Boasson 00] provide characterizations

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)>
]>
```

DTDs and their Weakness

DTDs

- DTDs are essentially generalized context-free grammars

⟶ [Berstel,Boasson 00] provide characterizations

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)>
]>
```

Weakness of DTDs

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

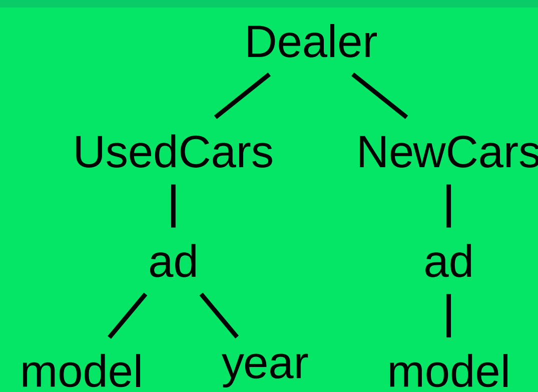
⟶ It would be nice to have types for elements

⟶ Extended DTDs

A classical example

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

Intention



Extended DTDs

Definition [Papakonstantinou, Vianu 2000]

An **extended DTD** (EDTD) 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

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

Note

Extended DTDs are often called *specialized DTDs*

Extended DTDs

Definition [Papakonstantinou, Vianu 2000]

An **extended DTD** (EDTD) 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

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

Note

Extended DTDs are often called *specialized DTDs*

EDTD for Boolean circuits

1-AND	$\longrightarrow$	(1-OR $\mid$ 1-AND $\mid$ 1-leaf)*
1-OR	$\longrightarrow$	$.^*$ (1-OR $\mid$ 1-AND $\mid$ 1-leaf) $.^*$
0-AND	$\longrightarrow$	$.^*$ (0-OR $\mid$ 0-AND $\mid$ 0-leaf) $.^*$
0-OR	$\longrightarrow$	(0-OR $\mid$ 0-AND $\mid$ 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

Extended DTDs and MSO

Observation

- A tree conforms to an extended DTD (d,μ) if there is a labeling of its nodes by types which is valid wrt. d

Extended DTDs and MSO

Observation

- A tree conforms to an extended DTD (d,μ) if there is a labeling of its nodes by types which is valid wrt. d
- This reminds us of something...

Extended DTDs and MSO

Observation

- A tree conforms to an extended DTD (d,μ) if there is a labeling of its nodes by types which is valid wrt. d
- This reminds us of something...

Theorem

Extended DTDs capture exactly the regular tree languages

Extended DTDs and MSO

Observation

- A tree conforms to an extended DTD (d,μ) if there is a labeling of its nodes by types which is valid wrt. d
- This reminds us of something...

Theorem

Extended DTDs capture exactly the regular tree languages

Remarks

- Regular tree languages and MSO-logic are a convenient framework for the study of XML schema languages
- Practical languages as XML Schema usually correspond to subclasses
- Full MSO power: Relax NG
- What about queries?

Contents

Node-selecting Queries

Question
Is there an equally robust class of node-selecting queries?

Node-selecting Queries

Question

Is there an equally robust class of node-selecting queries?

Observations

- 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

Node-selecting Queries

Question

Is there an equally robust class of node-selecting queries?

Observations

- 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 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 $s(q,\lambda(v)) = 1$

Node-selecting Queries

Question

Is there an equally robust class of node-selecting queries?

Observations

- 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 node-selecting automata
- Nondeterministic bottom-up automata plus select function:

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

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

Existential vs. universal semantics

- 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

Theorem

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

Node-selecting queries (cont.)

Theorem

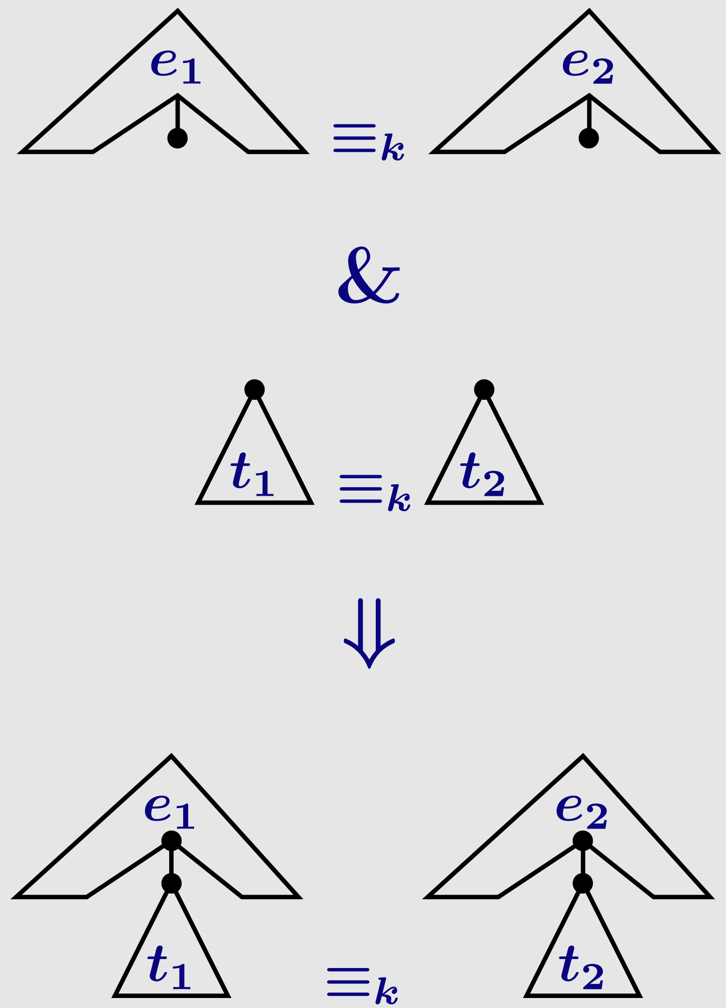
A node selecting query is MSO-definable iff it is expressible by a nondeterministic bottom-up node selecting automaton

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



Node-selecting queries (cont.)

More query models

- Same combined complexity as for language recognition (**PSPACE**-complete)

$\longrightarrow$ 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]

Data complexity

- MSO node-selecting queries can be evaluated in two passes

  First pass, bottom-up: Compute the types of the subtrees

  Second pass, top-down: Compute the types of the envelopes and combines it with the subtree information

$\longrightarrow$ 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: Data complexity is linear time

Contents

XML Transformations and MSO-Logic

Definition: XSLT TypeChecking

Given: DTDs d_1 and d_2

Transformation T

Result: Is $T(t) \models d_2$ for each document t with $t \models d_1$?

Questions

- Is XSLT TypeChecking decidable?
- What is the complexity?

XML Transformations and MSO-Logic

Definition: XSLT TypeChecking

Given: DTDs d_1 and d_2

Transformation T

Result: Is $T(t) \models d_2$ for each document t with $t \models d_1$?

Questions

- Is XSLT TypeChecking decidable?
- What is the complexity?

Outline

- 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
- [Milo,Suciu,Vianu 01]

XSLT in more detail

How XSLT roughly works

Templates: ⟨`xsl:template name=TName match=pattern mode=MName`⟩

Template application:
⟨`xsl:apply-templates select=Expression mode=MName`⟩

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 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

```
⟨xsl:template match=”a”⟩
  ⟨a⟩ ⟨xsl:apply-templates⟩ ⟨/a⟩
⟨/xsl:template⟩
⟨xsl:template match=”a” mode=”below”⟩
  ⟨d⟩ ⟨xsl:apply-templates⟩ ⟨/d⟩
⟨/xsl:template⟩
⟨xsl:template match=”b”⟩
  ⟨b⟩ ⟨xsl:apply-templates mode=”below”⟩ ⟨/b⟩
⟨/xsl:template⟩
⟨xsl:template match=”b” mode=”below”⟩
  ⟨b⟩ ⟨xsl:apply-templates mode=”below”⟩ ⟨/b⟩
⟨/xsl:template⟩
⟨xsl:template match=”c”⟩
  ⟨c⟩ ⟨/c⟩
⟨/xsl:template⟩
⟨xsl:template match=”c” mode=”below”⟩
  ⟨c⟩ ⟨/c⟩
⟨/xsl:template⟩
```

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⟩ ⟨/...⟩

XSLT: Example

Example Transformation

Remove everything below a c. Translate a below b into d

Example XSLT (Abbreviated)

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

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

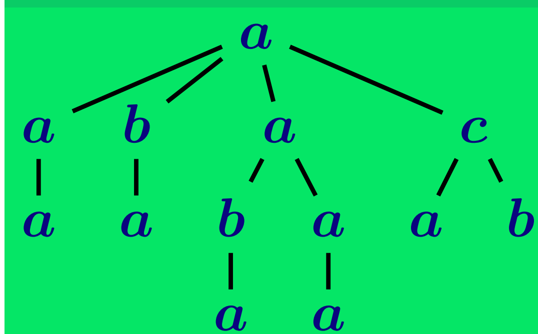
⟨... match=”b”⟩ ⟨b⟩ ⟨xsl:apply-templates mode=”below”⟩ ⟨/b⟩ ⟨/...⟩

