XML TRANSFORMATION LANGUAGE BASED ON MONADIC SECOND ORDER LOGIC

by

Kazuhiro Inaba

A Master Thesis

Submitted to
the Graduate School of the University of Tokyo
on February 28, 2006
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Information Science

Thesis Supervisor: Haruo Hosoya
Assistant Professor of Information Science

ABSTRACT

Although monadic second-order logic (MSO) has been a foundation of XML queries, little work has attempted to take MSO formulae themselves as a programming construct. Indeed, MSO formulae are capable of expressing (1) all regular queries, (2) deep matching without explicit recursion, (3) queries in a "don't-care semantics" for unmentioned nodes and (4) n-ary queries for locating n-tuples of nodes. While previous frameworks for subtree extraction (path expressions, pattern matches, etc.) each had some of these properties, none has satisfied all of them.

In this thesis, we have designed and implemented a practical XML transformation language called MTran that fully exploits the expressiveness of MSO. MTran is a language based on "select-and-transform" templates similar in spirit to XSLT. However, we specialize our templates for expressing structure-preserving transformation so as to avoid any recursive calls to explicitly be written. Moreover, we allow templates to be nested so as to make use of an n-ary query that depends on the $n-1$ nodes selected by the preceding templates.

For the implementation of the MTran language, we have developed, as the core part, an efficient evaluation strategy for n-ary MSO queries. This consists of (a) an exploitation of the existing MONA system for the translation from MSO formulae to tree automata and (b) our novel query evaluation algorithm for tree automata. Although a linear time algorithm has been known for nullary and unary queries, the best known algorithm for n-ary queries takes $O(rt+s)$ time in the worst case where t is the size of the input document, s is that of the output, and r is the size of *potential* matches during the query calculation, which can grow up to t^{n-1} in the worst case. We have developed a more efficient $O(t+s)$ algorithm for n-ary queries by using partially lazy evaluation of set operations.

Acknowledgements

We would like to thank Professor Haruo Hosoya for his invaluable comments and suggestions. We gratefully acknowledge Professor Mizuhito Ogawa for the discussion of our ideas. We have also learned a great deal from discussions with the members of the Principle of Programming Languages seminar in the University of Tokyo.

Contents

1 Introduction **1**

1.1 XML Transformation Language 1

1.2 Our Approach 2

1.2.1 Monadic Second Order Logic 2

1.2.2 XML Transformation with MSO 4

1.2.3 Algorithm 7

1.3 Related Work 8

1.3.1 Query Algorithm 8

1.3.2 Language Design 9

1.4 Structure of the Thesis 10

2 Language **11**

2.1 Binary Trees and XML representation 11

2.2 Query Expressions 12

2.2.1 Core Syntax 13

2.2.2 Semantics 14

2.2.3 Auxiliary Syntax 15

2.3 Transformation Templates 18

2.3.1 Overview 18

2.3.2 Syntax 20

2.3.3 Semantics 20

3 Evaluation Algorithm **26**

3.1 Preliminaries 26

3.1.1 Queries over trees 26

3.1.2 Tree Automata 27

3.2 From MSO to Tree Automata 28

3.3 N-ary Query Algorithm 28

3.3.1 Basic Algorithm 28
3.3.2 Our Algorithm 34
3.4 Transformation Template Evaluation 38
3.4.1 Efficient N-ary Queries Using Context Information 39

4 Experiments **41**
4.1 Examples 41
4.1.1 XHTML Table of Contents 41
4.1.2 Linguistic Queries 44
4.1.3 Relax NG Simplification 45
4.2 Preliminary Performance Evaluation 48
4.2.1 N-ary query performance 49

5 Conclusion and Future Work **52**
5.1 Conclusion 52
5.2 Future Work 52

References **54**

Chapter 1

Introduction

1.1 XML Transformation Language

Extensible Markup Language (XML [1]) is a language to represent tree-like data, standardized by World Wide Web Consortium (W3C). XML is becoming more and more commonly adopted in many areas as standard data formats such as XHTML, RDF, SVG, MathML, etc. To cover these increasing demands, flexible and expressive framework for XML processing is desired.

XML processing is essentially a process of XML *transformation*. We query data from input XML documents, convert them into another desired XML format, and output them. For XML queries, as an analogy to first-order logic being a basis for relational queries, monadic second-order logic (MSO) has gradually stabilizing its position as a foundation. For example, there have been proposals for XML query languages whose expressivenesses are provably MSO-equivalent [2, 3] and for theoretical models for XML transformation with MSO as a sublanguage for node selection [4, 5]. However, little attempt has been made for bringing MSO logic formulae themselves into an actual language system for XML processing.

The goal of our work is to design and implement a practical XML transformation language based on MSO queries. We implement a language MTran, and in particular, address the following two challenges:

- a surface language design for XML transformation that leverages the strength of MSO queries, and
- an efficient algorithm to process MSO queries.

	Regularity	No Recursion	Don't care	N-ary
Pattern	√			√
Path		√	√	
Datalog	√		√	
MSO	√	√	√	√

Table 1.1: Comparisons between query languages

1.2 Our Approach

1.2.1 Monadic Second Order Logic

MSO is first-order logic extended with second-order variables ranging over sets of domain elements—i.e., tree nodes, in the setting of XML queries—in addition to first-order variables ranging over domain elements themselves. Why do we think that such logic is suitable for writing queries on XML documents? The reasons are fourfold.

- The class of all regular queries can be captured.
- No explicit recursions are required to locate nodes distant from context nodes.
- There is no need to mention the nodes that are irrelevant to the query ("don't-care semantics").
- N-ary queries are naturally expressible.

While existing languages such as path expressions [6, 7, 8, 9, 10, 11], pattern matches [12, 13], and monadic datalog queries [3] have some of these properties, MSO is the only language that has all of them, as we argue below.

Regularity

A query over trees is called regular when there is an equivalent tree automaton with an appropriate alphabet (Section 3.1.2). MSO is known to be able to express all regular queries [14], while most of existing path-based node selection languages (including XPath [6], currently the most popular path language) do not have this property. This lack of regularity does not only indicate theoretical weakness in expressive power, but also has a practical impact since it fails to be able to represent even slightly complicated conditions for node selection. An obvious example is that one cannot write "select every node that conforms to a specified schema for XML" since schemas written in usual schema languages like DTD [1], XML Schema [15], and RELAX NG

[16] heavily rely on regular expressions for trees (in particular, RELAX NG schemas can represent any regular tree languages) and thus exceed the expressiveness of those path-based selection languages. As a more realistic example, the following query

> "select, from an XHTML document, every `<h2>` node that appears between the current node and the next `<h1>` node from the current node in the document order"

is naturally expressible in MSO as we will see in Section 4.1.1, whereas it is not in most path languages.

No recursion

The way that MSO formulae express retrieval conditions is in a sense logically direct and, in particular, it does not require recursively defined constraints for reaching nodes that are located in arbitrarily deep positions. Several query languages such regular expression patterns and monadic datalog, while being able to capture all regular queries, incur recursive definitions for deep matching. As a result, even an extremely simple query like "select all `<a>` elements in the input document" needs an explicit recursion. Writing down recursion is often tedious work and in particular unfriendly to naive programmers; it is much more helpful to be able to express such a simple query like

```
  x in <a>
```

as in MSO.

Don't-care semantics

The directness of MSO's way of expression also allows us to completely avoid mentioning nodes that are irrelevant to the query. It is in contrast to some languages such as regular expression patterns, where we need to specify conditions that the whole tree structure should satisfy. For example, suppose that we want to write a query that retrieves the set of nodes x whose at least one child node labeled `<b>`. In regular expression patterns, we would write as follows

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

where we have to "mention" the siblings and the content of the `<b>` node by the wild card `Any` to complete the pattern. In MSO, on the other hand, we can write in the following way

```
ex1 y: x/y & y in <b>
```

where we only refer to the nodes of our interest: the node x itself and the child `<b>` node y. No condition is ever explicitly specified for other irrelevant nodes, even by wildcards. This "don't-care semantics" might not be advantageous for specifying a very complicated constraint such as conformance to a schema, while it makes most of usual queries extremely concise.

N-ary queries

An n-ary query locates n-tuples of nodes of the input XML tree that simultaneously satisfy a specified condition. MSO, as it is a formal logic, can naturally express n-ary queries by formulae with distinct n free variables. For example, the following ternary query

```
ex1 p: p/x & p/y & p/z &
       x in <a> & y in <b> & z in <c>
```

expresses the condition that given three nodes x, y, and z share a common parent node p and are tagged with `<a>`, `<b>`, and `<c>`, respectively. Although path-based query languages like XPath suit to express binary queries that express relations between a previously selected node and another, they cannot represent general n-ary queries. Similarly, monadic datalog can express arbitrary unary MSO formulae but not any higher-arity queries.

1.2.2 XML Transformation with MSO

MSO by itself is thus a powerful specification language for node selection. However, our aim is to further make use of MSO formulae for the transformation of XML documents. Then, the question is: what is a language design principle that fully exploits the high expressive power of MSO?

Structure-preserving transformation

Since it is one of MSO's advantages that we can select nodes in any depth with no explicit recursions, it would paradigmatically be smooth if we can also express a transformation of trees of any depth without any recursions. For example, given an input document where `<A>` elements may be nested one inside another, consider the transformation that encloses every `<A>` element in the input document by a new `<B>` element. In our language, this transformation can be written by the following one line:

```
{visit x :: x in <A> :: B[x]}
```

Here, we first select all the `<A>` elements by the MSO formula ”`x in <A>`” and then transform each of these elements accordingly to the rule “`B[x]`,” i.e., enclose the element by the `<B>` tag. The whole output is the reconstruction of the input tree where each select element is replaced by the result of its local transformation.

Compare the above program in our language with the same transformation written in XSLT:

```
<xsl:stylesheet version="1.0" ...>
  <xsl:template match="A">
    <B><A><xsl:apply-templates/></A></B>
  </xsl:template>
  <xsl:template match="@*|node()">
    <xsl:copy>
       <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

In this, we similarly select all the `<A>` elements (by an XPath expression) and then, for each selected element, we create a `<B>` element containing an `<A>` element. The most important difference, however, is that XSLT requires us to explicitly write a recursive application of the template to the child nodes (`<xsl:apply-templates/>`) for computing the content of the `<A>` element. Our design principle is to eliminate such explicit recursion for avoiding the necessity to follow the data flow for understanding the program, thus making transformation more intuitively readable and writable for naive programmers.

Choice of `visit` and `gather`

In MSO, we only need to mention the nodes that are relevant to the query and can directly jump into them without any navigation. We would like to lift this concept to our transformation language. Above, we have illustrated a transformation that jumps directly to each matched node and replaces it with an output fragment. In this case, the nodes that are *not* matched are retained in the result. However, in general, we provide two choices for the treatment of such unmatched nodes, either to keep them or to drop them. These correspond to the keywords `visit` and `gather`, respectively, and can be specified by the user for each use of an MSO formula. For example, the following

```
<root>
  {gather x :: x in <A> :: B[x]}
</root>
```

is similar to the previous example except that it uses `gather` instead of `visit`. As a result, the `root` element of the output XML contains a list of all `<A>` elements appearing in the input XML, each wrapped by a `<B>` element.

Nested templates

XSLT uses XPath binary queries for selecting a node with respect to a single previously selected node, i.e., the context node. Our language pushes this approach further for exploiting MSO's capability to express general n-ary queries. Specifically, we allow templates to be nested and an inner MSO formula to refer to variables that are bound in the outer templates. For example, see the following program:

```
List[
  {gather p :: p in <map> ::
    {gather n :: p/<name>/n ::
      {gather v :: p/<value>/v ::
        Pair[ n ", " v ]
  }}}
]
```

Figure 1.1: A transformation using binary queries

This program converts a document representing a one-to-many mapping, e.g.,

```
<mapping>
  <map>
    <name>Hello</name>
    <value>1</value>
    <value>2</value>
  </map>
  <map>
    <name>World</name>
    <value>3</value>
    <value>4</value>
  </map>
</mapping>
```

to another representing a many-to-many mapping:

