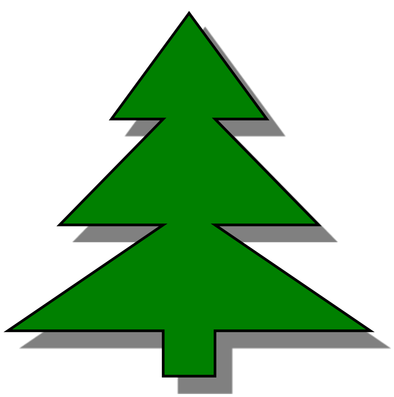
XML Transformation Language Based on Monadic Second Order Logic

Kazuhiro Inaba
Haruo Hosoya

University of Tokyo



PLAN-X 2007

Monadic Second-order Logic (MSO)

- First-order logic extended with “monadic second-order variables” ranging over sets of elements

e.g. $\forall A.(A \neq \varphi \Rightarrow \exists x.\ (x \text{ in } A \ \&\ \forall y.(y \text{ in } A \Rightarrow x \leqq y)))$

Variables Denoting Sets

Set Operations

Monadic Second-order Logic (MSO)

- As a foundation of XML processing
  - XML Query languages provably MSO-equivalent in expressiveness (Neven 2002, Koch 2003)
  - Theoretical models of XML Transformation with MSO as a sub-language for node selection (Maneth 1999, 2005)

Monadic Second-order Logic (MSO)

- Although used in theoretical researches …
  - No actual language system exploiting MSO formulae themselves for querying XML

- Why?
  - Little investigation on advantages of using MSO as a construct for XML programming
  - High time complexity for processing MSO (hyper-exponential in the worst-case), which makes practical implementation hard

What We Did

- Bring MSO into a practical language system for XML processing!
  - Show the advantages of using MSO formulae as a query language for XML
  - Design an MSO-based template language for XML transformation
  - Establish an efficient implementation strategy of MSO

MTran : http://arbre.is.s.u-tokyo.ac.jp/~kinaba/MTran/

Outline

- Why MSO Queries?
- MSO-Based Transformation Language
- Efficient Strategy for Processing MSO

Why MSO Queries?

MSO's Advantages

- No explicit recursions needed for deep matching
- Don't-care semantics to avoid mentioning irrelevant nodes
- N-ary queries are naturally expressible
- All regular queries are definable

	MSO	XPath	RegExp Patterns (XDuce)	Monadic Datalog
NoRecursion	○	○		
Don't-care	○	○		○
N-ary	○		○	
Regularity	○		○	○

Why MSO?
(1) No Explicit Recursion

- MSO does not require recursive definition for reaching nodes in arbitrary depth.
  - “Select all <img> elements in the input XML”

```
x in <img>
```

	MSO	XPath	RegExp Patterns	Monadic Datalog
NoRecursion	O	O		

Why MSO?
(2) Don't-care Semantics

- No need to mention irrelevant nodes in the query
  - MSO

```
ex1 y. x/y & y in <date>
```

  - Regular Expression Patterns
    - Requires specification for whole tree structures

```
x as ~[Any, date[Any], Any]
```

	MSO	XPath	RegExp Patterns	Monadic Datalog
Don't-care	○	○		○

Why MSO? (3) N-ary Queries

- Formulae with N free variables define N-ary queries
  - MSO

```
ex1 p. (p/x:<foo> & p/y:<bar> & p/z:<buz>)
```

  - XPath
    - Limited to 1-ary (absolute path) and 2-ary (relative path) queries

	MSO	XPath	RegExp Patterns	Monadic Datalog
N-ary	○		○	

Why MSO?
(4) Regularity

- MSO can express any “regular” queries.
  - i.e. the class of all queries that are representable by finite state tree automata

Lack of regularity is not just a sign of theoretical weakness, but has a practical impact…