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

⟨... match=”c”⟩ ⟨c⟩ ⟨/c⟩ ⟨/...⟩

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

Example Trees



XSLT: Example

Example Transformation

Remove everything below a c*. Translate* a *below* b *into* d

Example XSLT (Abbreviated)

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

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

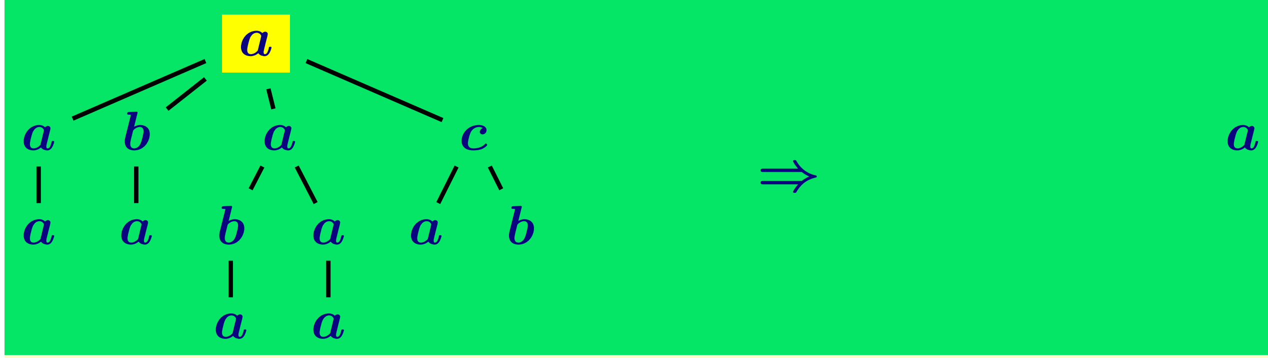
⟨... match="b"⟩ ⟨b⟩ ⟨xsl:apply-templates mode="below"⟩ ⟨/b⟩ ⟨/...⟩

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

⟨... match="c"⟩ ⟨c⟩ ⟨/c⟩ ⟨/...⟩

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

Example Trees



XSLT: Example

Example Transformation

Remove everything below a c. *Translate* a *below* b *into* d

Example XSLT (Abbreviated)

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

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

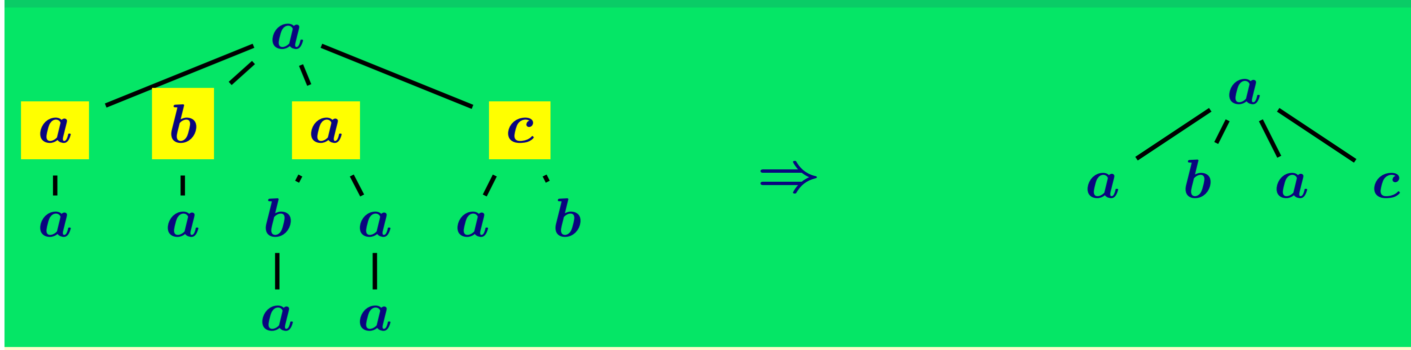
⟨... match=”b”⟩ ⟨b⟩ ⟨xsl:apply-templates mode=”below”⟩ ⟨/b⟩ ⟨/...⟩

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

⟨... match=”c”⟩ ⟨c⟩ ⟨/c⟩ ⟨/...⟩

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

Example Trees



XSLT: 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⟩ ⟨/...⟩

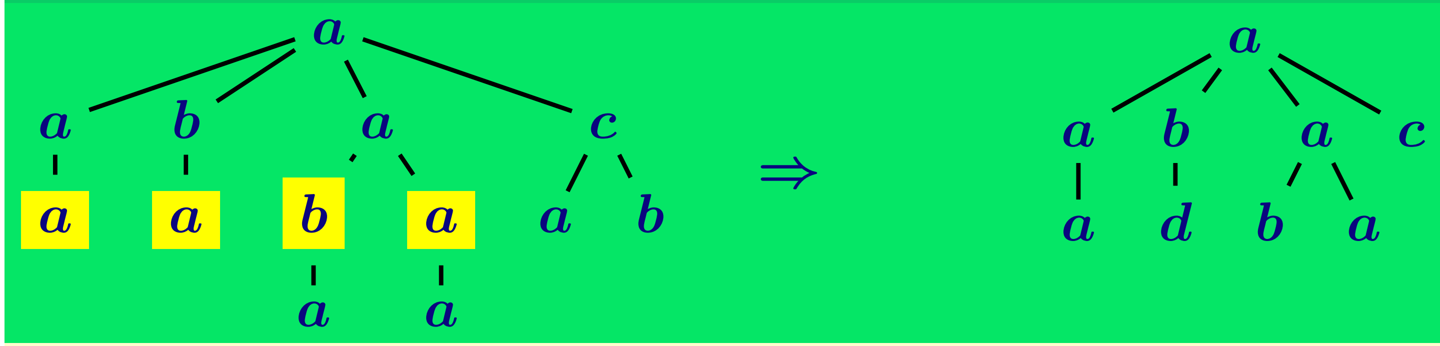
⟨... match="b"⟩ ⟨b⟩ ⟨xsl:apply-templates mode="below"⟩ ⟨/b⟩ ⟨/...⟩

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

⟨... match="c"⟩ ⟨c⟩ ⟨/c⟩ ⟨/...⟩

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

Example Trees



XSLT: 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⟩ ⟨/...⟩

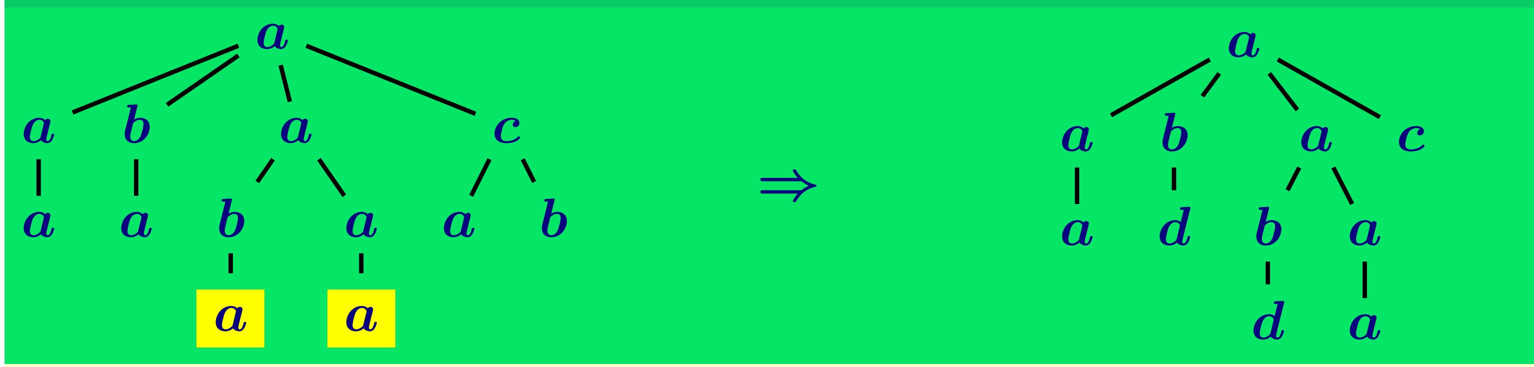
⟨... match="b"⟩ ⟨b⟩ ⟨xsl:apply-templates mode="below"⟩ ⟨/b⟩ ⟨/...⟩

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

⟨... match="c"⟩ ⟨c⟩ ⟨/c⟩ ⟨/...⟩

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

Example Trees



An automaton model for XSLT

Definition: k-pebble Transducer

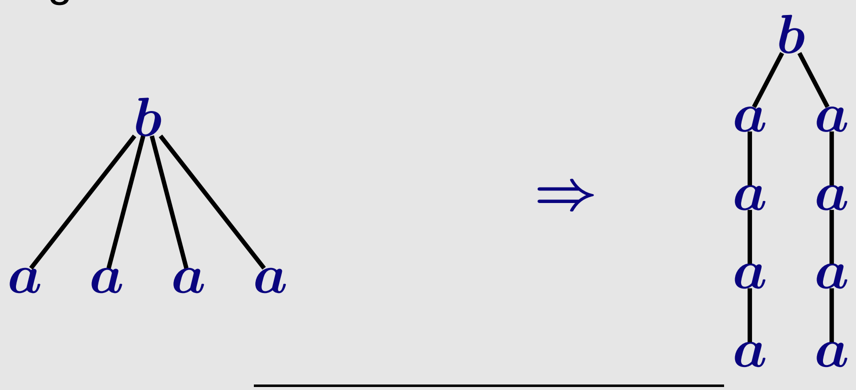
- Work on binary tree encodings of unranked trees
- Up to k pebbles can be placed on the tree
- Only pebble with highest number (*current pebble*) can move, depending on state, number of pebbles, symbols under pebbles and incidence of pebbles
- Possible pebble movements:
  - stay
  - go to left child, right child or parent
  - lift current pebble
  - place new pebble on the root
- Nondeterminism allowed