```
<List>
  <Pair>Hello, 1</Pair>
  <Pair>Hello, 2</Pair>
  <Pair>World, 3</Pair>
  <Pair>World, 4</Pair>
</List>
```

Note that the selection conditions for the variables `n` and `v` both depend on the outer variable `p`. Although this example only uses binary queries, it is easy to see that we can also specify higher-arity queries in the same framework. Another point of this example is that, inside the scope where we have already selected a node for `n`, we can directly refer to the variable `p` that is bound in the two-block outer scope. Note that such flexibility, which is not present in XSLT, can naturally be obtained by the combination of logic formulae with free variables and nested templates with lexical scoping.

1.2.3 Algorithm

In order to implement a practical system for our transformation language, what we critically need is an efficient evaluation algorithm for n-ary MSO queries, that is, one that takes, as inputs, an MSO formula with n free variables and a tree structure and calculates the set of n-tuples that satisfy the formula. Our language system uses an n-ary query algorithm for the following two situations.

- selection of tuples of nodes that simultaneously satisfy the condition, and
- selection of nodes that satisfy the condition relative to nodes already selected by previous queries.

The second point has first been pointed out by Berlea et al. [17] in the context of their language based on binary queries, but can easily be extended to our case with n-ary queries. To illustrate this point, let us see again the example in Figure 1.1. First of all, notice that, even if we do not have an n-ary query algorithm, we can process the transformation using only a unary query algorithm. That is, we first locate all the `<map>` elements in the input document. Then, for *each* `<map>` element, we execute the inner formula `p/<name>/n`, interpreting it as a unary query on the variable `n` under the fixed the binding of the variable `p` to the `map` element. Unfortunately, this strategy is inefficient since the above formula is evaluated as many times as the number of the `<map>` elements present in the input document; since a unary query takes a linear time in the size of the input, the binary query that we wanted would take a quadratic time. Berlea et al.'s observation was that this could be improved if there is an efficient algorithm for binary queries that evaluates, in our example, the above formula only once for locating all pairs of an element and a `<name>` element that are in the parent-child relation.

We have developed an efficient implementation strategy for n-ary MSO queries. This consists of usual two steps: (1) compilation of MSO formulae to tree automata and (2) evaluation of n-ary queries represented by tree automata. The first step is well known to take a non-elementary time in the worst case. Our approach is to exploit the MONA system [18], which has an established reputation in its compact and efficient representation of MSO formulae by tree automata with binary decision diagrams and is experimentally shown to work quickly for large formulae even of dozens of kilobytes [19]. We have also made a preliminary experiment and confirmed that, for many typical examples of XML queries, MONA yields an enough performance. (Section 4.2)

For the second step, we have developed a novel linear-time algorithm for n-ary queries. The previously best known result was Berlea and Seidl's $O(rt + s)$ algorithm for binary queries where t is the size of the input document, s is the number of output tuples, and r is the size of *potential* matches during the query calculation. Since r may grow up to t, their algorithm takes a quadratic time in the worst case. When generalized to n-ary queries, r may grow up to t^{n-1} and the total worst case complexity is $O(t^n)$. We have improved Berlea's algorithm and obtained an $O(t + s)$ algorithm for general n-ary queries. We have achieved this by using partially lazy operations on sets of nodes so as to eliminate unnecessary computation of potential matches.

1.3 Related Work

1.3.1 Query Algorithm

Several algorithms for MSO queries have been proposed. For unary queries, Neven and den Bussche have given a linear-time, two-pass algorithm based on boolean attribute grammars [20]. Koch [3], in his XML query language Arb based on monadic datalog, has developed a different linear-time, two-pass algorithm using *selecting tree automata*, which exactly captures the class of unary MSO queries.

The binary query algorithm proposed by Berlea and Seidl [17], which has been used for the implementation of the fxt functional XML transformer [21], is the basis of our algorithm presented in this paper. We have shown that their worst case time complexity $O(t^2 + s)$ can be improved to $O(t + s)$ (where t and s are respectively the sizes of the input and the output) by using lazy set operations, and furthermore our strategy can be generalized it to n-ary queries without sacrificing the linear time complexity.

Niehren et al.[22] have presented a linear time algorithm for n-ary MSO queries in the restricted case where the query is essentially a Cartesian product of n unary queries. They have shown that existential runs of unambiguous tree automata capture this class of queries and presented an effective algorithm to decide the unambiguity of tree automata. However, as far as we can tell from our experiences using our language, almost all n-ary queries appearing in practical settings use variables in a dependent way and thus do not fulfill their restriction. (See, for example, the MSO queries shown in Chapter 4).

1.3.2 Language Design

DTL [4] and its generalization TL [5] are theoretical models for XML transformation that use, like us, MSO formulae for node selection. However, their purpose of research is to seek for theoretical properties of transformation models whereas ours is to obtain a concrete design and an efficient implementation technique for a transformation language leveraging the full strength of MSO. Thus, on one hand, they have found desirable properties such as decidability of precise typechecking, which is not known for our language. On the other hand, we have incorporated a number of design considerations not present in their languages. Specifically, our language allows transformation of arbitrarily deep trees without recursion, while theirs incurs explicit recursion; ours provides the choice of retaining and dropping for nodes not selected by queries, while theirs allows only the second; ours allows nested templates to make use of n-ary queries, while their is limited to binary queries.

Nakano has proposed an XML transformation language XTiSP [23] that has `visit` and `invite` constructs. Although these look similar to our `visit` and `gather`, they have slightly simpler semantics (e.g., they can go only forward) for their primary purpose of stream processing. Also, XTiSP uses path expressions for node selection and thus is limited to binary queries.

1.4 Structure of the Thesis

The rest of this thesis is organized as follows. In Chapter 2, we introduce the formal syntax and semantics of our MTran language. We first argue about the query part of MTran, and then explain the whole transformation language. Chapter 3 gives a detailed explanation of our query algorithm of n-ary MSO formulae after a brief review of some known facts on MSO and tree automata. In Chapter 4, we give slightly bigger examples in MTran, and show how the expressiveness of MSO is useful in practice. Chapter 5 concludes this thesis and suggests possible directions for future work.

Chapter 2

Language

2.1 Binary Trees and XML representation

Definition 1. Let Σ be an alphabet set. A binary tree t over Σ is a mapping from a finite prefix-closed set $Pos(t) \subseteq \{\mathtt{l}, \mathtt{r}\}^*$ into Σ. An element $p \in Pos(t)$ is called a *position* or a *node* of t, and the alphabet $t(p) \in \Sigma$ assigned to p is called the *label* of p. The empty sequence node ε is called the *root* of t.

Our goal is to handle XML documents, which in general are not binary trees but are unranked trees (trees whose each node has an arbitrary number of child nodes). Since our algorithm to query XML is designed to use binary trees, we encode an unranked tree as a binary tree using a widely accepted encoding. The first child of a node in unranked trees is regarded as the left child of the node in the encoded binary tree, and the right neighboring sibling in unranked trees is encoded as the right node of the node. Figure 2.1 shows an example of this encoding.

In MTran, XML documents are modeled as trees containing three types of node: element nodes, attribute nodes, and text nodes. The alphabet Σ for the model tree is

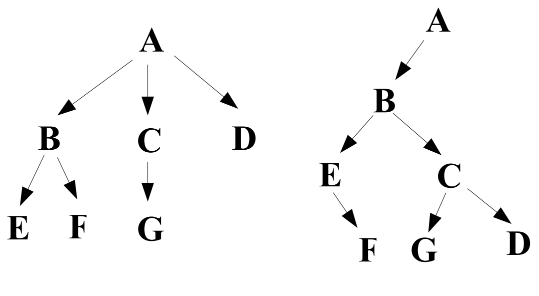


Figure 2.1: Encoding an unranked tree(left) to a binary tree(right)

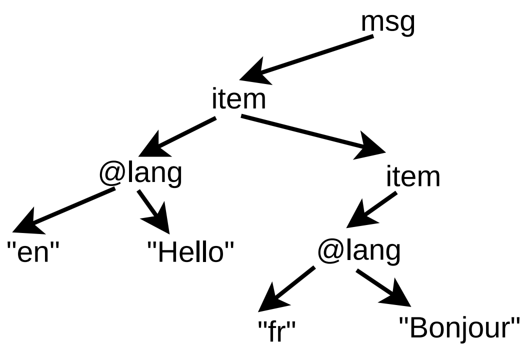


Figure 2.2: The binary tree representation of the example XML

the union of element names, attribute names, and text content appearing in the input document. For example, the following XML document:

```
<msg>
  <item lang="en">Hello</item>
  <item lang="fr">Bonjour</item>
</msg>
```

is modeled as a tree with $\Sigma = \{$`msg`, `item`, `@lang`, `"en"`, `"Hello"`, `"fr"`, `"Bonjour"`$\}$. Here, the symbol `@` of `@lang` represents that this name is for attributes. Attribute nodes are treated as children of the belonging element node inserted before other ordinary child nodes. We distinguish attributes nodes from other nodes only by means of the labels. Thus, the previous example is represented as shown in the binary tree in Figure 2.2.

Our current implementation does not support comments and processing instructions, and simply ignores them. Also, it does not yet treat XML namespaces. Those are left as future work.

2.2 Query Expressions

Query expressions specify conditions of nodes that we want to extract from the input XML documents, and can be embedded in transformation templates. In this section, we first describe the core syntax of query expressions. We next define the semantics of expressions in the core syntax. We then give syntax sugars on top of the core primitives for the convenient and reusable description of queries.

2.2.1 Core Syntax

As we mentioned in the introduction, we adopt Monadic Second-Order Logic (MSO) formulae as our query language. MSO is a kind of logic that allows the use of second-order variables ranging over sets of domain elements in addition to first-order variables ranging over domain elements themselves. The following BNF defines the core syntax:

$$
\begin{aligned}
x &\in \text{First-order variable symbols}\\
X &\in \text{Second-order variable symbols}\\
e &\in \text{XML element names}\\
a &\in \text{XML attribute names}
\end{aligned}
$$

$$
\begin{aligned}
p &::= x \mid \texttt{root}\\
S &::= X \mid \texttt{<}e\texttt{>} \mid \texttt{@}a \mid \texttt{<*>} \mid \texttt{@*} \mid \texttt{\#}\\
\varphi &::= p \texttt{ = } p \mid S \texttt{ = } S \mid p \texttt{ in } S\\
&\quad\mid \texttt{\~{}}\varphi \mid \varphi\texttt{\&}\varphi \mid \varphi\texttt{|}\varphi \mid \varphi\texttt{=>}\varphi\\
&\quad\mid \texttt{ex1 } x\texttt{:}\varphi \mid \texttt{all1 } x\texttt{:}\varphi \mid \texttt{ex2 } X\texttt{:}\varphi \mid \texttt{all2 } X\texttt{:}\varphi\\
&\quad\mid \texttt{firstChild(}p\texttt{,}p\texttt{)} \mid \texttt{nextSibling(}p\texttt{,}p\texttt{)}
\end{aligned}
$$

A first-order term is denoted by p and a second-order term by S. The first-order constant symbol `root` denotes the root node of the input tree. Each constant set symbol `<`e`>` denotes the set of element nodes labeled e in the input XML document, and each constant set symbol `@`a denotes the set of attribute nodes labeled `@`a similarly. Th other three second-order terms `<*>`, `@*`, and `#` are called *wild-card* terms denoting the set of any element nodes, any attribute nodes, and any text nodes, respectively.

The last two constructs of φ are the primitives to describe the property of tree structures. As its name shows, `firstChild(`p,q`)` becomes true if and only if the first child of node p is q. The relation `nextSibling(`p,q`)` is defined similarly. These two primitives correspond to the edge relations in the binary tree encoding we introduced in Section 2.1.

Example 1. The following query requires that for a node `x`, this node and all its left descendants are element nodes labeled as `a`:

```
ex2 R: (x in R  &  all1 y:all1 z:(y in R & firstChild(y,z) => z in R)
                &  all1 w:(w in R => w in <a>))
```

2.2.2 Semantics

Now we give the semantics for MSO formulae. We first define the interpretation of terms.

Definition 2. Let Σ be an alphabet and t be a binary tree over Σ. Under an assignment γ that maps each first-order variable to an element of $Pos(t)$, first-order terms are interpreted as follows:

$$\begin{aligned}\gamma[x] &= \gamma(x)\\ \gamma[\texttt{root}] &= \varepsilon\end{aligned}$$

Also, under an assignment Γ for second-order variables, second-order terms are interpreted as follows:

$$\begin{aligned}\Gamma[X] &= \Gamma(X)\\ \Gamma[\texttt{<}e\texttt{>}] &= \{p \mid t(p) = e\}\\ \Gamma[\texttt{@}a] &= \{p \mid t(p) = \texttt{@}a\}\\ \Gamma[\texttt{<*>}] &= \bigcup_{e\in\Sigma}\{p \mid t(p) = e\}\\ \Gamma[\texttt{@*}] &= \bigcup_{\texttt{@}a\in\Sigma}\{p \mid t(p) = \texttt{@}a\}\\ \Gamma[\texttt{\#}] &= \bigcup_{\texttt{"s"}\in\Sigma}\{p \mid t(p) = \texttt{"s"}\}\end{aligned}$$

Definition 3. An MSO formula φ is interpreted under a binary tree t over Σ, a first-order assignment γ, and a second-order assignment Γ, as follows:

$$\begin{aligned}
t,\gamma,\Gamma \models \texttt{p}_1 = \texttt{p}_2 &\iff \gamma[\texttt{p}_1] = \gamma[\texttt{p}_2]\\
t,\gamma,\Gamma \models \texttt{S}_1 = \texttt{S}_2 &\iff \Gamma[\texttt{S}_1] = \Gamma[\texttt{S}_2]\\
t,\gamma,\Gamma \models \texttt{p in S} &\iff \gamma[\texttt{p}] \in \Gamma[\texttt{S}]\\
t,\gamma,\Gamma \models \texttt{\~{}}\varphi &\iff t,\gamma,\Gamma \not\models \varphi\\
t,\gamma,\Gamma \models \varphi_1\&\varphi_2 &\iff t,\gamma,\Gamma \models \varphi_1 \text{ and } t,\gamma,\Gamma \models \varphi_2\\
t,\gamma,\Gamma \models \varphi_1|\varphi_2 &\iff t,\gamma,\Gamma \models \varphi_1 \text{ or } t,\gamma,\Gamma \models \varphi_2\\
t,\gamma,\Gamma \models \texttt{ex1 } x\texttt{:}\varphi &\iff \text{there exists } a \in Pos(t) \text{ s.t. } t,\gamma_{x:=a},\Gamma \models \varphi\\
t,\gamma,\Gamma \models \texttt{all1 } x\texttt{:}\varphi &\iff \text{for all } a \in Pos(t) \quad t,\gamma_{x:=a},\Gamma \models \varphi\\
t,\gamma,\Gamma \models \texttt{ex2 } X\texttt{:}\varphi &\iff \text{there exists } A \in 2^{Pos(t)} \text{ s.t. } t,\gamma,\Gamma_{X:=A} \models \varphi\\
t,\gamma,\Gamma \models \texttt{all2 } X\texttt{:}\varphi &\iff \text{for all } A \in 2^{Pos(t)} \quad t,\gamma,\Gamma_{X:=A} \models \varphi
\end{aligned}$$

$$t, \gamma, \Gamma \models \texttt{firstChild}(\texttt{p}_1, \texttt{p}_2) \iff \gamma(p_1).\texttt{l} = \gamma(p_2)$$

$$t, \gamma, \Gamma \models \texttt{nextSibling}(\texttt{p}_1, \texttt{p}_2) \iff \gamma(p_1).\texttt{r} = \gamma(p_2)$$

Here, $\gamma_{x:=a}$ represents a variable assignment that maps the variable x to a and the other variables to the same nodes as γ does. An assignment $\Gamma_{X:=A}$ for second-order variables is defined similarly. The dot operator . used in the definition of `firstChild` and `nextSibling` denotes the concatenation of sequences from {`l`,`r`}.

2.2.3 Auxiliary Syntax

Macro Expressions

Programmers can define their own macros for frequently used formulae by the form

$$\begin{aligned} m &\in \text{Macro names} \\ V &::= \texttt{var1}\ x \mid \texttt{var2}\ X \\ \mathit{MacroDef} &::= \texttt{pred}\ m\texttt{(}V\texttt{,}\cdots\texttt{,}V\texttt{)}\ \texttt{=}\ \varphi\texttt{;} \end{aligned}$$

where each of V can be either a first-order or a second-order variable declaration. We also augment the syntax for formulae as follows

$$\begin{aligned} T &::= p \mid S \\ \varphi &::= \ldots \mid m\texttt{(}T\texttt{,}\cdots\texttt{,}T\texttt{)} \end{aligned}$$

where macro call form $m(T_1, \cdots, T_n)$ is expanded to its definition whose parameter variables are replaced with the supplied arguments. Note that macro definitions cannot be recursive. Macros are useful not only for concise description of queries, but also for efficient static processing by separate compilation.

Path Expressions

Our core syntax has only two primitive relations on tree nodes, `firstChild` and `nextSibling`. This reflects our encoding of XML into binary trees described in Section 2.1. Although these two primitives have sufficient expressiveness, it would be convenient if one can directly use relations based on the *unranked* view of the input XML tree. For this purpose, we define four relations by macros as shown in Figure 2.3. Namely, the relation `p_cs(p,C)` asserts that the set of all children of the node `p` (in the unranked view) is `C`, and the relation `p_c(p,c)` asserts that `c` is one of the children