	MSO	XPath	RegExp Patterns	Monadic Datalog
Regularity	O		O	O

Example: Generating a Table of Contents

- Input: XHTML
  - Essentially, a list of headings: <h1>, <h2>, <h3>, …

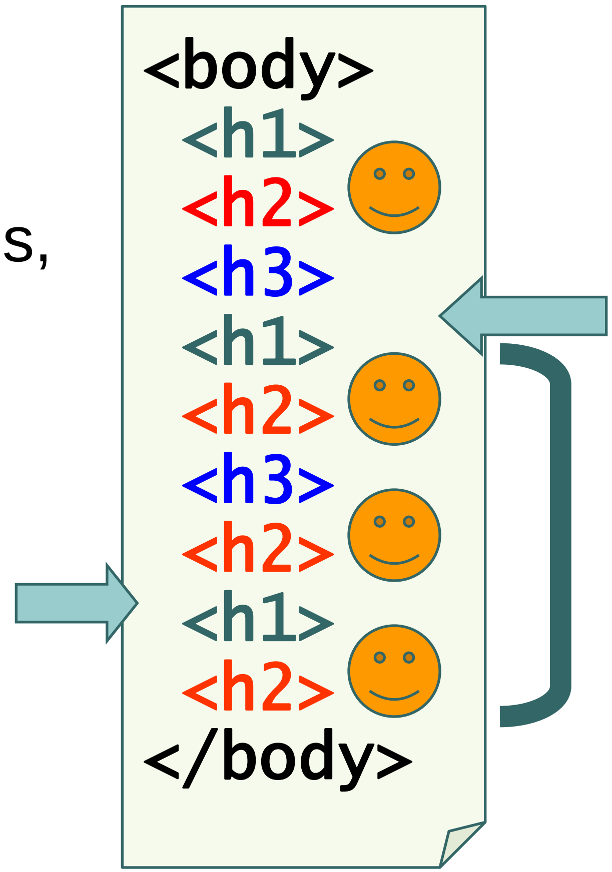
```
<html><body>
  <h1> <p> <h2> <p>
  <p> <h2> <p> <h3>
  <h1> <p> <h2> <p>
  <p> <p> <h3> <h1>
  <p> <p> <p> <p>
</body></html>
```

- Output
  - Tree structure

```
<ul>
  <li> h1 <ul>
    <li> h2 </li>
    <li> h2 <ul>
      <li> h3 </li>
    </ul></li>
  </ul></li>
  <li> h1 <ul>
    <li> h2 </li>
  </ul></li>
</ul>
```

Example: Generating a Table of Contents

- Queries required in this transformation
  - Gather all <h1> elements
  - For each <h1> element x,
    - Gather all subheading of x, that is,
      - All <h2> elements y that
      - Appears after x, and
      - No other <h1>s appear between x and y
    - For each <h2>, …
      - …



Example:
Generating a Table of Contents

- Straightforward in MSO
  - <h2> element y that
  - Appears after x, and
  - No other <h1>s appear between x and y.

```
y in <h2>
 & x < y
 & all1 z.(z in <h1> => ~(x<z & z<y))
```

Each condition is expressible in, e.g., XPath 1.0, but **combining them is difficult**. (Due to the lack of universal quantification.)

Example:
LPath[Bird et al., 2005] Linguistic Queries

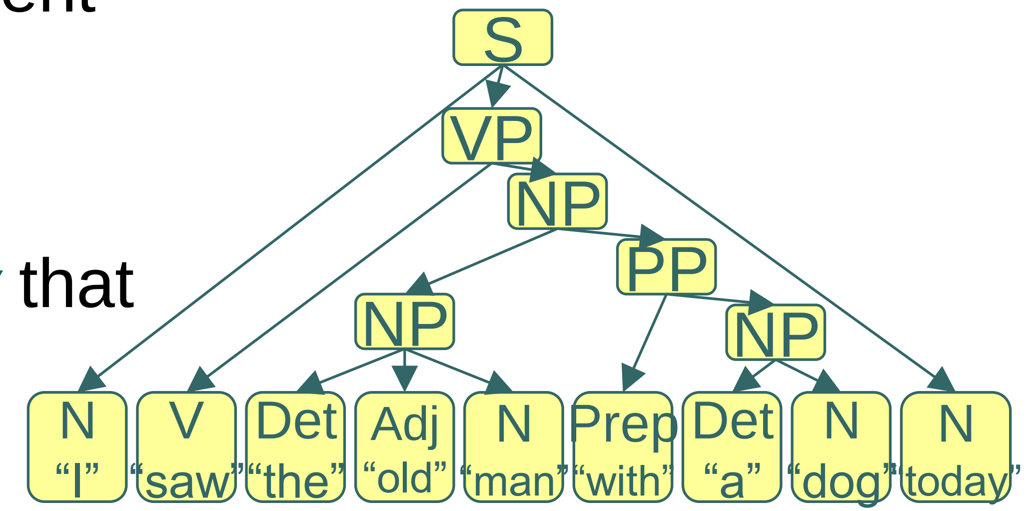
A linguistic query requiring "immediately following" relation