  (Proof presented here: deterministic case)
- 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

Back to the Typechecking Question

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:

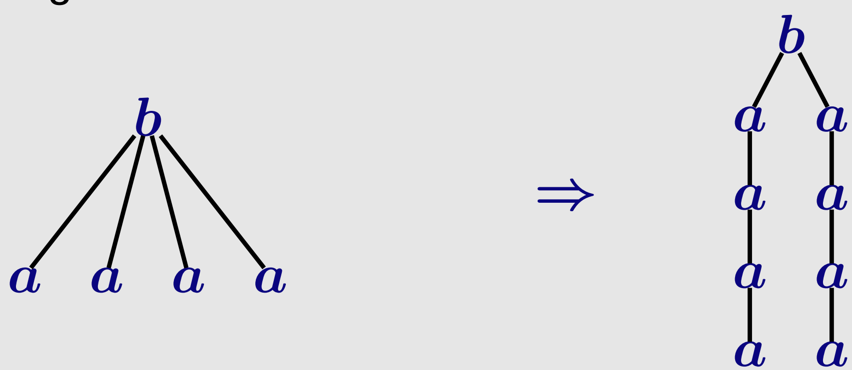


- Better approach:
  - Compute $T^{-1}(\overline{L(d_2)})$
  - Check $L(d_1) \cap T^{-1}(\overline{L(d_2)}) = \emptyset$

Back to the Typechecking Question

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:



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

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 computed by k-pebble transducer

$$\Rightarrow T = T_1 \circ \cdots \circ T_{k+1}$$

with 0-pebble transducers T_i

[Engelfriet, Maneth 03]

(ii) L regular, T_i 0-pebble transducer

$$\Rightarrow$$

$T_i^{-1}(L)$ accepted by 0-pebble acceptor

(iii) 0-pebble acceptors only accept regular tree languages

Step (ii)

Lemma

L regular, T_i 0-pebble transducer
$\Rightarrow T_i^{-1}(L)$ accepted by 0-pebble acceptor

Proof

- Let B be a nondeterministic top-down tree automaton which accepts $\overline{L}$
- We construct a 0-pebble acceptor A for $T_i^{-1}(\overline{L})$, i.e., an automaton which on input t decides whether $T(t)$ 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

Step (iii)

Lemma

0-pebble acceptors only accept regular tree languages

Proof idea

Show that the language of 0-pebble acceptors can be expressed by an MSO-formula:

1. Reduce 0-pebble automaton acceptance to AGAP (Alternating Graph Accessibility)
2. Show that AGAP can be expressed in MSO

Step (iii)

Lemma

0-pebble acceptors only accept regular tree languages

Proof idea

Show that the language of 0-pebble acceptors can be expressed by an MSO-formula:

1. Reduce 0-pebble automaton acceptance to AGAP (Alternating Graph Accessibility)
2. Show that AGAP can be expressed in MSO

Accessible Nodes

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

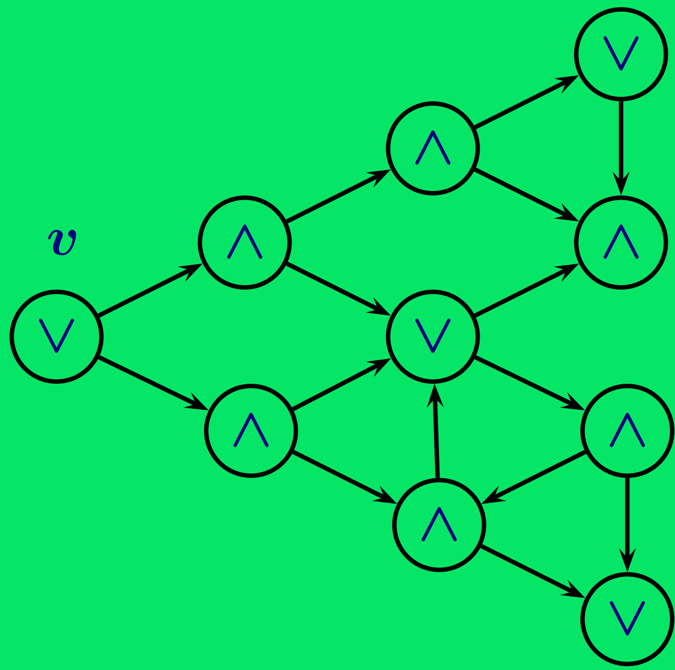
- $w \in V_\wedge$ and all successors of w are accessible, or
- $w \in V_\vee$ and at least one successor of w is accessible

Definition: AGAP

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

Question: Is v accessible?

Example



Step (iii)

Lemma

0-pebble acceptors only accept regular tree languages

Proof idea

Show that the language of 0-pebble acceptors can be expressed by an MSO-formula:

1. Reduce 0-pebble automaton acceptance to AGAP (Alternating Graph Accessibility)
2. Show that AGAP can be expressed in MSO

Accessible Nodes

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

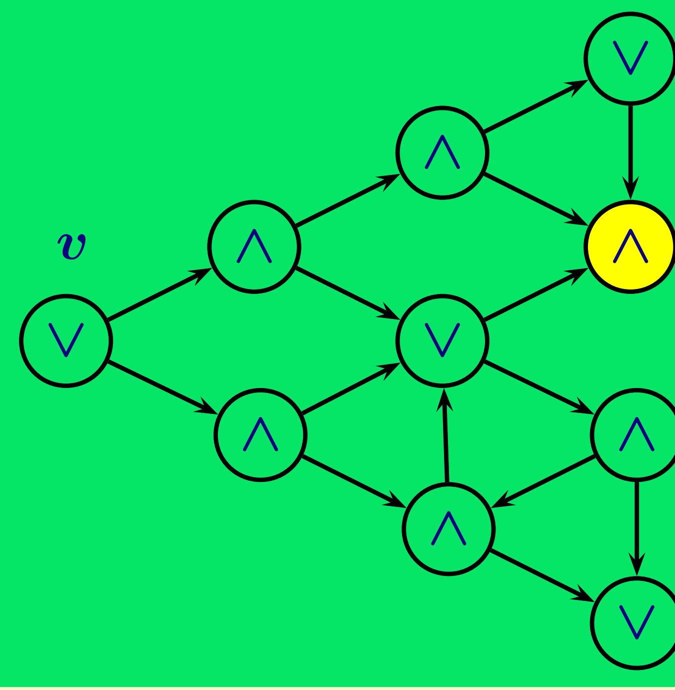
- $w \in V_\wedge$ and all successors of w are accessible, or
- $w \in V_\vee$ and at least one successor of w is accessible

Definition: AGAP

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

Question: Is v accessible?

Example



Step (iii)

Lemma

0-pebble acceptors only accept regular tree languages

Proof idea

Show that the language of 0-pebble acceptors can be expressed by an MSO-formula:

1. Reduce 0-pebble automaton acceptance to AGAP (Alternating Graph Accessibility)
2. Show that AGAP can be expressed in MSO

Accessible Nodes

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

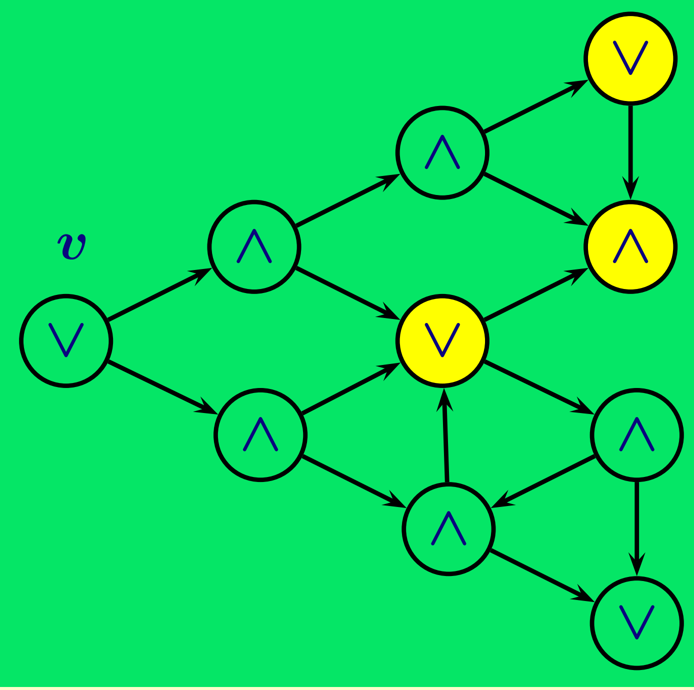
- $w \in V_\wedge$ and all successors of w are accessible, or
- $w \in V_\vee$ and at least one successor of w is accessible

Definition: AGAP

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

Question: Is v accessible?