```
pred p_cs( var1 p, var2 C ) =
   all1 c: (c in C <=>
      firstChild(p,c) | ex1 b: (b in C & nextSibling(b,C)));

pred p_c( var1 p, var1 c ) =
   ex2 C: (p_cs(p,C) & c in C);

pred a_ds( var1 a, var2 D ) =
   all1 d: (d in D <=>
      p_c(a,d) | ex1 b: (b in D & p_c(b,d)));

pred a_d( var1 p, var1 d ) =
   ex2 D: (a_ds(a,D) & d in D);
```

Figure 2.3: Representing unranked view in binary MSO

of p. The macros a_ds and a_d define similar relations except that these are based on the ancestor-descendant relationship in the unranked view.

Since those relations are frequently used in XML queries, we specially define a syntax to use those relations for mimicking XPath [6] expressions.

$$D ::= \texttt{/} \mid \texttt{//}$$
$$U ::= x \mid S \mid p\texttt{:}S$$
$$\varphi ::= \ldots \mid UDUD\cdots DU \mid \texttt{/}UD\cdots DU$$

Here, D stands for *path delimiters* and U stands for *path units*. The expression p/q (p//q) means that the node denoted by p is the parent (respectively, an ancestor) of the node denoted by q in the original unranked tree. In other words, p/q is equivalent to p_c(p,q) and p//q is equivalent to a_d(p,q). When applied to second-order terms, such a path expression has an existential meaning. For example, p/<tag> is a shorthand for ex1 x: p_c(p,x) & x in <tag>. When applied to a pair of first- and second-order terms p:S, the expression simultaneously declares that p in S and the term p satisfies the very path expression. For instance, the expression x:<a>/y is equivalent to x in <a> & x/y, and thus also equivalent to x in <a> & p_c(x,y).

When more than two units are connected by path delimiters, it has a conjunctive meaning. For example, x/y/z stands for p_c(x,y) & p_c(y,z). Delimiter-prefixed

```
pred doc_next( var1 p, var1 q ) =
  firstChild(p,q) | nextSibling(p,q)
  | ex1 r: (r//p & nextSibling(r,q));

pred doc_orders( var1 p, var2 Q ) =
  all1 q: (q in Q <=>
    doc_next(p,q) | ex1 b: (b in Q & doc_next(b,q));

pred doc_order( var1 p, var1 q ) =
   ex2 Q: doc_orders(p,Q) & q in Q;
```

Figure 2.4: Document Orders

expressions `/`$U_1 D_1 \dots D_n U_n$ represent *absolute paths* from the root node. When U_1 is a first-order term, the expressions are interpreted as `root=`U_1`&`$U_1 D_2 \dots D_n U_n$. When U_1 is a second-order term, the interpretation is `root:`$U_1 D_2 \dots D_n U_n$.

As an example, the path expression

```
/<a>//<b>/x:<c>/@d
```

is equivalent to the following formula without path expressions:

```
ex1 s: ex1 t: root in <a> & s in <b> & x in <c> & t in @d &
  a_d(root,s) & p_c(s,x) & p_c(x,t)
```

Order Relations

Another frequently used primitive in terms of XML processing is pre-order relations (often called *document order* relations) among tree nodes. We also define this relation as a macro `doc_order` as shown in Figure 2.4 and provide a short-hand syntax as follows:

$$\varphi ::= \dots \mid p < p$$

An expression $p_1 < p_2$ is equivalent to `doc_order`(p_1, p_2).

2.3 Transformation Templates

This section describes the syntax and semantics of MTran, an XML transformation language based on MSO.

2.3.1 Overview

Most of the definitions except for `gather` and `visit` expressions are straightforward. A `gather` expression, as its name shows, *gathers* all nodes in the input tree that satisfies the specified MSO query expression. For example, the template

```
{gather x :: x in <B> :: x}
```

with the input

```
<A>
  <B><C>ddd</C></B>
  <C><B>eee</B></C>
  <B><C><B>fff</B></C></B>
</A>
```

is evaluated to the list of nodes:

```
  <B><C>ddd</C></B>
  <B>eee</B>
  <B><C><B>fff</B></C></B>
  <B>fff</B>
```

Note that `gather` expression gathers all nodes that match the query regardless of their inclusion relations. When we want to obtain only outermost nodes, we need to explicitly specify the condition in the query expression like:

```
{gather x :: x in <B> & ~ex1 y:(y//x & y in <B>) :: x}
```

or more simply:

```
{gather x :: x in <B> & ~<B>//x :: x}
```

This query states "`x` is labeled `B` and none of the ancestors of `x` is labeled `B`," and therefore only the outermost `B` nodes are gathered in this case. A query to retrieve only innermost nodes can also be written similarly.

Semantics of `visit`

A `visit` expression *visits* every node that satisfies the associated query formulae in the subtree specified by the `from` clause (or the whole input tree if the `from` clause is not supplied). Each node that is matched by one of the queries is transformed according to the corresponding template, and the other unmatched nodes are left unchanged. The result of the `visit` expression is the list of trees obtained after the transformation above. This is a rough semantics of our visit expressions. However, there are subtle behaviors in details. To illustrate these, let us consider the following template that encloses each `B` node with a `Mark` tag:

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

Our semantics of `visit` expressions contains two kinds of recursion. We first explain the first one of them, namely, recursion on input trees. The template produces, e.g., from the input

```
<A>
  <B><C>ddd</C></B>
  <C><B>eee</B></C>
</A>
```

the result:

```
<A>
  <Mark><B><C>ddd</C></B></Mark>
  <C><Mark><B>eee</B></Mark></C>
</A>
```

Programmers do not need to write an explicit recursion to search and transform `B` nodes located deeply inside the tree structure. Instead, the recursive behavior is embedded in the semantics.

The second kind of recursion, namely, a recursion on generated trees, appears when we deal with nested elements. All `B` nodes are enclosed by `Mark` tag even if some of the `B`s are nested. When we feed the input document

```
<A>
  <B><C><B>fff</B></C></B>
</A>
```

the result is as follows:

```
<A>
  <Mark><B><C><Mark><B>fff</B></Mark></C></B></Mark>
</A>
```

First, the outer `B` is transformed with the specified template `Mark[x]`, and then we perform another visitation into *the generated tree.* (Note that in the case such recursion is not desired, we can specify the same condition used in the last `gather` example.) This recursion is performed in order to transform other nodes from the original input tree that are embedded in the generated nodes. We don't retransform newly generated nodes, nor the nodes already applied the transformation in the current `visit` expression. MTran is intended to be a language for one-pass transformation, and thus our semantics applies transformations only to the nodes from the original input tree.

In the rest of this section, we formally define the syntax and the semantics of the language.

2.3.2 Syntax

A whole program consists of a list of user-defined macros and a template, where templates are defined by the following syntax.

$$
\begin{aligned}
EL &::= E^* \\
E &::= x \\
&\quad \mid \texttt{"}s\texttt{"} \\
&\quad \mid e\texttt{[}EL\texttt{]} \\
&\quad \mid \texttt{@}a\texttt{[}EL\texttt{]} \\
&\quad \mid \{\texttt{gather } x \texttt{ :: } \varphi \texttt{ :: } EL\} \\
&\quad \mid \{\texttt{visit } x \ (\texttt{::}\ \varphi \texttt{ :: } EL)^*\} \\
&\quad \mid \{\texttt{visit } x \texttt{ from } y \ (\texttt{::}\ \varphi \texttt{ :: } EL)^*\}
\end{aligned}
$$

E stands for *expressions* and *EL* for *expression lists*. A `visit` expression without `from` clause is a syntax sugar for {`visit` *x* `from root ...`}.

2.3.3 Semantics

Internal form

MTran internally handles three different forms of XML representation. One is the normal textual representation as defined in the XML standard [1]. Another is the binary encoding of XML trees described in Section 2.1, which is used in our query algorithm to represent input trees. The last one is the *internal form*, which is a suitable representation for both input and output trees during the evaluation of our transformation templates. The internal trees I (trees in the internal form) is constructed as follows:

$$\begin{aligned}
&I ::= (n, p)\\
&n ::= e\texttt{[}I^*\texttt{]} \mid \texttt{@}a\texttt{[}I^*\texttt{]} \mid \texttt{"}s\texttt{"}\\
&p \in \{0,1\}^* \cup \{\perp\}
\end{aligned}$$

First part of the nodes of the internal form represents an unranked tree structure. Second part maintains the position where the node inhabited in the input tree. In the case of a node that is not from input tree (i.e. when the node is newly constructed according to the template), $\perp$ is assigned. We use this positional information only for the evaluation of `visit` expressions, which has to traverse and search the nodes derived from input trees embedded inside the result of associated templates, as we later explain. For the final output as an XML document, positional information is dropped.

An internal node $e\texttt{[}\ldots\texttt{]}$ corresponds to a standard XML element node $\texttt{<}e\texttt{>}\ldots\texttt{</}e\texttt{>}$, and $\texttt{@}a\texttt{[}\ldots\texttt{]}$ corresponds to an attribute $a\texttt{"}\ldots\texttt{"}$. Note that the above definition may yield an invalid XML, such as repeated attributes `elem[@x[...] @x[...]]` or non-text nodes inside attributes `@x[elem[...]]`. Since MTran currently does not exploit any kind of type checking, such ill-formed XMLs are detected at runtime and result in an error.

Definition 4. A conversion function *conv* from a pair of binary tree t and its node p into the internal form is defined as follows:

$$conv(t,p) = \begin{cases}
(e\texttt{[}conv(t,p.\texttt{l}),\ldots,conv(t,p.\texttt{l}\overbrace{\texttt{r}\cdots\texttt{r}}^{k})\texttt{]},p) & t(p) = e\\
(\texttt{@}a\texttt{[}conv(t,p.\texttt{l})\texttt{]},p) & t(p) = \texttt{@}a\\
(\texttt{"}s\texttt{"},p) & t(p) = \texttt{"}s\texttt{"}
\end{cases}$$

$$\text{where k is the maximum number such that } p.\texttt{l}\overbrace{\texttt{r}\cdots\texttt{r}}^{k} \in Pos(t)$$

Interpretation

An *EL* and E is interpreted under an input binary tree t, and a first-order variable assignment γ, and denotes a list of internal trees.

Now we give the semantics of expressions and expression lists. The interpretation of an expression list $E_1 \cdots E_k$ is simply the concatenation of the interpretations of $E_1, E_2, \cdots$, and E_k:

$$[\![E_1 \ldots, E_k]\!](t,\gamma) \quad = \mathit{concat}\ [\ [\![E_1]\!](t,\gamma) \cdots [\![E_k]\!](t,\gamma)\ ]$$

Here, $[\ldots]$ notation represents lists, and *concat* is the concatenation of all the given lists. The interpretation of expressions are as follows:

$$\begin{aligned}
[\![x]\!](t,\gamma) &= \mathit{conv}(t,\gamma(x)) \\
[\![\texttt{"}s\texttt{"}]\!](t,\gamma) &= [\ (\texttt{"}s\texttt{"},\bot)\ ] \\
[\![e\texttt{[}EL\texttt{]}]\!](t,\gamma) &= [\ (e\texttt{[}[\![EL]\!](t,\gamma)\texttt{]},\bot)\ ] \\
[\![\texttt{@}a\texttt{[}EL\texttt{]}]\!](t,\gamma) &= [\ (\texttt{@}a\texttt{[}[\![EL]\!](t,\gamma)\texttt{]},\bot)\ ] \\
[\![\{\texttt{gather}\ x\texttt{::}\varphi\texttt{::}EL\}]\!](t,\gamma) &= \mathit{concat}\ [\ [\![EL]\!](t,\gamma_{x:=p})\ \big|\ p \in \mathit{Pos}(t), \gamma_{x:=p} \models \varphi\ ] \\
[\![\{\texttt{visit}\ x\ \texttt{from}\ y\texttt{::}\varphi_1\texttt{::}EL_1\texttt{::}\ldots\texttt{::}\varphi_k\texttt{::}EL_k\}]\!](t,\gamma) &= \mathit{vis}(\mathit{Pos}(t), \mathit{conv}(t,\gamma(y)))
\end{aligned}$$

where

$$\mathit{vis}(V,(n,p)) =$$
$$\begin{cases}
\mathit{concat}\ [\mathit{vis}(V\backslash\{p\},I) |\ I \in [\![EL_1]\!](t,\gamma_{x:=p})] & \text{if } p \in V \text{ and } t,\gamma_{x:=p} \models \varphi_1 \\
\mathit{concat}\ [\mathit{vis}(V\backslash\{p\},I) |\ I \in [\![EL_2]\!](t,\gamma_{x:=p})] & \text{if } p \in V \text{ and } t,\gamma_{x:=p} \models \varphi_2 \\
 & \vdots \\
\mathit{concat}\ [\mathit{vis}(V\backslash\{p\},I) |\ I \in [\![EL_k]\!](t,\gamma_{x:=p})] & \text{if } p \in V \text{ and } t,\gamma_{x:=p} \models \varphi_k \\
[(e\texttt{[}\mathit{concat}\ [\mathit{vis}(V,I) |\ I \in IL]\texttt{]},p)] & \text{if } n = e\texttt{[}IL\texttt{]} \\
[(\texttt{@}a\texttt{[}\mathit{concat}\ [\mathit{vis}(V,j) |\ I \in IL]\texttt{]},p)] & \text{if } n = \texttt{@}a\texttt{[}IL\texttt{]} \\
[(\texttt{"}s\texttt{"},p)] & \text{if } n = \texttt{"}s\texttt{"}
\end{cases}$$

More than one condition in the definition of *vis* may be satisfied at a time. In that case, the first case is chosen.

The notation $[f(x)\ |\ x \in \mathit{List}$ (, a condition on x)] is a list comprehension. First the elements in the *List* that fail to satisfy the condition are filtered out, then the function f is applied to each remained elements and yields a result list of the form $[f(x_{i_1}) \cdots f(x_{i_n})]$. Here the set $\mathit{Pos}(t)$ is treated as a list of elements ordered by the document order. All MSO queries in `gather` and `visit` expressions are evaluated under an empty second-order variable assignment (therefore omitted in the above definitions) since we only bind first-order variables in translation templates.

Let us see the formal definition of the *vis* function. It takes two parameters. The first parameter V denotes the set of input nodes that are *not* yet transformed in the current evaluation of the `visit` expression. The second parameter (n, p) denotes the node currently visited. When the node matches one of the query formulae, if the node is from the input tree and still not transformed, then the node is replaced by the list of nodes generated from the associated template. Then again we recursively visit these nodes; recording the current node to be already visited (the second kind of recursion explained in the preceding subsection). When no query formulae match the node, we just recursively go down in the tree (the first kind of recursion).

To see more clearly the subtlety of our semantics, let us consider a slightly modified version of the previous example, which encloses each `B` node with a `B` tag:

```
{visit x :: x in <B> :: B[x]}
```

Consider the input `<B><C><B>hello</B></C></B>`. This input tree is converted to an internal tree by *conv* function:

```
(B[(C[(B[("hello", lll)], ll)], l)], ε)
```

The *vis* function is first applied to this whole internal tree. This time p is ε, which matches the condition `x in <B>` and evaluated to the result of `B[x]` expression, i.e.

```
(B[(B[(C[(B[("hello", lll)], ll)], l)], ε)], ⊥)
```

Then we *re*visit this generated node with the first argument $V \setminus \{\varepsilon\}$. Although current node is labeled `B` again, the value of p is $\perp$, which does not satisfy the condition to apply transformation. Next node is again $p = \varepsilon$, but this time, ε is already removed from V. Therefore we don't transform this node anymore, and go down more deeply. Next node is labeled `C`, which does not satisfy the query expression `x in <B>`. We get:

```
(B[("hello", lll)], ll)
```

This node $p =$ `ll` matches the query and is transformed similarly to the case of $p = \varepsilon$. The final result of the whole `visit` expression successfully becomes:

```
(B[(B[(C[(B[(B[("hello", lll)], ll)], ⊥)], ε)], ⊥)
```

that each `B` node in the input is doubled. Finally the MTran processor should output the result in the standard XML format as follows:

```
<B><B><C><B><B>hello</B></B></C></B></B>
```

Termination

Since the semantics of `visit` expressions is defined in terms of rather complicated recursion, it is not clear whether its evaluation terminates for any input trees. Therefore we give a proof of termination of whole transformation templates for any inputs.

To prove the termination of the *vis* function and `visit` expressions, we first prove the following lemma.

Lemma 1. *In each step of recursion in the vis function, the pair (V, I) of arguments always strictly decreases in the lexicographical order, where V is ordered by set-inclusion and I is ordered by the descendant-ancestor order of internal trees.*

Proof. Case analysis based on the definition of *vis* function. When any one of the φ_j query expression is satisfied, the recursive application of *vis* is done with the first argument $V \setminus \{p\}$, which strictly decreases from V that contains p. When none of the φ_j is satisfied, the recursive application of *vis* is done with the first argument V unchanged and the second argument with the elements of IL, which is the subpart of I. Thus in the arguments (V, I) strictly decrease also in this case. □

Using this result, we can prove the termination of a `visit` expression.

Lemma 2. *Assuming the termination of each subexpression list EL_i, evaluation of a* `visit` *expression under any input tree t and an assignment γ terminates.*

Proof. Since the semantics of a `visit` expression is actually a evaluation of *vis* function. We have to prove the termination of the *vis* function. Since both set-inclusion for finite sets and descendant-ancestor ordering are well-founded relations, by Lemma 1, *vis* function is guaranteed to do well-founded recursion, which terminates in finite steps. Each step of the recursion also terminates since it just evaluates MSO queries (whose deterministic terminating algorithm is given in the next chapter), evaluates subexpression list EL_i (that is assured to terminated by the assumption), and concatenates several lists. Combining these results yields the termination of the whole *vis* function. □

Now we finally prove the termination of any transformation templates.

Prop 1. *Evaluation of E and EL always terminates.*

Proof. By simultaneous induction on the structure of E and EL. For EL, since evaluation of each E_i terminates by the induction hypothesis, evaluation of the whole expression also terminates. For E, we need to prove six cases. The case of variable expressions `x` resolves to the termination of *conv* function, which is clear from the definition of the function. The case of `"s"` expression is trivial. The cases of $e[EL]$, `@`$a[EL]$, and `gather` expressions are straightly derived from the induciton hypothesis. The case of `visit` expressions is already proven by Lemma 2. $\square$

Chapter 3

Evaluation Algorithm

We describe our evaluation strategy for the MTran language in this chapter. In this, efficient evaluation of MSO query expressions is particularly a difficult problem and explained in detail. Our MSO query evaluation consists of usual two steps as shown in Figure 3.1 : (1) compilation of MSO formulae to tree automata and (2) evaluation of n-ary queries represented by tree automata. In this chapter, we first review known facts on tree automata in Section 3.1, and then explain about these two steps in Sections 3.2 and 3.3. Finally in Section 3.4, we explain how to integrate the query algorithm into our evaluation strategy for whole translation templates.

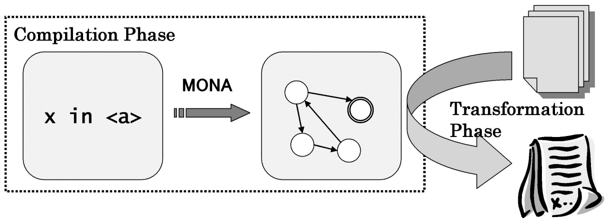


Figure 3.1: Execution Model

3.1 Preliminaries

3.1.1 Queries over trees

First we formalize the concept of queries.

Definition 5. An n-ary query for binary tree over Σ is a function q that maps each tree t to a set of n-tuples A of its positions.

A *tree language over* Σ is a set of trees. A query can also be defined in terms of tree languages.

Definition 6. Let $\mathbb{B} = \{0, 1\}$. An n-ary query defined by a tree language L over $\Sigma \times \mathbb{B}^n$ is a function q such that

$$\begin{aligned} q(t) = \{(v_1, \ldots, v_n) \in \mathit{Pos}(t)^n \mid \\ \exists \beta_1, \ldots, \beta_n : \mathit{Pos}(t) \to \mathbb{B} \qquad \forall i. \forall v \in \mathit{Pos}(t).(\beta_i(v) = 1 \iff v = v_i) \\ \& \; t \times \beta_1 \times \cdots \times \beta_n \in L\} \end{aligned}$$

where the product $t \times s$ of trees is a function defined as $(t \times s)(v) = (t(v), s(v))$.

Intuitively, each β_i in the definition above represents selection marks corresponding to the i-th members of tuples. That is, a query defined by a language L selects a tuple $(v_1, \ldots, v_n)$ on a tree t if and only if L contains a tree where $\beta_1, \ldots, \beta_n$ respectively mark the elements $v_1, \ldots, v_n$ as 1.

Note that we only consider selection marks that select exactly one node v_i in an input tree. In general, a tree language over $\Sigma \times \mathbb{B}^n$ may contain a tree that has no marked nodes, or more than one marked nodes. But such a language defines exactly the same query as the language with all ill-marked trees removed.

3.1.2 Tree Automata

An important class of tree languages and queries is defined in terms of tree automata.

Definition 7. Let Σ be an alphabet. A *bottom-up deterministic tree automaton over* Σ is a tuple $(\Sigma, Q, \delta, q_0, F)$ where Q is a set of states, $\delta : Q \times Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of accepting states.

Definition 8. A *run* of a bottom-up deterministic tree automaton on a binary tree t is a mapping $\rho : \mathit{Pos}(t) \to Q$ such that $\rho(v) = \delta(\rho(v.\mathtt{l}), \rho(v.\mathtt{r}), t(v))$ for each node $v \in \mathit{Pos}(t)$. When $v.\mathtt{l}$ or $v.\mathtt{r}$ does not belong to the domain $\mathit{Pos}(t)$, we use q_0 instead of $\rho(v.\mathtt{l})$ or $\rho(v.\mathtt{r})$. The run ρ is called *accepting* if $\rho(\varepsilon) \in F$. Note that a run is uniquely determined from any tree t. In this case, we say that the tree is *accepting*.

Each tree automaton defines a tree language, which contains all trees accepted by the automaton. Thus, we can regard a tree automaton over $\Sigma \times \mathbb{B}^n$ as an n-ary query over Σ.

Definition 9. An n-ary query over Σ is *regular* if there exists a bottom-up deterministic tree automaton that defines the query.

3.2 From MSO to Tree Automata

An MSO formula with n free first-order variables can naturally be seen as an n-ary query. A formula $\varphi(\vec{x})$ whose free variables are $\vec{x} = (x_1, \ldots, x_n)$ defines a query $q(t) = \{\vec{v} \in Pos(t)^n \mid t \vDash \varphi(\vec{v})\}$. It is well-known that there is an exact correspondence between MSO and tree automata.

Theorem 1. *[14, 24] For every MSO formula $\varphi(\vec{x})$ with n free variables, there exists a bottom-up deterministic tree automaton A over $\Sigma \times \mathbb{B}^n$ that defines the equivalent query. Also for every bottom-up deterministic tree automaton A over $\Sigma \times \mathbb{B}^n$, there exists an equivalent MSO formula with n free variables.*

We evaluate n-ary MSO queries using this equivalence. That is, we first compile every given MSO formula into an equivalent automaton. We exploit the MONA system [18] for this compilation. MONA is a remarkably efficient implementation and is experimentally shown to work in practice for large formulae, even though this compilation step is well known to take a non-elementary time in the worst case. We make some experiments in Section 4.2.

One note must be added here about the alphabets Σ. A tree automaton is defined on top of some fixed alphabet Σ and its run and acceptance is defined only on binary trees over the Σ. Our algorithm, however, takes encoded XML documents as input trees, which may have arbitrary set of labels. To absorb this difference, we always add an extra symbol `others` to Σ of each automaton, and defines the compilation of MSO formulae, the run, and the acceptance of the automaton over the alphabet $\Sigma \cup \{\texttt{others}\}$. Any label on input trees that does not belong to Σ is uniformly treated as an `others` for these automata.

3.3 N-ary Query Algorithm

In this section, we explain our novel algorithm for n-ary queries. First, in Section 3.3.1, we introduce an algorithm that is essentially no different from the binary query algorithm proposed by Berlea and Seidl, and formalize its extension to general n-ary queries. Section 3.3.2 explains our improvements to the algorithm.

3.3.1 Basic Algorithm

Definition 6 yields a naive evaluation algorithm for n-ary queries represented by tree automata. That is, for every n-tuples of nodes of a given input tree t, generate the corresponding selection mark β_i's as in the definition and calculate the bottom-up run of the automaton. If the run is accepting, the tuple belongs to the result set of the query. There can be $|t|^n$ n-tuples where $|t|$ is the size of the input tree, and each run of a tree automaton takes $O(|t|)$ time. So the total time complexity of this naive algorithm is $O(|t|^{n+1})$.

This high time complexity can be improved by sharing intermediate results among calculations of the runs corresponding to the n-tuples in the following way. Our n-ary query algorithm assigns a set $m(v, q)$ of n-tuples of nodes to each pair of a node v of t and a state q of automaton.

Intuitively, $m(v, q)$ is the set of tuples such that, if the tree is marked according to the tuple, then the bottom-up run on the tree reaches the state q at the node v. We call such m a *marking run* of an automaton. Let us consider an example of a ternary query. If it is the case that $(\mathtt{rl}, \mathtt{r}, \bot) \in m(\mathtt{r}, q_1)$, this means that the automaton reaches the state q_1 at the node $\mathtt{r}$, when the node $\mathtt{rl}$ is selected as the first element, the node $\mathtt{r}$ is selected as the second element, and no node in the subtree rooted at $\mathtt{r}$ is selected as the third element. The $\bot$ element represents the last condition.

A marking run can be calculated in a bottom-up fashion. Each $m(v, q)$ is calculated from that of the children of v, as the union of all products of $m(v.\mathtt{l}, q_L)$ and $m(v.\mathtt{r}, q_R)$ where q_L and q_R are the states transitable to the state q by appropriate markings.

Formally, we define marking runs as follows. For the sake of efficient calculation, we define each $m(v, q)$ as a disjoint union of sets $m_s(v, q)$, where $s \in \mathbb{B}^n$. A sequence of bits s represents non-bottom elements of the tuples in $m_s(v, q)$. That is, if $(u_1, \ldots, u_n) \in m_{s_1 \ldots s_n}(v, q)$, then $u_i \neq \bot$ if and only if $s_i = 1$ for every i.

Definition 10. A *marking run* of a deterministic bottom-up tree automaton $(\Sigma \times \mathbb{B}^n, Q, \delta, q_0, F)$ on a tree t is a set of functions $m = \{m_s \mid s \in \mathbb{B}^n\}$. Each m_s is a function of type $\{\mathtt{l}, \mathtt{r}\}^* \times Q \to 2^{\{Pos(t) \cup \{\bot\}\}^n}$ defined as follows. When $v \notin Pos(t)$,

$$\begin{aligned} m_{0\cdots0}(v, q_0) &= \{(\bot, \ldots, \bot)\} \\ m_s(v, q) &= \emptyset \qquad\qquad \text{if } s \neq 0\cdots0 \text{ or } q \neq q_0 \end{aligned}$$

and when $v \in Pos(t)$, it is recursively defined as

$$\begin{aligned} m_s(v, q) = \bigcup \big\{ m_l(v.\mathtt{l}, q_L) * m_r(v.\mathtt{r}, q_R) * \{sing(v, c)\} \bigm| & \; q_L, q_R \in Q, \quad l, r, c \in \mathbb{B}^n, \\ & \; \delta(q_L, q_R, (t(v), c)) = q, \\ & \; (l, r, c) \in sub(s) \big\} \end{aligned}$$

The tuple $sing(v, c)$ is defined as a tuple $(u_1, \ldots, u_n)$ such that $u_i = v$ if $c_i = 1$ and $u_i = \bot$ if $c_i = 0$. For each $s \in \mathbb{B}^n$, we define $sub(s) \subseteq (\mathbb{B}^n)^3$ by the set of splitting (l, r, c) of s such that $\forall i.(s_i = 1 \iff$ exactly one of l_i, r_i, and c_i is 1). The asterisk product $*$ is defined as follows:

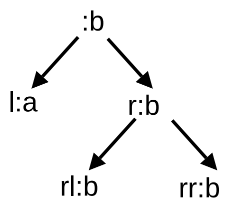
$$\begin{aligned} S * T = \{(u_1, \ldots, u_n) \mid\ & (s_1, \ldots, s_n) \in S, (t_1, \ldots, t_n) \in T, \\ & (u_i = s_i \ \&\ \bot = t_i) \text{ or } (\bot = s_i \ \&\ u_i = t_i)\} \end{aligned}$$

Using a marking run, we can obtain the query result as:

$$\bigcup_{q \in F} m_{1\ldots1}(\varepsilon, q)$$

Example

Before we discuss the evaluation algorithm, let us clarify a little complicated definition of marking runs by example. Consider the following tree t over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ defined as



(for the sake of clarity, we present each node by a pair `n:L` where `n` is its position from $\{\mathtt{l}, \mathtt{r}\}^*$ and `L` is its label from Σ) and the automaton $(\Sigma \times \mathbb{B}^2, Q, \delta, q_0, F)$ with the set of states $Q = \{q_0, q_1, q_2, q_3\}$, the accepting states $F = \{q_2\}$, and the transition function δ defined as follows:

$$\begin{aligned} \delta(q_0, q_0, (s, 00)) &= q_0 && \text{for any } s \in \Sigma \\ \delta(q_0, q_0, (\mathtt{b}, 01)) &= q_1 \\ \delta(q_0, q_1, (\mathtt{b}, 10)) &= q_2 \\ \delta(q_0, q_2, (s, 00)) = \delta(q_2, q_0, (s, 00)) &= q_2 && \text{for any } q_i \in Q,\ s \in \Sigma \\ \delta(q_i, q_j, x) &= q_3 && \text{for any other cases} \end{aligned}$$

This automaton defines a binary query that selects all pairs of a node and its right child both labeled `b`, i.e. $(\varepsilon, \mathtt{r})$ and $(\mathtt{r}, \mathtt{rr})$ for the above tree. Intuitively, q_0 represents the starting state, and q_1 represents the state where the automaton finds a `b` node as the second element. The accepting state q_2 means that the automaton finds a desired pair. When the automaton recognized that the marked pair is not an answer for the query, it goes to the *junk* state q_3. When the input tree is ill-marked, that is, when more than one nodes are selected for the same side of the pair, the automaton also moves to q_3. For simplicity of explanation, we here introduce the junk state q_3 to construct an automaton that rejects all ill-marked trees. Note that we do not always have to introduce such a state for ill-marked trees, since the query defined by an automaton is not affected by whether or not the automaton accepts ill-marked trees, as we already stated.

Marking runs on automata representing binary queries consists of four functions m_{00}, m_{01}, m_{10}, and m_{11} of the type $\{\mathtt{l},\mathtt{r}\}^* \times Q \to 2^{\{Pos(t)\cup\{\perp\}\}^2}$. We first calculate those functions on leaf nodes $\{\mathtt{l},\mathtt{rl},\mathtt{rr}\}$. Since the set $m_{00}(p, q_0)$ is defined to be $\{(\perp,\perp)\}$ for $p \notin Pos(t)$ and $m_s(p,q) = \emptyset$ for other s, q, we only need to consider the case $q_L = q_0$ and $q_R = q_0$ in the Definition 10. Thus, for each leaf node e we have:

$$m_{00}(e,q) = \{(\perp,\perp) \mid \delta(q_0,q_0,(t(e),00)) = q\}$$
$$m_{01}(e,q) = \{(\perp,e) \mid \delta(q_0,q_0,(t(e),01)) = q\}$$
$$m_{10}(e,q) = \{(e,\perp) \mid \delta(q_0,q_0,(t(e),10)) = q\}$$
$$m_{11}(e,q) = \{(e,e) \mid \delta(q_0,q_0,(t(e),11)) = q\}$$

The concrete values of m_s is shown in the following tables:

Table 3.1: $m_{00}(v,q)$

	q_0	q_1	q_2	q_3
`l`:a	$\{(\perp,\perp)\}$	$\emptyset$	$\emptyset$	$\emptyset$
`rl`:b	$\{(\perp,\perp)\}$	$\emptyset$	$\emptyset$	$\emptyset$
`rr`:b	$\{(\perp,\perp)\}$	$\emptyset$	$\emptyset$	$\emptyset$

Table 3.2: $m_{01}(v,q)$

	q_0	q_1	q_2	q_3
`l`:a	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\perp,\mathtt{l})\}$
`rl`:b	$\emptyset$	$\{(\perp,\mathtt{rl})\}$	$\emptyset$	$\emptyset$
`rr`:b	$\emptyset$	$\{(\perp,\mathtt{rr})\}$	$\emptyset$	$\emptyset$

Table 3.3: $m_{10}(v,q)$

	q_0	q_1	q_2	q_3
`l`:a	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\mathtt{l},\perp)\}$
`rl`:b	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\mathtt{rl},\perp)\}$
`rr`:b	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\mathtt{rr},\perp)\}$

Table 3.4: $m_{11}(v,q)$

	q_0	q_1	q_2	q_3
`l`:a	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\mathtt{l},\mathtt{l})\}$
`rl`:b	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\mathtt{rl},\mathtt{rl})\}$
`rr`:b	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\mathtt{rr},\mathtt{rr})\}$

For example, the entry $m_{01}(\mathtt{rr}, q_1) = \{(\perp, \mathtt{rr})\}$ means that if the first bit is not marked for any nodes and the second bit of the node `rr` is marked, then the automaton reaches state q_1 at the node `rr`.

Next let us calculate m for the node `r:b`, which can be constructed from m for `rl:b` and `rr:b`. We demonstrate the calculation of the two cases that are actually required for obtaining the result of the query. The first case is $m_{11}(\mathtt{r}, q_2)$:

$$\begin{aligned} m_{11}(\mathtt{r}, q_2) = \bigcup\{ & m_l(\mathtt{rl}, q_L) * m_r(\mathtt{rr}, q_R) * \{sing(\mathtt{r}, c)\} \mid \\ & \delta(q_L, q_R, (\mathtt{b}, c)) = q_2, \quad (l, r, c) = sub(11)\} \end{aligned}$$

Seeing the definition of the transition function δ, there are three choice of q_L, q_R, and c to satisfy the condition $\delta(q_L, q_R, (\mathtt{b}, c)) = q_2$. Namely, the case $\{q_L = q_0, q_R = q_2, c = 00\}$, the case $\{q_L = q_2, q_R = q_0, c = 00\}$, and the case $\{q_L = q_0, q_R = q_1, c = 10\}$. Only the last case is significant, since $m_l(\mathtt{rl}, q_2)$ and $m_r(\mathtt{rr}, q_2)$ are empty. Thus, we have:

$$\begin{aligned} m_{11}(\mathtt{r}, q_2) &= \bigcup\{m_l(\mathtt{rl}, q_0) * m_r(\mathtt{rr}, q_1) * \{(\mathtt{r}, \perp)\} \mid (l, r, 10) = sub(11)\} \\ &= m_{00}(\mathtt{rl}, q_0) * m_{01}(\mathtt{rr}, q_1) * \{(\mathtt{r}, \perp)\} \cup m_{01}(\mathtt{rl}, q_0) * m_{00}(\mathtt{rr}, q_1) * \{(\mathtt{r}, \perp)\} \\ &= \{(\perp, \perp)\} * \{(\perp, \mathtt{rr})\} * \{(\mathtt{r}, \perp)\} \cup \emptyset * \{(\perp, \perp)\} * \{(\mathtt{r}, \perp)\} \\ &= \{(\mathtt{r}, \mathtt{rr})\} \end{aligned}$$

The other interesting case is $m_{01}(\mathtt{r}, q_1)$. There is only one choice of q_L, q_R, and c in this case:

$$\begin{aligned} m_{01}(\mathtt{r}, q_1) &= m_{00}(\mathtt{rl}, q_0) * m_{00}(\mathtt{rr}, q_0) * \{(\perp, \mathtt{r})\} \\ &= \{(\perp, \mathtt{r})\} \end{aligned}$$

Finally, we show the calculation of $m_{11}(\varepsilon, q_2)$, which is the result of the query. As well as in the case of $m_{11}(\mathtt{r}, q_2)$, we have three choice of q_L, q_R, and c. This time, two cases are significant.

$$\begin{aligned} m_{11}(\varepsilon, q_2) = \quad & \bigcup\{m_l(\mathtt{l}, q_0) * m_r(\mathtt{r}, q_1) * \{(\varepsilon, \perp)\} \mid (l, r, 10) = sub(11)\} \\ & \cup \bigcup\{m_l(\mathtt{l}, q_0) * m_r(\mathtt{r}, q_2) * \{(\perp, \perp)\} \mid (l, r, 00) = sub(11)\} \\ = \quad & m_{00}(\mathtt{l}, q_0) * m_{01}(\mathtt{r}, q_1) * \{(\varepsilon, \perp)\} \cup m_{01}(\mathtt{l}, q_0) * m_{00}(\mathtt{r}, q_1) * \{(\varepsilon, \perp)\} \\ & \cup m_{00}(\mathtt{l}, q_0) * m_{11}(\mathtt{r}, q_2) * \{(\perp, \perp)\} \\ = \quad & \{(\perp, \perp)\} * \{(\perp, \mathtt{r})\} * \{(\varepsilon, \perp)\} \cup \emptyset * \{(\perp, \perp)\} * \{(\varepsilon, \perp)\} \\ & \cup \{(\perp, \perp)\} * \{(\mathtt{r}, \mathtt{rr})\} * \{(\perp, \perp)\} \\ = \; & \{(\varepsilon, \mathtt{r}),\ (\mathtt{r}, \mathtt{rr})\} \end{aligned}$$

Complexity

Naive bottom-up calculation of m still takes $O(|t|^{n+1})$ time, since we need to calculate $m_{1\dots1}$ that grows up to $O(|t|^n)$ in the worst case for all $|t|$ nodes.

Berlea and Seidl has proposed an algorithm for binary queries in [17], based on the bottom-up calculation of m. They used their formalism of regular queries based on push-down forest automata, but the same approach is applicable also for simple tree automata. They took two-phase approach to avoid operating on large m_{11} sets. During the first phase, only m_{00}, m_{01}, and m_{10} are calculated as in Definition 10. In the second phase, they collect the elements of $\bigcup_{q\in F} m_{11}(\varepsilon, q)$ from each node through one extra top-down run. This is possible since $m_{11}(v, Q_c) \equiv \bigcup_{q\in Q_c} m_{11}(v, q)$ is definable in terms of $m_{11}(v, \mathtt{l}, Q_1)$ and $m_{11}(v, \mathtt{r}, Q_r)$ for appropriate Q_1 and Q_2 in the following way:

$$\begin{aligned}
m_{11}(v, Q_c) =& \{m_{00}(v.\mathtt{l}, q_L) * m_{11}(v.\mathtt{r}, q_R) * \{(\bot,\bot)\} \mid \delta(q_L, q_R, (t(v), 00)) \in Q_c\} \\
&\cup \{m_{11}(v.\mathtt{l}, q_L) * m_{00}(v.\mathtt{r}, q_R) * \{(\bot,\bot)\} \mid \delta(q_L, q_R, (t(v), 00)) \in Q_c\} \\
&\cup \text{ (other terms that do not contain the } m_{11} \text{ of child nodes) ...} \\
=& \, m_{11}(v.\mathtt{r}, \{q_R \mid \exists q_L. m_{00}(v.\mathtt{l}, q_L) \neq \emptyset, \delta(q_L, q_R, (t(v), 00)) \in Q_c\}) \\
&\cup m_{11}(v.\mathtt{l}, \{q_L \mid \exists q_R. m_{00}(v.\mathtt{r}, q_R) \neq \emptyset, \delta(q_L, q_R, (t(v), 00)) \in Q_c\}) \\
&\cup \text{ (others...)}
\end{aligned}$$

They have concluded that their algorithm runs in $O(rt + s)$ time where t is the size of input documents and r is the maximum size of m_{10}'s and m_{01}'s – potential matches – and s is the number of the result tuples. Union operation in the definition of m_{10} and m_{01} takes the time proportional to the size of operand sets (r), and the calculation is repeated t times. The size r of potential matches may be smaller compared to t in practical cases, but it is $O(t)$ in the worst case. Therefore the total time complexity is $O(t^2)$. As Berlea et al. mentioned (but did not formalize), their algorithm can be generalized to n-ary queries, and the complexity is $O(t^n)$ time. This has been the best known result for the n-ary MSO query problem.

We have improved this result by optimizing the intermediate set operations and achieved $O(t + s)$ time complexity for n-ary queries. We give the detail in next subsection.

3.3.2 Our Algorithm

Our strategy is rather straightforward. We calculate the whole marking run including $m_{1\ldots1}$ directly following Definition 10, using *lazy set operations.*

As we have seen in the previous example, not all intermediate sets are necessary for query outputs: only the sets associated with a node and a state that can finally lead to acceptance at the root node are useful. In the previous example, the marking run m consists of 100 sets – the product of 4 kinds of bits, 5 nodes, and 4 states – but few of them are actually required to calculate the result of the query. If we could know whether the value of each $m_s(v, q)$ is required for the final result, and avoid unnecessary calculations, it would improve the efficiency.

Our algorithm delays set operations like unions and products for non-empty sets until actually enumerating the final result, yet still eagerly does calculations on empty sets ($\emptyset$) and *unit* sets (a singleton set of the tuple whose all elements are the bottom: $\{(\perp, \cdots, \perp)\}$). We show that by this approach the time complexity of n-ary query reduces to $O(t + s)$, linear in the size of the input and the output.

Partially Lazy Set Operations

We first define lazy set operations required to calculate marking runs. Here is our implementation of the set data type and operations over it in Objective Caml [25]. Only two operations, union and product, are necessary.