Input:

- Parse tree of a statement in a natural language

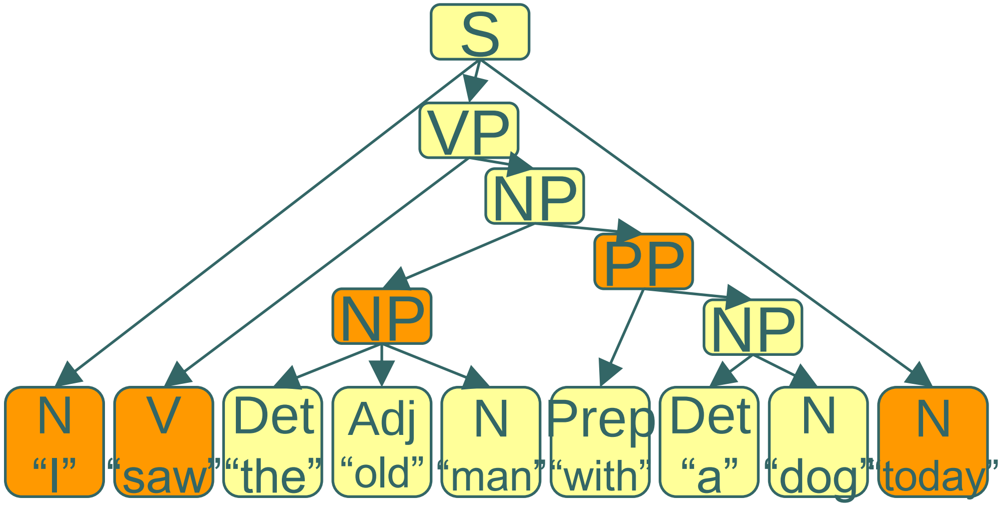
Query:

- "Select all elements *y* that follow after *x* in some proper analysis…"



Example: LPath[Bird et al., 2005] Linguistic Queries

- Proper analysis
  - A set P of elements such that
    - Every leaf node in the tree has exactly one ancestor contained in P



Example:
LPath[Bird et al., 2005] Linguistic Queries

- Straightforward in MSO
  - Every leaf node in the tree has exactly one ancestor contained in P.

```
pred is_leaf(var1 x) =
  ~ex1 y.(x/y);

pred proper_analysis(var2 P) =
  all1 x.(is_leaf(x) =>
    ex1 p.(p//x & p in P &
      all1 q.(q//x & q in P => p=q)));
```

Example:
LPath[Bird et al., 2005] Linguistic Queries

- "Immediately follows" query in MSO
  - "Select all elements *y* that follows after *x* in some proper analysis"

```
pred follow_in(var2 P, var1 x, var1 y) =
 x in P & y in P
  & ~ ex1 z. (z in P & x<z & z<y);

ex2 P. (proper_analysis(P) & follow_in(P,x,y))
```

Second-order variable!