Example



Step (iii)

Lemma

$\mathbf{0}$-pebble acceptors only accept regular tree languages

Proof idea

Show that the language of $\mathbf{0}$-pebble acceptors can be expressed by an MSO-formula:

1. Reduce $\mathbf{0}$-pebble automaton acceptance to AGAP (Alternating Graph Accessibility)
2. Show that AGAP can be expressed in MSO

Accessible Nodes

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

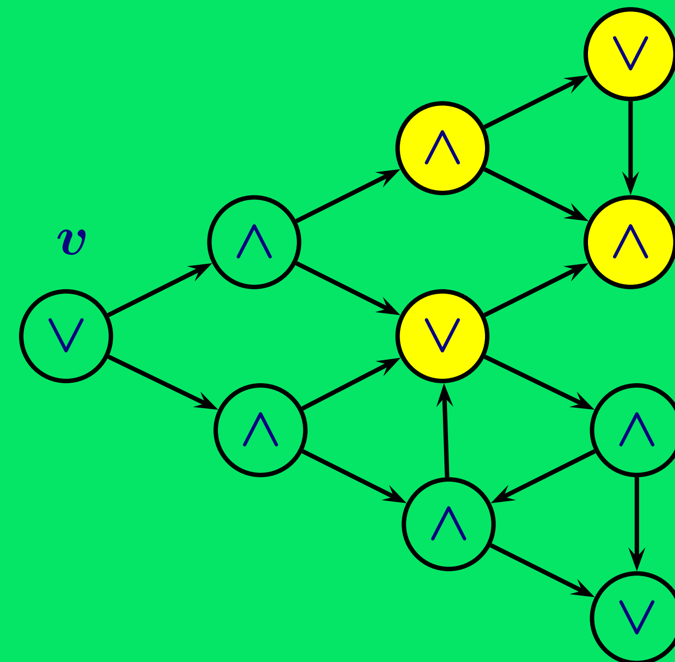
- $w \in V_\wedge$ and all successors of w are accessible, or
- $w \in V_\vee$ and at least one successor of w is accessible

Definition: AGAP

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

Question: Is v accessible?

Example



Step (iii)

Lemma

0-pebble acceptors only accept regular tree languages

Proof idea

Show that the language of 0-pebble acceptors can be expressed by an MSO-formula:

1. Reduce 0-pebble automaton acceptance to AGAP (Alternating Graph Accessibility)
2. Show that AGAP can be expressed in MSO

Accessible Nodes

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

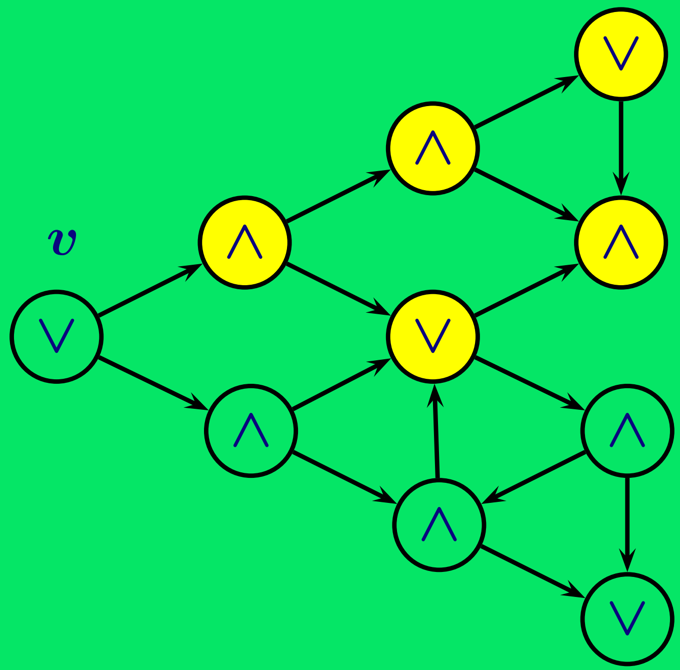
- $w \in V_\wedge$ and all successors of w are accessible, or
- $w \in V_\vee$ and at least one successor of w is accessible

Definition: AGAP

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

Question: Is v accessible?

Example



Step (iii)

Lemma

0-pebble acceptors only accept regular tree languages

Proof idea

Show that the language of 0-pebble acceptors can be expressed by an MSO-formula:

1. Reduce 0-pebble automaton acceptance to AGAP (Alternating Graph Accessibility)
2. Show that AGAP can be expressed in MSO

Accessible Nodes

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

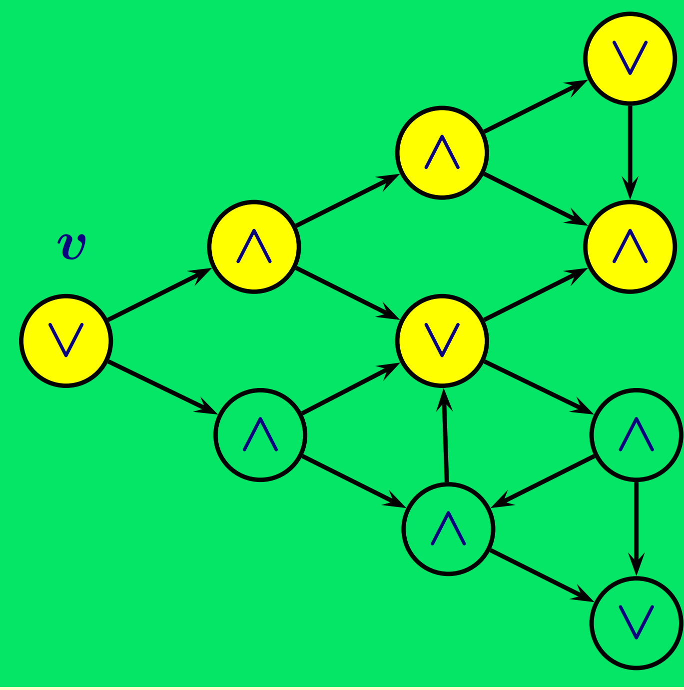
- $w \in V_\wedge$ and all successors of w are accessible, or
- $w \in V_\vee$ and at least one successor of w is accessible

Definition: AGAP

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

Question: Is v accessible?

Example



0-Pebble Acceptors, AGAP and MSO

Construction of $G_{A,t}$

- Nodes in $V_\vee$ are the **configurations** of A on t:
  pairs $[q,v]$, state q, node v of t
- Nodes in $V_\wedge$ are ϵ and certain pairs (γ_1,γ_2) of configurations
- 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

Facts

- A accepts $t \iff$ $[q,\text{root}]$ is accessible in $G_{A,t}$ with $q \in F$
- $|G_{A,t}| = O(|t|)$

0-Pebble Acceptors, AGAP and MSO

Construction of $G_{A,t}$

- Nodes in $V_\vee$ are the **configurations** of A on t: pairs $[q,v]$, state q, node v of t
- Nodes in $V_\wedge$ are ϵ and certain pairs (γ_1,γ_2) of configurations
- 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

Facts

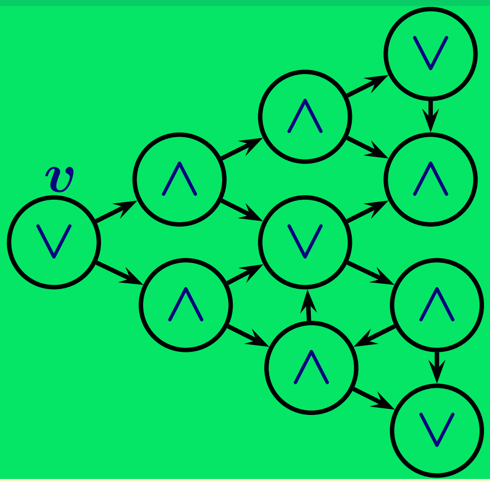
- A accepts $t \iff [q,\text{root}]$ is accessible in $G_{A,t}$ with $q \in F$
- $|G_{A,t}| = O(|t|)$

Definition: Reverse-closed Sets of Nodes

A set S of nodes is **reverse-closed** $:\iff$

- 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

Reverse-closed as MSO-Formula

$\forall S\ (\text{rc}(S) \longrightarrow S(v))$, where $\text{rc}(S)$ is

$$\forall x\,([V_\wedge(x) \wedge \forall y\,(E(x,y) \longrightarrow S(y))] \longrightarrow S(x)) \wedge$$
$$([V_\vee(x) \wedge \exists y\,(E(x,y) \wedge S(y))] \longrightarrow S(x))$$

XSLT Typechecking: Summary

Summary of proof

- 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 $T(t) \notin L(d_2)$
  (4) Construct a nondeterministic bottom-up automaton A' equivalent to $\neg\Phi$
  (5) Check that $L(d_1) \subseteq L(A')$
- Complexity: non-elementary

XSLT Typechecking: Summary

Summary of proof

- 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 $T(t) \notin L(d_2)$
  (4) Construct a nondeterministic bottom-up automaton A' equivalent to $\neg\Phi$
  (5) Check that $L(d_1) \subseteq L(A')$
- Complexity: non-elementary

Related work

- TYPECHECKING is decidable for compositions of macro tree transducers [Engelfriet, Maneth 03]
- 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. 01]
- 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, Neven 03]

Contents

Navigation and Temporal Logics

Remarks

- MSO-logic offers a framework for dealing with schemas
- Close connection to automata
- Helpful in proofs
- Extends to (node-selecting) queries
- But: for many querying tasks MSO-power is not needed
- `XPath`: Navigation along paths

$\longrightarrow$ Similarity to temporal logics

- Presentation follows [Libkin 05]

Navigation and Temporal Logics (cont.)