```
type node_set_ne
  = Singleton of (node option) list
  | Union     of node_set_ne * node_set_ne
  | Product   of node_set_ne * node_set_ne
```

The type of non-empty sets. The `node option` type is a type that can be either `None` ($\perp$) or `Some x` where `x` can be any value of type `node`. Elements of the sets are length-n lists of `node options`. Unions and products are represented as an *operation tree* that symbolically represents a set. When it comes to the point where the actual members of the set are required, the operation tree is evaluated to an explicit form.

```
type node_set
  = Empty
  | UnitSet
  | NonEmpty of node_set_ne
```

The type of sets of nodes. To deal with emptiness, empty sets and non-empty sets are strictly distinguished.

```
let union a b = match (a,b) with
  | Empty,s   | s,Empty      -> s
  | UnitSet,s | s,UnitSet    -> UnitSet
  | NonEmpty s1, NonEmpty s2 -> NonEmpty (Union s1 s2)

let product a b = match (a,b) with
  | Empty,s   | s,Empty      -> Empty
  | UnitSet,s | s,UnitSet    -> s
  | NonEmpty s1, NonEmpty s2 -> NonEmpty (Product s1 s2)
```

Union and product operation. An empty set and a unit set is dealt as a special case. That is, when one of the arguments is an empty set (a unit set, respectively), the set operation is explicitly performed at that point. This is to avoid delayed unnecessary computations of the left operand of the form like `Product(LargeSet,` $\emptyset$`)` (`Product(LargeSet,` $\{\perp, \ldots, \perp)\}$`)`). These special treatments are essential for linear time complexity. Note that each definition of union and product contains one pattern matching and two applications of data constructors, therefore either operation can be done in constant time.

We next show the function to convert an operation tree to the actual list of elements that belong to the set represented by the operations tree.

```
(* 'a option list -> 'a option list -> 'a option list *)
let prod_one a b =
  match (a, b) with
    | (None::al,    b::bl) -> b :: prod_one al bl
    | (   a::al, None::bl) -> a :: prod_one al bl
    | _ -> raise "should not happen"

(* 'a option list list -> 'a option list list
                             -> 'a option list list *)
let prod sa sb acc =
  fold_right (fun a acc ->
    fold_right (fun b acc -> (prod_one a b)::acc) sb acc) sa acc
```

The function `prod` is to calculate the asterisk product exactly as in Definition 10 and append them to the list `acc`.

```
(* 'a node_set -> 'a list list *)
let to_list = function
  | Empty      -> []
  | UnitSet    -> raise "should not happen"
  | NonEmpty s -> let rec f s acc = function
      | Singleton vs -> vs::acc
      | Union    a b -> f a (f b acc)
      | Product  a b -> prod (f a []) (f b []) acc
    in map (map (fun (Some e)->e)) (f s [])
```

The function `to_list` is the main routine, which converts a symbolic operation tree to an actual set of tuples. Since this function is assumed to be applied to the set that represents the result of the query, it does not consider the case of `UnitSet`, which means that no node is selected. For `Singleton`, we just create a singleton list. For `Product`, we use the `prod` function that is already defined. For `Union` operation, just concatenating two lists will work fine. To see this, let us review the Definition 10. Our query algorithm always takes the union of two sets in the form

$$\bigcup_{l,r,c} m_l(v.\texttt{l}, q_L) * m_r(v.\texttt{r}, q_R) * \{(...)\}$$

where $m_l(v.\texttt{l}, q_L)$ consists of tuples whose all elements belong to the subtree rooted at $v.\mathtt{l}$, and $m_r(v.\mathtt{r}, q_R)$ consists of tuples in the subtree rooted at $v.\mathtt{r}$. This means that the ith element of a tuple in the set $m_l(v.\mathtt{l}, q_L) * m_r(v.\mathtt{r}, q_R)$ belongs to the subtree of $v.\mathtt{l}$ *if and only if* the ith bit of l is 1. Thus, if $l \neq l'$ or $r \neq r'$, the set $m_l(v.\mathtt{l}, q_L) * m_r(v.\mathtt{r}, q_R)$ and the set $m_{l'}(v.\mathtt{l}, q_L) * m_{r'}(v.\mathtt{r}, q_R)$ are always disjoint. This is why simple concatenation works well.

Complexity

Using these lazy set operations, we can calculate all operation trees corresponding to m_s including $m_{1\dots1}$, by simply following Definition 10. As shown in the definition, all m_s for each node v can be constructed as the union of the set of $m_l(\dots) * m_r(\dots) * \{v, \dots, v\}$. Since there are $|Q|^2$ choices for q_L and q_R, and at most 3^n choices for (l, r, c), using lazy set operations, we can construct the operation trees corresponding to m_s in $O(3^n|Q|^2)$ steps. Thus, to calculate marking runs for all nodes in the tree takes $O(3^n|Q|^2 t)$ time, which is linear to the size of the input tree.

To show the total complexity of query evaluation is linear, we next show that the evaluation of operation trees to obtain the list of elements in the denoted set will take only linear time with respect to the size of the output list. The following lemma assures that the `to_list` function can be evaluated in linear time to the size of the set denoted by the argument.

Lemma 3. *Let $|s|$ be the length of the resulting list of operation tree s. The inner function `f s acc` in the definition of `to_list` can be evaluated in $O(3^k n|s|)$ time where n is the arity of the query, and k is the maximum number of `Product` nodes in each path from the root to a leaf node of the operation tree `s`.*

The maximum number of `Product` nodes on each path are limited to n in our algorithm, since we only generate `Product` nodes between non-empty non-unit sets. Such products strictly decrease the number of $\perp$ in each tuple, and the tuples we consider here are always n-tuples. Therefore we can derive $O(3^n n|s|)$ time from this lemma. The whole query computation ends in $O(3^n(|Q|^2 t + n|s|))$ time. When the query formula is fixed, the values 3^n, $|Q|^2$, and n are just constant factors, so we obtain the complexity $O(t + |s|)$. The proof of Lemma 3 can be done by simple induction on the structure of the arguments of type `node_set_ne`:

Proof. The `Singleton` case is trivial. For the `Union` case, by induction hypothesis, `(f b acc)` is evaluated in $O(3^k n|b|)$ time and then `(f a ...)` is evaluated in $O(3^k n|a|)$ time. So totally $O(3^k n|a| + 3^k n|b|) = O(3^k n|s|)$ time is consumed. For the `Product` case, by induction hypothesis, `(f a [])` is evaluated in $O(3^{k-1}n|a|)$ time and then `(f b [])` is evaluated in $O(3^{k-1}n|b|)$ time. Since the `prod` operation is $O(n|a||b|)$, total complexity is $O(3^{k-1}n|a| + 3^{k-1}n|b| + n|a||b|) < O(3^k n|a||b|) = O(3^k n|s|)$ $\square$

Example

Running our algorithm on the previous example in the explanation of basic algorithm, we obtain $m_{11}(\varepsilon, q_2)$ as the following operation tree