MTran: MSO-Based Transformation Language

MTran: Overview

- “Select and transform” style templates (similar to XSLT)
  - Select nodes with **MSO queries**
  - Apply **templates** to each selected node
- Question:
  - “What is a design principle for templates that fully exploits the power of MSO?”
    - Simply adopting XSLT templates is not our answer

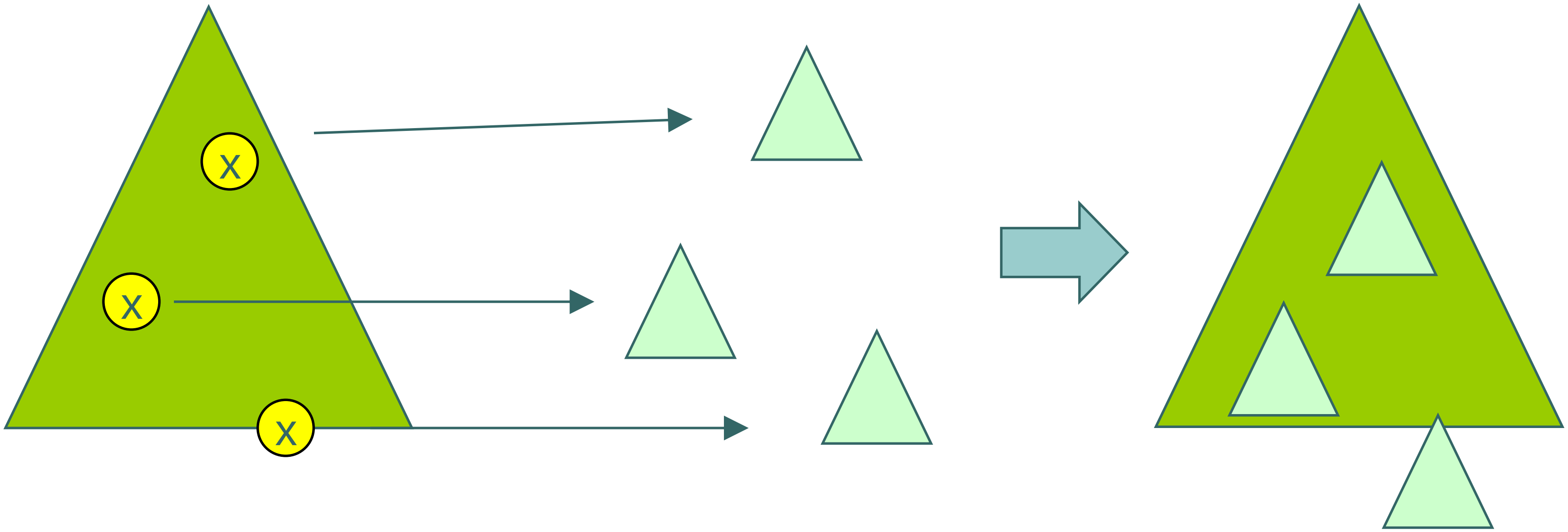
MTran: Overview

- MSO does not require explicit recursion
  - Natural design: transformation also does not require explicit recursion
- MSO enables us to write N-ary queries
  - Select a target node depending on N-1 previously selected nodes
    - XSLT uses XPath (binary queries) where the selection depends only on a single “context node”

1. No-recursion in Templates

- "Visit" template
  - Locally transform each node that matched φ(x)
  - Reconstruct whole tree, preserving unmatched part

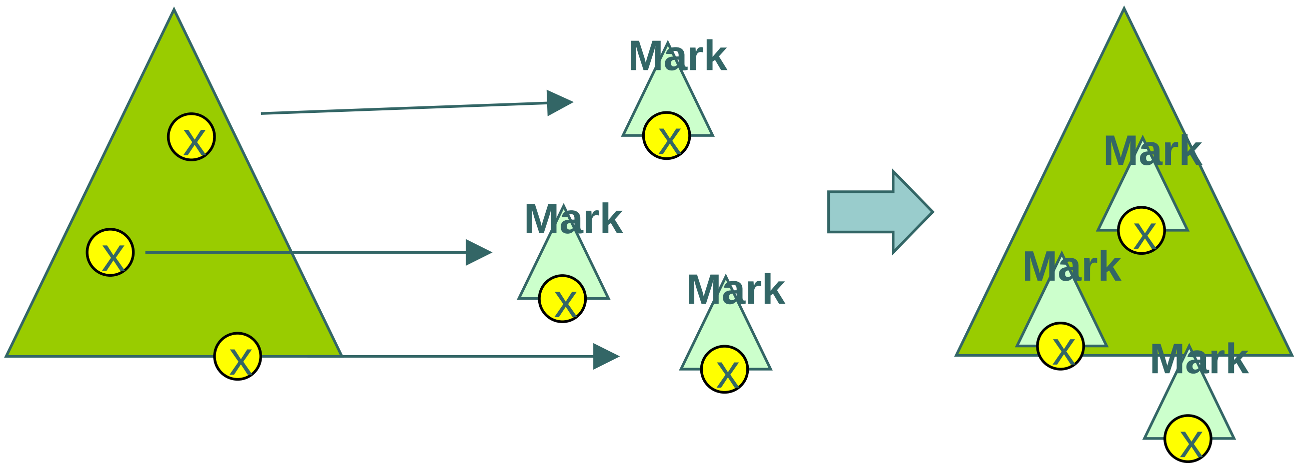
```
{ visit x :: Φ(x)  :: Subtemplate }
```