Definition: An LTL-like logic: TL^{tree}

- $\boldsymbol{A}$, $\varphi \vee \psi$, $\neg\varphi$
- $\boldsymbol{X}_*\varphi$, $\boldsymbol{X}_*^-\varphi$, $\varphi\boldsymbol{U}_*\psi$, $\varphi\boldsymbol{U}_*^-\psi$

($* = \longrightarrow$ or $\downarrow$)

Theorem [Marx 04]

A unary or binary query over unordered trees is FO-definable (with $\downarrow^*$, $\longrightarrow^*$) iff it is definable in TL^{tree}

Proof idea

Proof similar as equivalence of LTL and FO on orders [Kamp 68]

Navigation and Temporal Logics (cont.)

Definition: An LTL-like logic: $\mathrm{TL}^{\mathrm{tree}}$

- A, $\varphi \vee \psi$, $\neg\varphi$
- $X_*\varphi$, $X_*^-\varphi$, $\varphi U_* \psi$, $\varphi U_*^- \psi$

($* = \longrightarrow$ or $\downarrow$)

Theorem [Marx 04]

A unary or binary query over unordered trees is FO-definable (with $\downarrow^*$, $\longrightarrow^*$) iff it is definable in $\mathrm{TL}^{\mathrm{tree}}$

Proof idea

Proof similar as equivalence of LTL and FO on orders [Kamp 68]

Definition: A CTL*-like logic: $\mathrm{CTL}^*_{\mathrm{past}}$

- **Node formulas:**
  - A, $\alpha \vee \alpha'$, $\neg\alpha$
  - $E\beta_*$
- **Path formulas:**
  - α, $\neg\beta_*$, $\beta \vee \beta'$
  - $X_*\beta$, $X_*^-\beta$, $\beta U_* \beta'$, $\beta U_*^- \beta'$

($* = \longrightarrow$ or $\downarrow$)

Theorem [Barcelo, Libkin 05]

A unary or binary query over unordered trees is FO-definable (with $\downarrow^*$, $\longrightarrow^*$) iff it is definable in $\mathrm{CTL}^*_{\mathrm{past}}$

Proof idea

Proof similar as equivalence of CTL* and FO on binary trees [Hafer, Thomas 87]

A Related Result

Theorem [Marx 04]

Containment of Navigational `XPath` queries in the presence of DTDs is is in **EXPTIME**

Proof idea

- Navigational `XPath` can be translated into propositional dynamic logic (PDL) (over structures with $\downarrow$, $\longrightarrow$)
- DTDs can also be expressed by PDL-formulas
- Implication for PDL is **EXPTIME**-complete

Remark

For much weaker fragments, containment is already **EXPTIME**-hard

Contents

XPath and Fragments of First-order Logic

Characterizations of XPath

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

[Benedikt, Fan, Kuper 03]

Proof idea

- Basic idea:
  For each node u 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

XPath and Fragments of First-order Logic

Characterizations of XPath

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

[Benedikt, Fan, Kuper 03]

Proof idea

- Basic idea:
  For each node u 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

Theorem [Marx, de Rijke 05]

Node-selecting navigational XPath queries correspond to unary two-variable first-order formulas over $\downarrow, \downarrow^*, \longrightarrow, \longrightarrow^*$

Proof idea

Proof follows the same lines as characterization of unary temporal logic by two-variable logic

[Etessami, Vardi, Wilke]

XPath and Full First-Order Logic

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

XPath and Full First-Order Logic

Question

What is needed to capture full first-order logic?

Conditional XPath

- Conditional axes: Expressions of the kind P^+, where P is a location step
- Conditional XPath: Navigational XPath plus conditional axes

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 Full First-Order Logic

Question

What is needed to capture full first-order logic?

Conditional XPath

- **Conditional axes**: Expressions of the kind P^+, where P is a location step
- **Conditional XPath**: Navigational XPath plus conditional axes

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

Theorem [Marx 04,05]

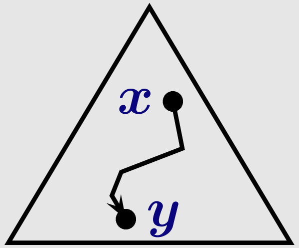
Conditional XPath corresponds exactly to first-order logic over $\downarrow^*$, $\longrightarrow^*$ (wrt node-selecting and binary queries)

First-Order Logic Plus Regular Expressions

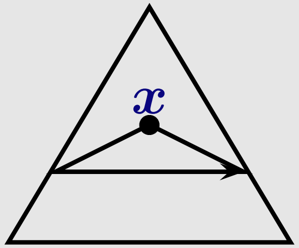
FOREG

First-Order Logic

+ vertical regular expressions (over paths)

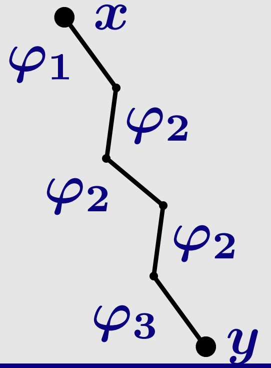


+ horizontal regular expressions (over children)



+ Nesting of formulas and regular expressions

$$[\varphi_1(s,t) \cdot \varphi_2(s,t)^* \cdot \varphi_3(s,t)]^{\downarrow}_{s,t}(x,y)$$



Fact

FO $\subsetneq$ FOREG $\subsetneq$ MSO

First-Order Logic and Automata

- MSO-logic $\equiv$ Parallel tree automata
- FO-logic $\equiv$???

First-Order Logic and Automata

- MSO-logic $\equiv$ Parallel tree automata
- FO-logic $\equiv$???

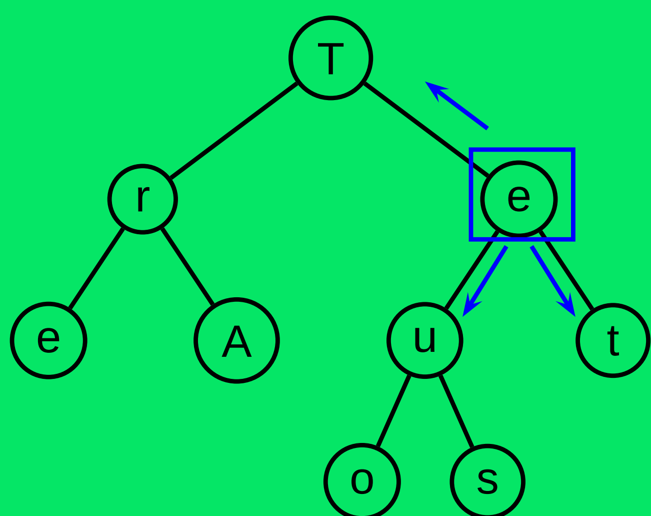
Definition: Tree-walk automaton

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

Illustration



First-Order Logic and Automata

- MSO-logic ≡ Parallel tree automata
- FO-logic ≡ ???

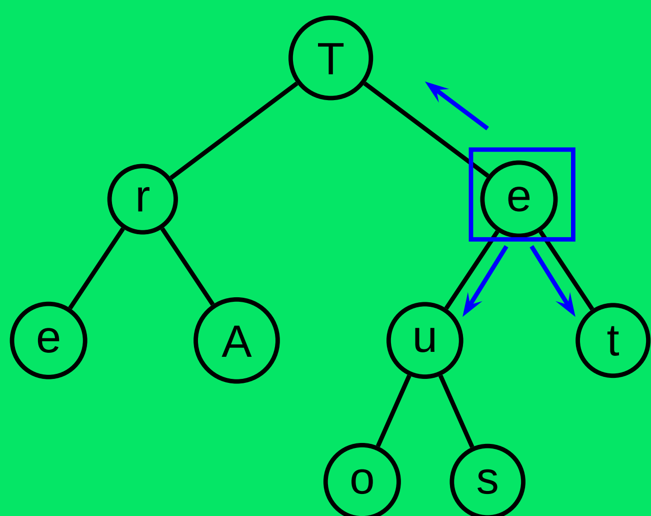
Definition: Tree-walk automaton

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

Illustration



Fact [Neven, Sch. 00]

FO-sentences over binary trees with $\downarrow$ and $\longrightarrow$ can be evaluated by deterministic Tree-Walk automata

Proof idea

- Simple application of Gaifman's Theorem
- Does not hold for unranked trees

First-Order Logic and Automata

- MSO-logic $\equiv$ Parallel tree automata
- FO-logic $\equiv$???