```
NonEmpty(
  Union(
    Product( Singleton [⊥;r],  Singleton [ε;⊥] ),
    Product( Singleton [⊥;rr], Singleton [r;⊥] )
  ))
```

instead of a concrete set $\{(\varepsilon, \mathtt{r}),\ (\mathtt{r}, \mathtt{rr})\}$. By evaluating this tree using the `to_list` function, we get the desired result: `[[r; rr]; [ε; r]]`.

Two features of the algorithm has eliminated unnecessary calculation. Th first point is its laziness. By delaying the actual construction of the sets, we only need to calculate the sets contained in the query result $m_{11}(\varepsilon, q_2)$.

The second point is its eager evaluation for empty and unit sets. As a consequence of this strategy, resulting `NonEmpty` trees never contain subtrees that represent empty sets. Any calculations of sets that are taken a product between an empty set are skipped in this algorithm.

3.4 Transformation Template Evaluation

Evaluation of `gather` or `visit` expressions themselves are clear from the definition of semantics. For `gather` expressions, we first obtain the set of nodes that satisfied the specified query expression. Then for each node in the result, we bind the node to the specified variable, and recursively evaluate the associated subtemplates. All resulting trees from subtemplates are concatenated to form the result of whole `gather` expression. For `visit` expressions, we first obtain the query result and for each node in the input we record information whether the node matched each query. Then we recursively transform the specified subtree as the *vis* function in the semantics. The issue is, how to integrate the query algorithm introduced in Section 3.3 into the evaluation strategy for whole transformation templates.

3.4.1 Efficient N-ary Queries Using Context Information

Naive implementation of the semantics in Section 2.3.3 will evaluate each query as an unary query, by fixing the other variables than the one to be queried in outer templates. However, as we already discussed in Section 1.2.3, we do not take this strategy. Instead, we evaluate each query as an n-ary queries, where n is the number of free variables of the query expression. Consider the following case:

{gather x :: ϕ(x) :: {gather y :: ψ(x,y) :: ...}}

We evaluate only once the expression $\psi(\mathtt{x}, \mathtt{y})$ as a binary query, and obtain a list of 2-tuples $[(v_1^{\mathtt{x}}, v_1^{\mathtt{y}}) \ldots (v_s^{\mathtt{x}}, v_s^{\mathtt{y}})]$ that each tuple satisfies the expression. This list is referred as an table to obtain the list of all ys for each x bound by the outer template. In this way, each query is evaluated only once per one input document. Thus, the total time consumed for queries will be the sum of the time of each query.

This strategy, however, still has some waste. Obviously, we only need the lists of $(v_i^{\mathtt{x}}, v_i^{\mathtt{y}})$ such that the node $v_i^{\mathtt{x}}$ satisfies the query expression $\phi(\mathtt{x})$. Not all results of $\psi(\mathtt{x}, \mathtt{y})$ are necessary. This could be a problem, since as we proved in the preceding subsection, each query takes $O(3^n(|Q|^2 t + ns))$ time that depends on the size of query result s.

One way to handle this problem is to rewrite the query expression as follows:

{gather x :: ϕ(x) :: {gather y :: ϕ(x) & ψ(x,y) :: ...}}

Pushing outer query expressions into inner queries and concatenating them conjunctively. This rewriting keeps the result of transformation unchanged, and reduces the size of result from inner binary queries. Only the pairs $(v_i^{\mathtt{x}}, v_i^{\mathtt{y}})$ satisfying $\phi(v_i^{\mathtt{x}})$ is obtained. Although this rewriting solution indeed reduces the size of s, it causes another complexity problem. Due to the complication of the query formula from ψ(x,y) to ϕ(x) & ψ(x,y), the size $|Q|$ of compiled automaton may increase. The parameter $|Q|$ affects the total time complexity in quadratic order and cannot be ignored.

We developed a slightly modified version of n-ary query algorithm to remedy this problem. We now are able to reduces the query result as same as the rewriting approach while keeping the size of automata unchanged, by using the result of outer queries (namely, context information) during the query evaluation of inner queries.

We first execute the outermost unary query ϕ(x) in the same way as explained in the previous subsection, and obtain the list of results $R_{\mathtt{x}} = [u_1^{\mathtt{x}} \ldots u_k^{\mathtt{x}}]$. We next execute the inner query ψ(x,y) with one modification to the definition of a marking run:

$$\begin{aligned} m_s(v,q) = \bigcup \{ m_l(v.\mathtt{l}, q_L) {*} m_r(v.\mathtt{r}, q_R) * \{sing(v,c)\} \mid & \\ & q_L, q_R \in Q, \quad l, r, c \in \mathbb{B}^n, \\ & \delta(q_L, q_R, (t(v), c)) = q, \\ & (l, r, c) \in sub(s), \\ & \underline{\forall i. var[i] \neq \mathtt{y} \& c_i = 1 \implies v \in R_{var[i]}} \} \end{aligned}$$

where $var[i]$ is the variable that corresponds to the i-th selection mark of an automaton. In this modified definition, we select each node v as a marked node for variable $\mathtt{x}$ only when v is contained in $R_{\mathtt{x}}$. Final result obtained in this way thus only contains tuples $(v_i^{\mathtt{x}}, v_i^{\mathtt{y}})$ such that $v_i^{\mathtt{x}} \in R_{\mathtt{x}}$. Checking the extra condition (underlined part) does not incur any additional time complexity blowup, since by processing outer queries earlier than inner ones, we can always have each $R_{var[i]}$ already constructed and its inclusion can be determined in constant time using any appropriate data structure such as hash-tables.

Chapter 4

Experiments

In this chapter, we conduct several experiments of MTran language. In Section 4.1, we give experiments on expressiveness by writing several transformation templates in MTran. In Section 4.2, we measure the execution time of several examples in our system and confirm that it yields an enough performance.

4.1 Examples

In this section, we give several example templates written in MTran and demonstrate its expressiveness. The first example is a template to generate the table of contents from a given input XHTML document. The second example is a template using linguistic queries. The third example is a simplification of RELAX NG schemas.

4.1.1 XHTML Table of Contents

The first example shown in Figure 4.1 is a template to add a table of contents to a given input XHTML documents. That is, it retrieves the heading elements from the input document, constructs a table of contents as a tree of itemized lists that reflect the hierarchical structure of the input, and prepends the table to the original document. More specifically, the template converts the flat structure of `h2`, `h3`, `h4`, and `h5` in the input to an explicit hierarchy using `ul` and `li` itemization. For example, the template transforms the input

```
<html>
  <head><title>Title</title></head>
  <body>
    <h1>Title</h1>
    <h2>Chapter 1</h2>
```

```
pred subheading(var1 a, var1 b, var2 B, var2 A) =
   b in B & a<b & all1 x:(a<x & x in A => b<x);

{visit b :: /<html>/b:<body> :: body[
  h1["index"]
  ul[ {gather h2 :: h2 in <h2> ::
        li[ {gather t :: h2/t :: t} ul[
          {gather h3 :: subheading(h2,h3,<h3>,<h2>) ::
            li[ {gather t :: h3/t :: t} ul[
              {gather h4 :: subheading(h3,h4,<h4>,<h3>) ::
                li[ {gather t :: h4/t :: t} ul[
                  {gather h5 :: subheading(h4,h5,<h5>,<h4>) ::
                    li[ {gather t :: h5/t :: t} ]
                  } ]]} ]]} ]]} ]
  {gather c :: b/c :: c}
]}
```

Figure 4.1: Example: Table of Contents

```
    <h3>Section 1.1</h3> <p>The quick</p>
    <h4>Section 1.1.1</h4> <p>brown fox</p>
    <h3>Section 1.2</h3> <p>jumps over</p>
    <h2>Chapter2</h2>
    <p>the lazy</p>
    <h3>Section 2.1</h3> <p>dog.</p>
    <h2>Chapter3</h2>
  </body>
</head>
```

to the following XHTML document:

```
<html>
  <head><title>Title</title></head>
  <body>
    <h1>Index</h1>
    <ul> <li>Chapter 1 <ul>
           <li>Section 1.1 <ul>
```

```
          <li>Section 1.1.1 <ul/></li>
        </ul></li>
        <li>Section 1.2 <ul/></li>
      </ul></li>
      <li>Chapter 2 <ul>
        <li>Section 2.1 <ul/></li>
      </ul></li>
      <li>Chapter3 <ul/></li> </ul>
  <h1>Title</h1> (...snip... the same content as the input follows)
 </body>
</head>
```

Let us explain the overall structure, using the transformation template on `h3` elements. The part of the template related to `h3` elements is as follows:

```
ul[ {gather h3 :: subheading(h2,h3,<h3>,<h2>) ::
  li[ {gather t :: h3/t :: t}
      ...
    ]} ]
```

By the query `subheading(h2,h3,<h3>,<h2>)` we gather all `h3` elements that are subheadings of already selected `h2` elements. For each selected `h3` element, we generate a list item with its content to be the copies of all child elements of the corresponding `h3` element ({`gather t::h3/t::t`}). We nest similar transformations on `h4` and `h5` elements inside each list item and constructs the whole hierarchy.

The macro `subheading(a,b,B,A)` intuitively means as follows:

> "The node `b` belongs to the set `B`, and it appears after the node `a` and before any nodes `x` in `A` placed after `a` in the document order"

Thus, for example, the query `subheading(h2,h3,<h3>,<h2>)` gathers the nodes `h3` labeled `<h3>` and appearing between the current `h2` and the next node labeled `<h2>` if such `<h2>` exists, or otherwise all `<h3>` nodes appearing between the current `h2` and the end of the input. In other word, it gathers all sections (`<h3>`) in the current chapter (`<h2>`).

Although each sub-relation – "a node is labeled `B`", "a node appears after another node in document order", and so on – is expressible in almost all query languages, several languages such as XPath fails combining them up into one `subheading` predicate. This is because XPath cannot express a conjunction of two or more relations and an universal quantification in general, while MSO, which has regular expressiveness, naturally represent such combinations by using logical operators, as we have seen.

4.1.2 Linguistic Queries

The second example is taken from a motivating example of LPath language developed by Bird, Chen, Davidson, Lee, and Zheng [26]. LPath is an extension to XPath that supports *linguistic queries*. In the field of linguistics, parsed sentences are commonly represented as labeled trees. An example of such tree looks like:

```
<S id="1">
    <NP id="2">I</NP>
    <VP id="3">
        <V id="5">saw</V>
        <NP id="6">
             <NP id="7">
               <Det id="9">the</Det><Adj id="10">old</Adj><N id="11">man</N>
             </NP>
             <PP id="8">
               <Prep id="12">with</Prep>
               <NP id="13"><Det id="14">a</Det><N id="15">dog</N></NP>
             </PP>
        </NP>
    </VP>
    <N id="4">today</N>
</S>
```

The authors of LPath argued that there are mainly three requirements for linguistic queries: “subtree scoping” that restricts the scope of queries in a specified subtree, “edge alignment” condition to state whether a node is leftmost (or rightmost) within a particular subtree, and “immediately follow” relationship. A node q is defined to immediately follow p when p appears immediately after q in some proper analysis [27], where a proper analysis is a sequence obtained by several reverse applications of given grammar productions to a given sentence, e.g., `NP saw NP today` is an example of proper analysis for the sentence “I saw the old man with a dog today.”

LPath extends XPath to support the three features above, and enables us to write many queries that are not expressible in XPath. The authors give the following as test cases:

Q_1 Find noun phrases that immediately follow a verb.

Q_2 Within a given verb phrase, find nouns that follow a verb which is a child of the verb phrase.

Q_3 Find all verb phrases that are comprised of a verb, a noun phrase, and a prepositional phrase.

All these queries are already expressible in our MSO queries without any extensions as show in Figure 4.2. In LPath implementation, "immediately following" relation was defined algorithmically and its equivalence to the definition based on proper analyses needed to be proved. Using second-order variables, we can define the relation directly in terms of the concept of proper analyses. First we prepare a macro to assert that a set `A` is a proper analysis.

```
pred proper_analysis(var2 A) =
  all1 x: (x in A <=> ~(A//x | x//A));
```

That is, a proper analysis `A` is a set of positions such that for any node `x`, if `x` belongs to `A` then all ancestors and descendants of `x` do not belong to `A`, and otherwise there exists an element of `x` being an ancestor or a descendant of `x`. Using this macro, the "`p` immediately follows `q`" relation can be expressed directly through the definition "in some proper analysis `p` appears immediately after `q`."

```
pred imm_follow(var1 x, var1 y) =
  ex2 A: (proper_analysis(A) & x in A & y in A & x<y
                           & ~ex1 z:(z in A & x<z & z<y));
pred follow(var1 x, var1 y) =
  ex2 A: (proper_analysis(A) & x in A & y in A & x<y);
```

The `imm_follow` relation can be directly read as "in some proper analysis `A` that contains both `x` and `y`, there exists no `z` appearing between `x` and `y` (i.e., `y` appears just after `x`." By virtue of the existence of second-order variables, the condition like "in some proper analysis" is naturally representable as `ex2 A: (proper_analysis(A) ...)` in MSO.

4.1.3 Relax NG Simplification

RELAX NG specification [16] defines several simplification rules for transforming a RELAX NG schema into a simpler syntax. Although many of the transformations are easily realizable in traditional XML transformation languages, some of them require a

```
pred  leftmost(var1 x) = ~ex1 y: nextSibling(y,x);
pred rightmost(var1 x) = ~ex1 y: nextSibling(x,y);

pred lmd(var1 a, var1 d) =
  a//d & all1 x:(a//x//d | x=d =>  leftmost(x));
pred rmd(var1 a, var1 d) =
  a//d & all1 x:(a//x//d | x=d => rightmost(x));
pred comp(var1 c, var1 y1, var1 y2, var1 y3) =
  lmd(c,y1) & imm_follow(y1,y2) & imm_follow(y2,y3) & rmd(c,y3);

pred Q1(x) =
  ex1 v:(v in <V> & imm_follow(v,x) & x in <NP>);

pred Q2(x) =
  ex1 vp: ex1 v: (vp:<VP>/v:<V> & follow(v,x) & vp//x:<N>);

pred Q3(x) =
  ex1 v: ex1 np: ex1 pp:
    (v in <V> & np in <NP> & pp in <PP> & _ in <VP>
     & comp(_,v,np,pp));

test[
  Q1[ {gather x :: Q1(x) :: x} ]
  Q2[ {gather x :: Q2(x) :: x} ]
  Q3[ {gather x :: Q3(x) :: x} ]
]
```

Figure 4.2: Example: Linguistic Queries

more sophisticated approach. We take up their "`empty` element" rule as an example. The `empty` element in RELAX NG schema means an empty sequence of nodes. Here is an excerpt from their specification:

> In this rule, the grammar is transformed so that an `empty` element does not occur as a child of a `group`, `interleave`, or `oneOrMore` element or as the second child of a `choice` element. A `group`, `interleave` or `choice` element that has two `empty` child elements is transformed into an `empty` element. A `group` or `interleave` element that has one `empty` child element is transformed into its other child element. A `choice` element whose second child element is an `empty` element is transformed by interchanging its two child elements. A `oneOrMore` element that has an `empty` child element is transformed into an `empty` element. The preceding transformations are applied repeatedly until none of them is applicable any more.

Without a sufficient expressive query language, achieving the desired result requires us to repeat the above transformations many times *until none of them is applicable any more*. By using MSO's ability to capture all regular queries, we can write the condition whether a node should finally be converted to an `empty` element, as follows:

```
pred convertible_to_empty(var2 Empty) =
  all1 x: (x in Empty <=>
      x in <empty>
    | x in <group>      & all1 y: (x/y => y in Empty)
    | x in <interleave> & all1 y: (x/y => y in Empty)
    | x in <choice>     & all1 y: (x/y => y in Empty)
    | x in <oneOrMore>  & all1 y: (x/y => y in Empty);

pred emp(var1 x) =
  ex2 E: (convertible_to_empty(E) & x in E);
```

Using the predicate, the `empty` element simplification can be executed as a one-pass transformation, which is more efficient than repeated transformations.

Figure 4.3 shows the implementation in MTran. We assume that the input is a valid RELAX NG schema, and that each `group` and `interleave` node has exactly two children. If a node `x` is convertible to `empty`, then we output an `empty` node. Otherwise, if `x` is a `group` or `interleave` element with an `empty` child, we translate the node to the other child that is non-`empty`. If the node `x` is a `choice` node, then

```
{visit x
   :: emp(x) ::
     empty[]
   :: (x in <group> | x in <interleave>) & ex1 y:(x/y & emp(y)) ::
     {gather y :: x/y & ~emp(y) :: y}
   :: x in <choice> ::
     choice[ {gather y :: x/y &  emp(y) :: y}
             {gather y :: x/y & ~emp(y) :: y} ]
}
```

Figure 4.3: `empty` element simplification

we bring the `empty` child before, as stated in the simplification rule. Any other node is kept unchanged (which is ensured by the semantics of `visit`).

The example above shows the advantage of *no-recursion* of both MSO and our semantics of `visit` expressions. Regular expressiveness of MSO enables us to check whether a node is convertible to `empty` by a single query `emp(x)`, without writing any explicit recursive tree traversals. The semantics of `visit` expressions eliminates the necessity to explicitly write down a recursive application of the transformation in the template for `group`, `interleave`, and `choice` elements. Without our implicit recursion semantics, we would had to specify that we need to recursively transform `gather`ed elements `y`.

4.2 Preliminary Performance Evaluation

The experiments presented in this section are use the three examples we have presented in the preceding section. All benchmarks are run on Windows XP SP2 operating system on a 1.6GHz AMD Turion processor with 1GB RAM, using MONA 1.4 as a backend compiler of MSO formulae. We have implemented our system in C++ programming language and compiled in GNU C++ Compiler [28].

Our implementation strategy of transformation templates consists of two steps: (1) compilation of MSO formulae to tree automata using MONA system as explained in Section 3.2 and (2) query evaluation and transformation with the compiled tree automaton. We experiment on each step separately.

Table 4.1 shows the total time spent for compiling all query expressions in each

TableOfContents	0.970
Linguistic	0.655
RelaxNG	0.553

Table 4.1: Compilation Time (sec)

program. Although compilation of MSO formulae to tree automata is known to take hyper-exponential time in the theoretical worst case, the experiment shows that at least for these three examples of XML queries, our strategy yields enough performance.

We measure the performance of our evaluation algorithm using randomly generated XML documents of different sizes as inputs. Table 4.2 shows the result. We can confirm that the transformations are also executed in practical time consumption.

	10KB	100KB	1MB
TableOfContents	0.038	0.320	3.798
Linguistic	0.063	0.429	4.050
RelaxNG	0.068	0.540	5.684

Table 4.2: Evaluation Time (sec)

4.2.1 *N*-ary query performance

To demonstrate the efficiency of our *n*-ary query algorithm and template evaluation strategy, we compare the performances of same transformation between MTran and traditional XSLT processors: XT [29] and Xalan-C [30]. For benchmark, we write, both in MTran and XSLT, a transformation that appends, to the content of each `h2` element, the content of its preceding `h1` element.

```
pred preceding(var h1, var x) =
  h1 in <h1> & h1<x & all1 y:(y in <h1> & y<x => ~h1<y);

{visit x :: x in <h2> ::
   h2[ {gather c :: x/c:# :: c} " - "
       {gather h1 :: preceding(h1,x) :: {gather c::h1/c:#::c}} ]
}
```

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="h2">
    <h2><xsl:value-of select="."/> -
        <xsl:value-of select="preceding::h1[1]"/></h2>
  </xsl:template>
  <xsl:template match="@*|node()">
    <xsl:copy><xsl:apply-templates/></xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

As we explained in Section 3.4.1, MTran processes the query `preceding(h1,x)` as a binary query and evaluates it only once. On the other hand, XT and Xalan-C evaluate the XPath query `preceding::h1[1]` for each node `h2`. This difference becomes clear in the case that the input document contains a large number of `h2` elements. Our $O(t+s)$ algorithm evaluate the binary query `preceding(h1,x)` in MTran version in linear time with respect to the size of the input, while XT and Xalan-C evaluate the unary query `preceding::h1[1]` in linear time and repeats the evaluation for the number of `h2` times. Table 4.3 and Figure 4.4 show that the evaluation time for the following form of inputs

```
<html>
  <h1>aaa</h1>
  <h2>bbb</h2> <h2>bbb</h2> ... <h2>bbb</h2>
</html>
```

varying the number of `h2` elements from 1000 to 9000. MTran performs the transformations linearly, while XT and Xalan-C incur quadratic blowup.

(size)	1000	2000	3000	4000	5000	6000	7000	8000	9000
MTran	0.155	0.293	0.433	0.580	0.767	0.898	1.020	1.261	1.430
Xalan-C	0.155	0.416	0.850	1.442	2.242	3.175	4.289	5.579	7.040
XT	1.014	2.537	5.077	8.834	15.657	23.788	32.870	42.483	56.327

Table 4.3: Evaluation Time(sec)

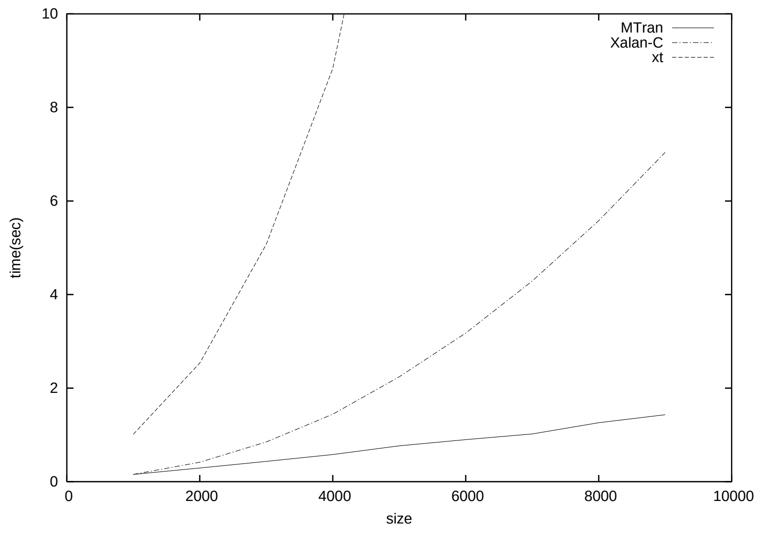


Figure 4.4: Evaluation Time

Chapter 5

Conclusion and Future Work

5.1 Conclusion

We designed and implemented an XML transformation language MTran, which adopts monadic second-order logic formulae as the query language. We showed four advantages of MSO: expressiveness to capture all regular queries, elimination of recursions, don't-care semantics that allows us not to mention the nodes irrelevant to a query, and natural n-ary queries. We have developed a tree automata based algorithm for efficient processing of n-ary queries. Our algorithm improved the worst case time complexity from the previously known best result $O(t^n + s)$ to $O(t + s)$, linear in the size of the input and the output. Using MONA system as a backend MSO compiler, we have implemented the algorithm in our MTran interpreter.

5.2 Future Work

Full-scale XML processing

Many issues still remain as future work. One direction is the full-scale language design and implementation to support all features of XML documents. Adding support for namespaces, comments, and processing instructions to current MTran is not so a difficult problem. The largest theme in this direction is the support of text transformations. Our current implementation only can copy or remove text nodes, and has no support for the manipulation of the content of text nodes. Although it is possible to extend our language with many built-in string manipulation functions, we plan to investigate more expressive and coherent approach: text transformation based on MSO. The advantages of MSO like regular expressiveness or don't-care semantics also could be applied in the field of text transformation. Moreover, by integrating the language for XML and text transformation, uniform manipulation of structured and unstructured data format (such as, comma separated texts) would be achieved.

Static Type Checking

Static type checking is an important problem for programming languages. Many researches [5, 31, 32, 33] have been done in the area of type checking for regular tree languages. We would like to investigate how those results are applicable to our language. Type checking problem for our distinct semantics of `visit` expressions is particularly interesting. Also we wish to study query optimization based on type information. The MONA system, which we adopted as a backend of MTran language, exploits a kind of automata optimization based on static information through *guided tree automata* [34]. We are considering the use of guided tree automata for optimization over types.

'Move' Transformation

As stated in Chapter 1, MSO is well-suited to describe relations between distant nodes. Our language leverages this advantage. Accumulation of distant nodes is achieved through `gather` expressions and simultaneous translation of distant nodes is realized by `visit` expressions. Nevertheless it is awkward in our language to describe a transformation that *moves* one node to another distant place in the input tree while keeping the other nodes unchanged.

We are already able to express such transformation through `visit` and `gather` expressions, by visiting and erasing the node to be moved, and on another front visiting the destination of the move and gathering the node from original position. However, this approach seems to be redundant that we need to write the same query in two places (in the `visit` expression for erasing and in the `gather` expression). Node moving transformation is occasionally required in practice, and should be solved in more concise manner. We plan to investigate the language design to achieve concise move transformation as further research.

References

[1] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franşois Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). `http://www.w3.org/TR/REC-xml`, Feb 2004.

[2] Frank Neven. Automata, logic, and xml, 2002.

[3] Christoph Koch. Efficient processing of expressive node-selecting queries on xml data in secondary storage: A tree automata-based approach. In *VLDB*, pages 249–260, 2003.

[4] Sebastian Maneth and Frank Neven. Structured document transformations based on XSL. In *DBPL*, pages 80–98, 1999.

[5] Sebastian Maneth, Thomas Perst, Alexandru Berlea, and Helmut Seidl. XML type checking with macro tree transducers. In *PODS*, 2005.

[6] James Clark and Steve DeRose. XML path language (XPath) version 1.0. `http://www.w3.org/TR/xpath`, Nov 1999.

[7] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[8] Peter Buneman, Mary F. Fernandez, and Dan Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *VLDB Journal*, 9(1):76–110, 2000.

[9] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML, 1998. `http://www.w3.org/TR/NOTE-xml-ql`.

[10] Anne Brüggemann-Klein and Derick Wood. Caterpillars: A context specification technique. *Markup Languages*, 2(1):81–106, 2000.

[11] Makoto Murata. Extended path expressions for XML. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, pages 126–137, 2001.

[12] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 13(6):961–1004, 2002. Short version appeared in Proceedings of The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 67–80, 2001.

[13] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 51–63, 2003.

[14] J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to a decision problem of second-order logic, 1968.

[15] David C. Fallside. XML Schema Part 0: Primer, W3C Recommendation. {`http://www.w3.org/TR/xmlschema-0/`}, 2001.

[16] James Clark and Makoto Murata. RELAX NG. {`http://www.relaxng.org`}, 2001.

[17] Alexandru Berlea and Helmut Seidl. Binary queries. In *Extreme Markup Languages 2002*, August 2002.

[18] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA 1.4. `http://www.brics.dk/mona/`, 1995.

[19] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002. World Scientific Publishing Company. Earlier version in Proc. 5th International Conference on Implementation and Application of Automata, CIAA '00, Springer-Verlag LNCS vol. 2088.

[20] Frank Neven and Jan Van den Bussche. Expressiveness of structured document query languages based on attribute grammars. In *PODS '98. Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 11–17. ACM press, 1998.

[21] Alexandru Berlea and Helmut Seidl. fxt - a transformation language for XML documents. In *XML Conference And Exposition 2001*, Dec 2001.

[22] Joachim Niehren, Laurent Planque, Jean-Marc Talbot, and Sophie Tison. N-ary queries by tree automata. In *DBPL*, Aug 2005.

[23] Keisuke Nakano. An implementation scheme for XML transformation languages through derivation of stream processors. In *The 2nd ASIAN Symposium on Programming Languages and Systems*, volume 3302 of *Lecture Notes in Computer Science*, pages 74–90, 2004.

[24] J. R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundladen der Mathematik*, 6:66–92, 1960.

[25] Xavier Leroy, Damien Doligez, Jacques Garrigue, Jérôme Vouillon, and Dider Rémy. The Objective Caml system. Software and documentation available on the Web, {`http://pauillac.inria.fr/ocaml/`}, 1996.

[26] Steven Bird, Yi Chen, Susan Davidson, Haejoong Lee, and Yifeng Zheng. Extending XPath to support linguistic queries. In *Informal Proceedings of the Workshop on Programming Language Technologies for XML PLAN-X 2005*, pages 35 – 46, 2005.

[27] A. Halevy, Z. Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semantic web applications, 2003.

[28] GNU compiler collection. `http://gcc.gnu.org/`.

[29] Bill Lindsey and James Clark. XT version 20051206. `http://www.blnz.com/xt/`, Dec 2005.

[30] The Apache Software Foundation. Xalan c++ 1.10. `http://xml.apache.org/xalan-c/`, Feb 2004.

[31] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM SIGPLAN Notices*, 35(9):11–22, 2000.

[32] Dan Suciu. Typecheking for semistructured data, 2001.

[33] Nils Klarlund, Thomas Schwentick, and Dan Suciu. Xml: Model, schemas, types, logics, and queries. In *Logics for Emerging Applications of Databases*, pages 1–41, 2003.

[34] Morten Biehl Christiansen, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata, 1997.