1. No-recursion in Templates

- E.g. wrap every <Traget> element by a <Mark> tag

```
{visit x :: x in <Target> :: Mark[x] }
```



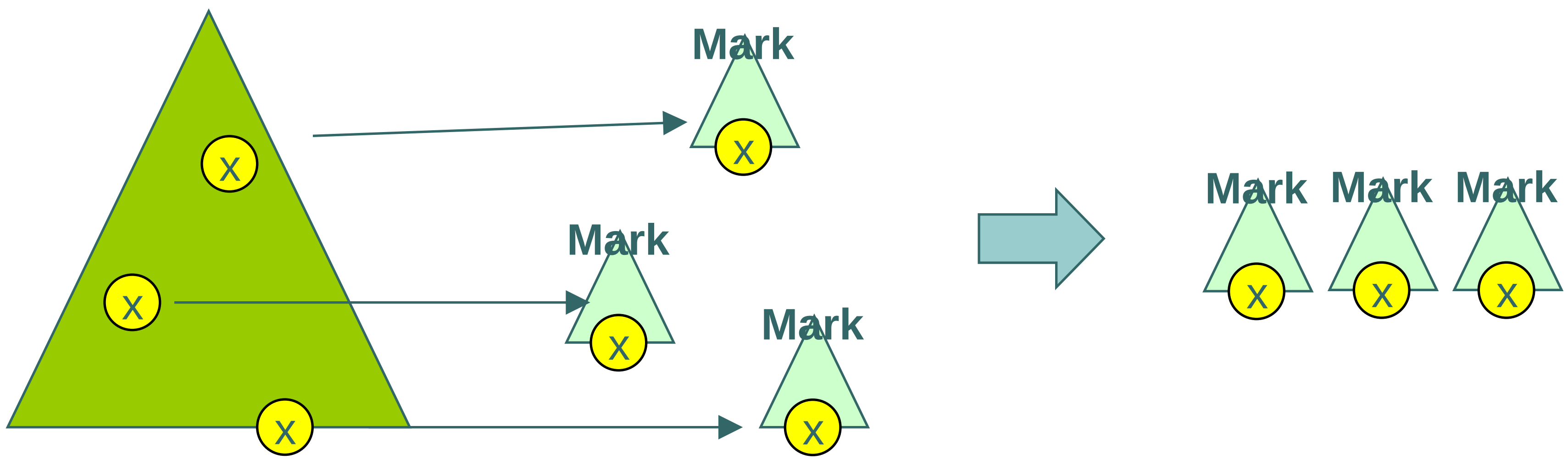

```
<Root>
  <Target/>
  <Target>
    <N><Target/></N>
  </Target>
</Root>
```

```
<Root>
  <Mark><Target/></Mark>
  <Mark><Target>
    <N><Mark><Target/></Mark></N>
  </Target></Mark>
</Root>
```

1. No-recursion in Templates

- “Gather” drops all unmatched part, and matched part are listed.

```
{gather x :: x in <Target> :: Mark[x]}
```