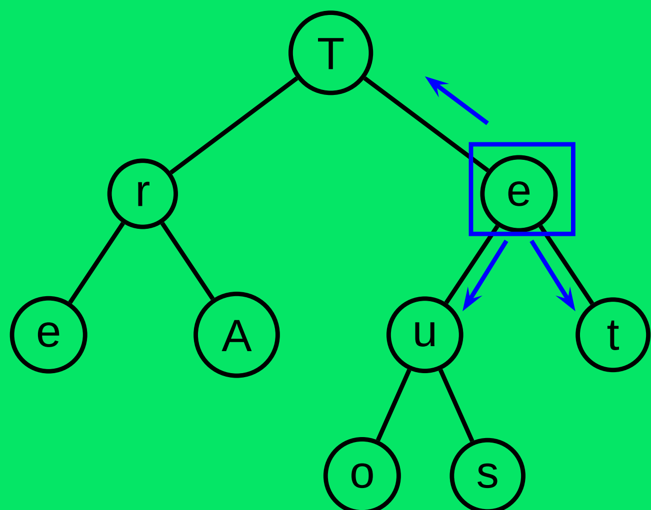
Definition: Tree-walk automaton

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

Illustration



Fact [Neven, Sch. 00]

FO-sentences over binary trees with $\downarrow$ and $\longrightarrow$ can be evaluated by deterministic Tree-Walk automata

Proof idea

- Simple application of Gaifman's Theorem
- Does not hold for unranked trees

Remark

To capture the expressive power of Tree-walk automata one needs a bit more...

Theorem [Neven, Sch. 00]

- Nondet. TWA $\equiv TC^1[FO[\downarrow, \text{mod}]]$
- Det. TWA $\equiv DTC^1[FO[\downarrow, \text{mod}]]$

over binary trees with $\downarrow$ and $\longrightarrow$

Contents

MSO-logic vs. Unary Transitive Closure Logic

Remarks

- On strings: MSO $\equiv$ FO $+$ unary TC
- On (binary) trees???

MSO-logic vs. Unary Transitive Closure Logic

Remarks

- On strings: MSO $\equiv$ FO $+$ unary TC
- On (binary) trees???

Theorem [Engelfriet, Hoogeboom 05]

Nondeterministic pebble automata correspond to FO plus positive unary TC on binary trees

Corollary

Positive, unary TC-logic is weaker than MSO on binary trees iff pebble automata are weaker than parallel automata

MSO-logic vs. Unary Transitive Closure Logic

Remarks

- On strings: MSO $\equiv$ FO $+$ unary TC
- On (binary) trees???

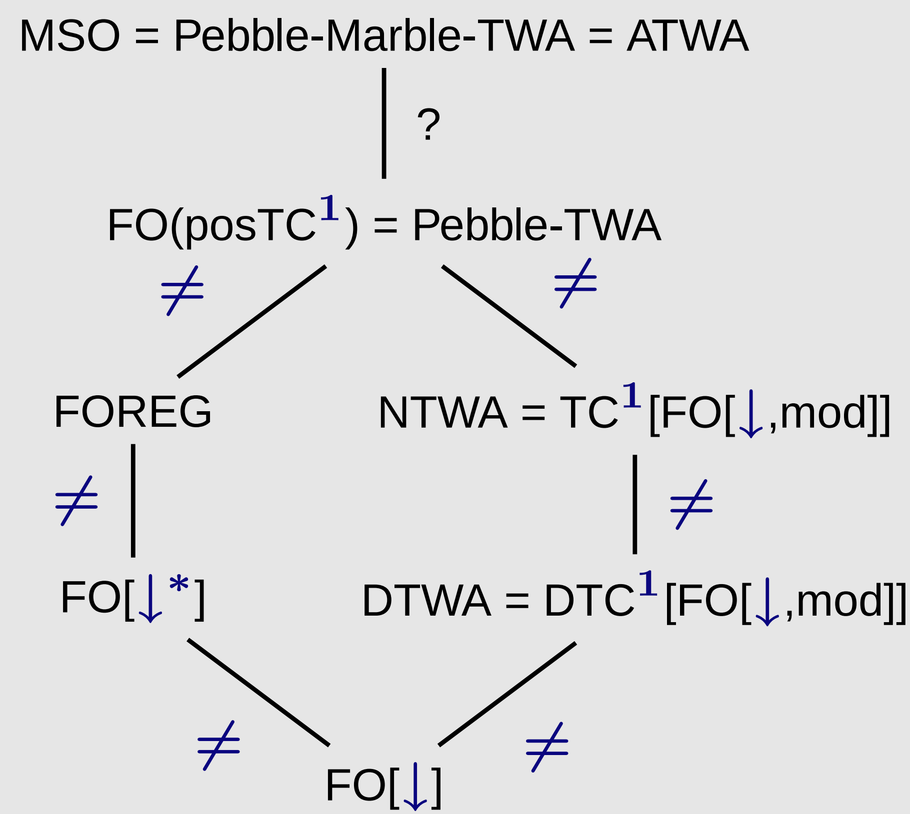
Theorem [Engelfriet, Hoogeboom 05]

Nondeterministic pebble automata correspond to FO plus positive unary TC on binary trees

Corollary

Positive, unary TC-logic is weaker than MSO on binary trees iff pebble automata are weaker than parallel automata

Inclusion structure



Contents

Extensions

So far...

- We have seen logics for
  - Validation, Typing
  - Navigation
  - Transformation
- What about more general queries?
  - results of higher arity?
  - joins, i.e., comparisons of data values
  - counting

Extensions

So far...

- We have seen logics for
  - Validation, Typing
  - Navigation
  - Transformation
- What about more general queries?
  - results of higher arity?
  - joins, i.e., comparisons of data values
  - counting

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:
  - $\equiv$ EMSO 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

Data Values

Remark

- In (database) queries or constraints comparisons of data values are very common
- Most XML theory concentrates on structure of trees instead

Example queries/constraints

- Did we reserve a room for every participant?

-
- Does every participant give at most one talk?
-

Data Values

Remark

- In (database) queries or constraints comparisons of data values are very common
- Most XML theory concentrates on structure of trees instead

Example queries/constraints

- Did we reserve a room for every participant?
- $\forall x$ Partic.Name$(x) \longrightarrow$ $\exists y$Room.Name$(y) \wedge x$.data $= y$.data

---

- Does every participant give at most one talk?
-

Data Values

Remark

- In (database) queries or constraints comparisons of data values are very common
- Most XML theory concentrates on structure of trees instead

Example queries/constraints

- Did we reserve a room for every participant?
- $\forall x$ Partic.Name$(x) \longrightarrow \exists y$Room.Name$(y) \wedge x.$data $= y.$data

---

- Does every participant give at most one talk?
- $\forall x \forall y\ [x \neq y \wedge$ Talk.Speaker$(x) \wedge$ Talk.Speaker$(y)] \longrightarrow x.$data $\neq y.$data

Data Values

Remark

- In (database) queries or constraints comparisons of data values are very common
- Most XML theory concentrates on structure of trees instead

Example queries/constraints

- Did we reserve a room for every participant?
- $\forall x$ Partic.Name$(x) \longrightarrow$ $\exists y$Room.Name$(y) \wedge x$.data $= y$.data

---

- Does every participant give at most one talk?
- $\forall x \forall y\ [x \neq y \wedge$ Talk.Speaker$(x) \wedge$ Talk.Speaker$(y)] \longrightarrow x$.data $\neq y$.data

The setting

- We concentrate on data strings
- Access to data values only via equality tests

Example: data string

2	**3**	**3**	**3**	**2**	**2**	**7**	**17**	**17**	**3**	**4**	**5**	**2**	**3**	**3**	**4**	**4**
c	b	c	a	a	b	b	b	c	a	b	a	c	b	b	a	a

Definition

- **Data string**: Finite sequence over $\Sigma \times \mathcal{D}$, where
  - Σ finite
  - $\mathcal{D}$ infinite
- Logical language:
  - $\boxed{a(x)}$: Letter position x is $a \in \Sigma$
  - order relation $<$, successor relation $+1$
  - $\boxed{\sim}$: $x \sim y$ if positions x and y have the same $\mathcal{D}$-value

$\longrightarrow$ Equivalence relation

Some Known Results about FO^2

Over arbitray relational structures	Over strings
• Finite model property [Mortimer 75] • Satisfiability **NEXPTIME**-complete [Grädel et al. 97] • On structures with 1 or 2 equivalence relations: decidable [Kieroński, Otto 05] • On structures with 3 equivalence relations: undecidable [Kieroński, Otto 05] • On structures with a linear order: in **coNEXPTIME** [Otto] • On structures with several well-orderings: undecidable [Otto]	• Satisfiability **NEXPTIME**-complete • Expressive power: – unary LTL and $\Sigma^2 \cap \Pi^2$ [Etessami, Vardi, Wilke 97] – variety DA [Thérien, Wilke 98] – two way, partially-ordered DFA [Sch., Thérien, Vollmer 01]

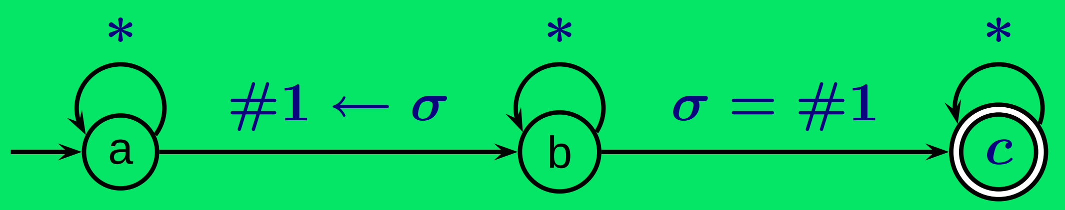
Automata on Data Strings

Models

- Register-Automata: finitely many registers for data values
- Pebble-Automata: finitely many pebbles with stack discipline
- Variations:

  1-way or 2-way, Deterministic or Nondeterministic

Example



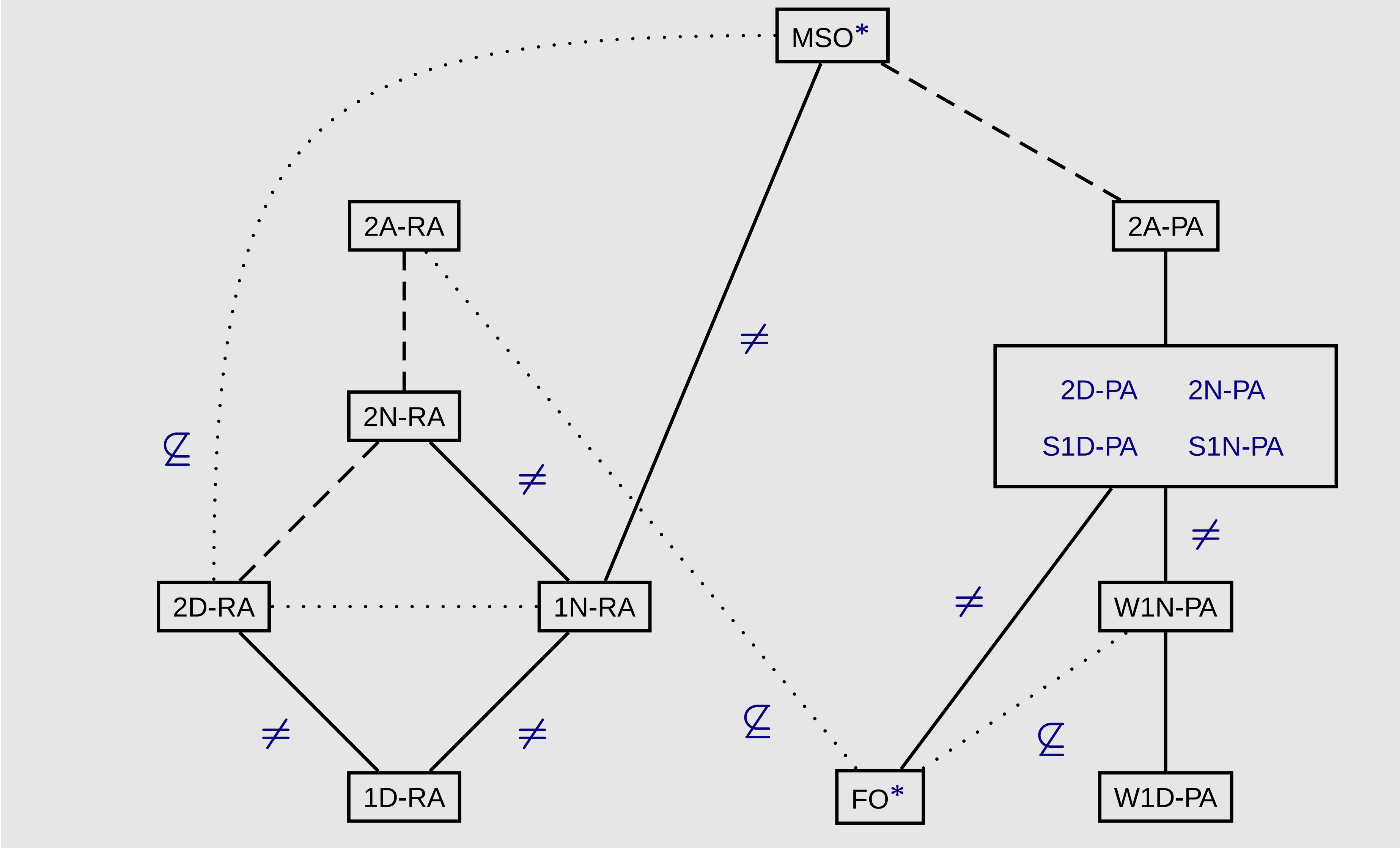
Facts

- Non-emptiness undecidable for most models
- Exception: 1N-register automata
- This holds also for **FO** logic

⟶ [Kaminski, Francez 94; Neven, Sch., Vianu 01]

Overview of Automata Models

Inclusion Structure



Decidability Results

Theorem

Satisfiability of $\mathbf{FO}^2(+1, <)$ on data strings is decidable

Further results

- Emptiness of multicounter automata can be reduced to satisfiability of $\mathbf{FO}^2(+1, <)$ on data strings
- $\mathbf{FO}^2(<)$ on data strings is complete for **NEXPTIME**
- $\mathbf{FO}^2(+1)$ on data strings is hard for **NEXPTIME** (and in 2**NEXPTIME**)
- $\mathbf{FO}^3(+1)$ on data strings is undecidable

Decidability Results

Theorem

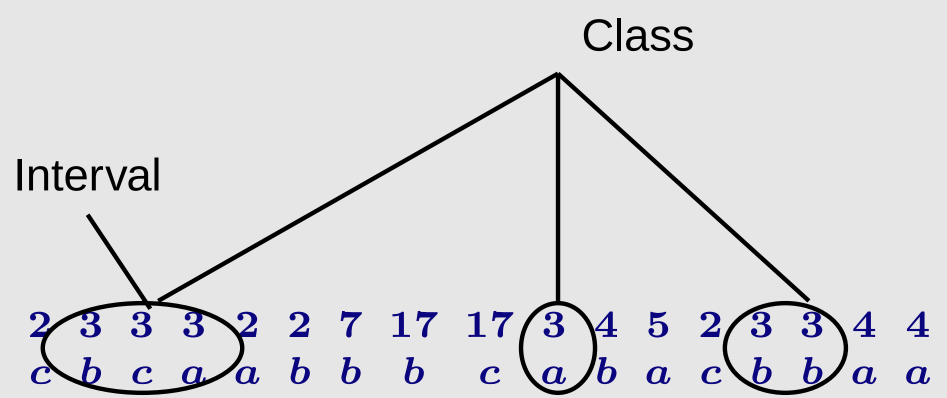
Satisfiability of $\textbf{FO}^2(+1, <)$ on data strings is decidable

Further results

- Emptiness of multicounter automata can be reduced to satisfiability of $\textbf{FO}^2(+1, <)$ on data strings
- $\textbf{FO}^2(<)$ on data strings is complete for **NEXPTIME**
- $\textbf{FO}^2(+1)$ on data strings is hard for **NEXPTIME** (and in **2NEXPTIME**)
- $\textbf{FO}^3(+1)$ on data strings is undecidable

Some Definitions

- Data string s:



- **Class**: all positions with the same data value
- **Interval**: (maximal set of) contiguous positions of a class
- **String projection $P(s)$**:

Decidability Results

Theorem

Satisfiability of **FO**$^2(+1, <)$ on data strings is decidable

Further results

- Emptiness of multicounter automata can be reduced to satisfiability of **FO**$^2(+1, <)$ on data strings
- **FO**$^2(<)$ on data strings is complete for **NEXPTIME**
- **FO**$^2(+1)$ on data strings is hard for **NEXPTIME**
  (and in 2**NEXPTIME**)
- **FO**$^3(+1)$ on data strings is undecidable

Some Definitions

- Data string s:

Class

Interval

2	3	3	3	2	2	7	17	17	3	4	5	2	3	3	4	4
c	b	c	a	a	b	b	b	c	a	b	a	c	b	b	a	a

- **Class**: all positions with the same data value
- **Interval**: (maximal set of) contiguous positions of a class
- **String projection $P(s)$**:

$$cbcaabbbcabacbbaa$$

What $\mathbf{FO^2}$ Can Express

Example properties

- Let α and β denote unary types
- $\mathbf{FO^2}$ can express
  - –
  - –
  - –
  - –
  - –
  - –

What FO^2 Can Express

Example properties

- Let α and β denote unary types
- FO^2 can express
  - data-blind properties, i.e., properties not using $\sim$
  -
  -
  -
  -
  -

What FO^2 Can Express

Example properties

- Let α and β denote unary types
- FO^2 can express
  - data-blind properties, i.e., properties not using $\sim$
  - All occurrences of a type α are in the same class:

$$\theta = \forall x \forall y \left((\alpha(x) \wedge \alpha(y)) \rightarrow x \sim y \right)$$

  -
  -
  -
  -

What FO^2 Can Express

Example properties

- Let α and β denote unary types
- FO^2 can express
  - **data-blind** properties, i.e., properties not using $\sim$
  - All occurrences of a type α are in the same class:
    $$\theta = \forall x \forall y \left((\alpha(x) \wedge \alpha(y)) \rightarrow x \sim y \right)$$
  - Each class contains at most one occurrence of α:
    $$\theta = \forall x \forall y \left((\alpha(x) \wedge \alpha(y) \wedge x \sim y) \rightarrow x = y \right)$$
  -
  -
  -