2. Nested Templates

- Nested query can refer outer variables

```
{visit x :: x in <textBox> ::
  {visit y from x
    :: textnode(y) ::
   span[
     @style[{gather z::ex1 p.(x/p/y & p/@style/z)::z}]
     y]
    :: y in <span> :: }}
```

```
<Document>
 <textBox>
  <span style="bold;">
   <span style="red;">
    Hi!
   </span>
  </span>
 </textBox>
</Document>
```

```
<Document>
  <textBox>

<span style="bold;red;">Hi!</span>

  </textBox>
</Document>
```

Efficient Strategy for Processing MSO

MSO Evaluation

- We follow the usual 2-step strategy…
  1. Compile a formula to a tree automaton
  2. Run queries using the automaton

Our Approach

① Compilation

- Exploit MONA[Klarlund et al.,1999] system
- Our contribution: experimental results in the context of XML processing

② Querying by Tree Automata

- Similar to Flum-Frick-Grohe [01] algorithm
  - O(|input| + |output|)
- Our contribution: simpler implementation via <u>partially lazy evaluation of set operations.</u>

Defining Queries by Tree Automata

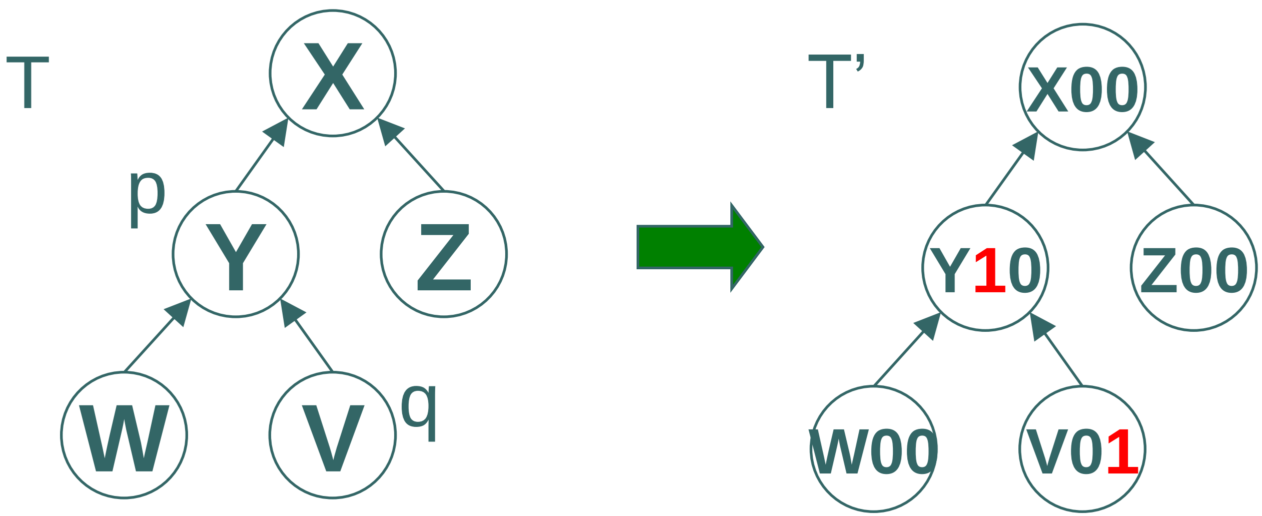
- An automaton runs on trees with alphabet $\Sigma \times \{0,1\}^N$ defines an N-ary query over trees with alphabet Σ

- $A = (\Sigma \times \{0,1\}^N, Q, \delta, q_0, F)$
  - $\Sigma \times \{0,1\}^N$: alphabet
  - Q : the set of states
  - δ : $Q \times Q \times \Sigma \times \{0,1\}^N \rightarrow Q$
  - q_0 : initial state
  - F : accepting states

Defining Queries by Tree Automata

"A pair (p,q) in tree T is an answer for the binary query defined by an automaton A"

⇔ "The automaton A accepts a *marked* tree T', (augmentation of T with "1" at p and q)"