What FO^2 Can Express

Example properties

- Let α and β denote unary types
- FO^2 can express
  - data-blind properties, i.e., properties not using $\sim$
  - All occurrences of a type α are in the same class:
    $$\theta = \forall x \forall y \left((\alpha(x) \land \alpha(y)) \rightarrow x \sim y \right)$$
  - Each class contains at most one occurrence of α:
    $$\theta = \forall x \forall y \left((\alpha(x) \land \alpha(y) \land x \sim y) \rightarrow x = y \right)$$
  - In each class, every α occurs before every β:
    $$\theta = \forall x \forall y \left((\alpha(x) \land \beta(y) \land x \sim y) \rightarrow x < y \right)$$
  -
  -

What FO^2 Can Express

Example properties

- Let α and β denote unary types
- FO^2 can express
  - data-blind properties, i.e., properties not using $\sim$
  - All occurrences of a type α are in the same class:
$$\theta = \forall x \forall y \left((\alpha(x) \wedge \alpha(y)) \rightarrow x \sim y \right)$$
  - Each class contains at most one occurrence of α:
$$\theta = \forall x \forall y \left((\alpha(x) \wedge \alpha(y) \wedge x \sim y) \rightarrow x = y \right)$$
  - In each class, every α occurs before every β:
$$\theta = \forall x \forall y \left((\alpha(x) \wedge \beta(y) \wedge x \sim y) \rightarrow x < y \right)$$
  - Each class with an α also has a β:
$$\theta = \forall x \exists y \left(\alpha(x) \rightarrow (\beta(y) \wedge x \sim y) \right)$$
  -

What FO^2 Can Express

Example properties

- Let α and β denote unary types
- FO^2 can express
  - data-blind properties, i.e., properties not using $\sim$
  - All occurrences of a type α are in the same class:
  $$\theta = \forall x \forall y \left((\alpha(x) \wedge \alpha(y)) \rightarrow x \sim y \right)$$
  - Each class contains at most one occurrence of α:
  $$\theta = \forall x \forall y \left((\alpha(x) \wedge \alpha(y) \wedge x \sim y) \rightarrow x = y \right)$$
  - In each class, every α occurs before every β:
  $$\theta = \forall x \forall y \left((\alpha(x) \wedge \beta(y) \wedge x \sim y) \rightarrow x < y \right)$$
  - Each class with an α also has a β:
  $$\theta = \forall x \exists y \left(\alpha(x) \rightarrow (\beta(y) \wedge x \sim y) \right)$$
  - It turns out: That's basically all!

Proof Structure

Main steps of the proof

- $$FO^2 \text{ formula } \varphi$$
  $$\downarrow$$
  $$\text{Scott normal form}$$
  $$\downarrow$$
  $$\text{Intermediate normal form}$$
  $$\downarrow$$
  $$\text{Data normal form } \psi$$
- Construct multicounter automaton $\mathcal{A}_\psi$ such that:
  - $\mathcal{A}_\psi$ accepts a string w if and only if
  - there is a data string s with
    * $s \models \varphi$
    * $P(s) = w$
- Check whether $L(\mathcal{A}_\psi) \neq \emptyset$

Proof Structure

Main steps of the proof

- FO^2 formula φ
  $\downarrow$
  Scott normal form
  $\downarrow$
  Intermediate normal form
  $\downarrow$
  Data normal form ψ
- Construct multicounter automaton $\mathcal{A}_\psi$ such that:
  - $\mathcal{A}_\psi$ accepts a string w if and only if
  - there is a data string s with
    * $s \models \varphi$
    * $P(s) = w$
- Check whether $L(\mathcal{A}_\psi) \neq \emptyset$

Definition: Multicounter-automaton

- Nondeterministic string automaton (not: data string!)
- Finite number of counters
- Counter values ≥ 0 (or reject)
- No intermediate test whether counter is 0
- Acceptance if finally all counters are 0

Remarks

- Closely related to Petri nets
- Non-emptiness: decidable [Sacerdote, Tenney 77]
- Lower bound: **EXPSPACE**
- No elementary upper bound known

Normalization

Normal forms

- We transform into equivalent EMSO formulas
- Scott normal form: $\exists R_1, \ldots, R_k\ \forall x \forall y\, \chi \ \wedge\ \bigwedge_i \forall x \exists y\, \chi_i$
- Intermediate normal form:

$$\exists R_1 \cdots R_m \theta_1 \wedge \cdots \wedge \theta_n$$

- θ_i:

(1) $\forall x \forall y \quad (\delta(x,y) \geq 2 \wedge \alpha(x) \wedge \beta(y) \wedge \boxed{\begin{matrix} x \sim y \\ x \not\sim y \end{matrix}}) \rightarrow \boxed{\begin{matrix} x < y \\ x > y \\ \mathit{ff} \end{matrix}}$

(2) $\forall x \exists y \quad \alpha(x) \rightarrow (\beta(y) \wedge \boxed{\begin{matrix} x \sim y \\ x \not\sim y \end{matrix}} \wedge \boxed{\begin{matrix} x+1 < y \\ x+1 = y \\ x = y \\ x = y+1 \\ x > y+1 \end{matrix}})$

- Note: $\forall\forall$ without $+1$

Normalization (cont.)

Normal forms (cont.)

- **Data normal form**: Disjunction of formulas
  $$\exists R_1 \cdots R_n R_\# \; \theta_1 \wedge \cdots \wedge \theta_n$$
- θ_i:
  (a) data-blind
  (b) All α are in the same class
  (c) Each class contains at most one α
  (d) In each class, every α occurs before every β
  (e) Each class with an α also has a β
  (f) $R_\#$ marks the first position of each interval:
  $\forall x R_\#(x) \leftrightarrow \forall y (x = y + 1 \rightarrow x \not\sim y)$

Normalization (cont.)

Normal forms (cont.)

- **Data normal form**: Disjunction of formulas

$$\exists R_1 \cdots R_n R_{\#}\ \theta_1 \wedge \cdots \wedge \theta_n$$

- θ_i:
  (a) data-blind
  (b) All α are in the same class
  (c) Each class contains at most one α
  (d) In each class, every α occurs before every β
  (e) Each class with an α also has a β
  (f) $R_{\#}$ marks the first position of each interval:
  $\forall x R_{\#}(x) \leftrightarrow \forall y(x = y + 1 \rightarrow x \not\sim y)$

Normalization steps

FO^2 $\longrightarrow$ **Scott normal form:** standard

Scott normal form

$\longrightarrow$ **intermediate normal form:**

relatively straightforward

Intermediate normal form

$\longrightarrow$ **data normal form:**

- For each type α we capture the two left-most classes with α and the two rightmost classes with α by unary relations $R_1^\alpha, \ldots, R_4^\alpha$
- Case distinction on possible formulas (1) and (2)

$\longrightarrow$ in each case θ_i can be replaced by some “data normal” formulas

Construction of Multicounter automaton

Proof (cont.)

- Recall ingredients of data normal form:
  (a) data-blind
  (b) All α are in the same class
  (c) Each class contains at most one α
  (d) In each class, every α occurs before every β
  (e) Each class with an α also has a β
  (f) $R_{\#}$ marks the first position of each interval:
  $$\forall x R_{\#}(x) \leftrightarrow \forall y(x = y + 1 \rightarrow x \not\sim y)$$
- (a), (f): straightforward
- (c), (d), (e) induce regular conditions for each class: L
- (b) specifies regular conditions for some special classes: $L_1, \ldots, L_k$
- Multicounter automaton $\mathcal{A}$ accepts basically shuffle of $L, L_1, \ldots, L_k$

Construction of Multicounter automaton (cont.)

Proof (cont.)

- $\mathcal{A}$ accepts string projections of models of disjunctions of formulas
$$\exists R_1 \cdots R_n R_\# \; \theta_1 \wedge \cdots \wedge \theta_n$$
- $\mathcal{A}$ guesses a disjunct
- $\mathcal{A}$ guesses, for each position $R_1, \ldots, R_n, R_\#$

$\longrightarrow$ In particular: guesses intervals

- But: $\mathcal{A}$ does not know which intervals belong to the same class

---

- Special classes ($L_1, \ldots, L_k$) can be checked directly
(if α occurs in only one class this class is R_1^α)
- To check that all other class strings are in L:
  - Let $\mathcal{B}$ be a string automaton for L
  - $\mathcal{A}$ has one counter per state of $\mathcal{B}$
  - Counter C_q counts how many (non-special) class strings seen so far led to a state q
  - Complication: At interval border $\mathcal{A}$ can proceed from a state q that was just "reached" only if $C_q \geq 2$

Contents

Conclusion

What we have seen

Logic is useful for the theory of XML languages:

- MSO offers framework for schema languages
- MSO $\equiv$ regular node selecting queries
- Two-variable logic $\equiv$ `XPath`
- FO-logic $\equiv$ natural extension of `XPath`
- MSO helpful in the context of transformations
- Two-variable logic with data is decidable

Open

There remains a lot to be done, e.g.

- `XQuery`
- Automata in the presence of data values
- Practical relevance of logic-automata approach?

Conclusion

What we have seen

Logic is useful for the theory of XML languages:

- MSO offers framework for schema languages
- MSO $\equiv$ regular node selecting queries
- Two-variable logic $\equiv$ `XPath`
- FO-logic $\equiv$ natural extension of `XPath`
- MSO helpful in the context of transformations
- Two-variable logic with data is decidable

Open

There remains a lot to be done, e.g.

- `XQuery`
- Automata in the presence of data values
- Practical relevance of logic-automata approach?

Finally

Thank You!