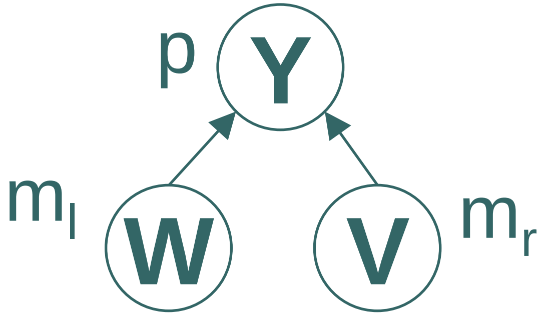


Algorithms for Queries in Tree Automata

- Naïve algorithm
  - For each tuple, generate a corresponding marked tree, and run the automaton
  - $O(|input|^{N+1})$

Algorithms for Queries in Tree Automata

- Naïve algorithm usings “sets”
  - For each node p and state q, calculate $m_p(q)$:
    - The set of tuples of nodes such that if they’re marked, the automaton reaches the state q at the node p
    - $\cup\{m_{root}(q) \mid q \text{ in } F\}$ is the answer
  - $m_p(q)$ is calculated in bottom-up manner



$$m_p(q) = \bigcup\{ m_l(q_1) \times \{p\} \times m_r(q_2) \mid \delta(q_1, q_2, Y1)=q \} \cup \bigcup\{ m_l(q_1) \times \{\} \times m_r(q_2) \mid \delta(q_1, q_2, Y0)=q \}$$

Flum-Frick-Grohe Algorithm

- Redundancies in naïve “set” algorithm
  - Calculation of sets that do not contribute to the final result ($m_{root}(q)$ for q in F)
  - Calculation on unreachable states
    - States that cannot be reached for any marking patterns
- Flum-Frick-Grohe algorithm avoids these redundancies by 3-pass algorithm
  - Detects two redundancies in 2-pass precalculations
  - Runs the “set” algorithm, avoiding those redundancies using results from the first 2-passes

Our Approach

- Eliminate the redundancies by simply implementing naive “set” algorithm by Partially Lazy Evaluation of Set Operations
  - Delays set operations (i.e., product and union) until it is really required
  - …except the operations over empty sets

```
type ‘a set   = EmptySet
              | NonEmptySet of ‘a neset
type ‘a neset = Singleton of ‘a
              | Union   of ‘a neset * ‘a neset
              | Product of ‘a neset * ‘a neset
```

Our Approach

- 2-pass algorithm
  - Run “set” algorithm using the partially lazy operations
  - Actually evaluate the lazy set

- Easier implementation
  - Implementation of partially lazy set operations is straightforward
  - Direct implementation of “set” algorithm is also straightforward (compared to the one containing explicit avoidance of redundancies)

Experimental Results

- Experiments on 4 examples
  - Compilation Time (in seconds)
  - Execution Time for 3 different sizes of documents

	Compile	10KB	100KB	1MB
ToC	0.970	0.038	0.320	3.798
LPath	0.655	0.063	0.429	4.050
MathML	0.703	0.236	1.574	16.512
RelaxNG	0.553	0.068	0.540	5.684

On 1.6GHz AMD Turion Processor, 1GB RAM, (sec). Units are in seconds.

Related Work

Related Work (MSO-based Transformation)

- DTL [Maneth and Neven 1999]
- TL [Maneth, Perst, Berlea, and Seidl 2005]
  - Adopt MSO as the query language.
  - Aim at finding theoretical properties for transformation models (such as type checking)
- MTran aims to be a practical system.
  - Investigation on: the design of transformation templates and the efficient implementation

Related Work (MSO Query Evaluation)

- Query Evaluation via Tree-Decompositions [Flum, Frick, and Grohe 2001]
  - Basis of our algorithm
  - Our contribution is “partially lazy operations on sets”, which allows a simpler implementation
- Several other researches in this area… [Neven and Bussche 98] [Berlea and Seidl 02] [Koch 03] [Niehren, Planque, Talbot and Tison 05]
  - Only restricted cases of MSO treated, or have higher complexity

Future Work

- Exact Static Type Checking
- Label Equality
  - “The labels of x and y are equal” is not expressible in MSO
    - But is useful in context of XML processing (e.g., comparison between @id and @idref attribute)
  - Can we extend MSO allowing such formulae, yet while maintaining the efficiency?

Thank you for listening!

- Implementation available online:
  - http://arbre.is.s.u-tokyo.ac.jp/~kinaba/MTran/