XML Transformation Language Based on Monadic Second Order Logic

Kazuhiro Inaba
The University of Tokyo
kinaba@arbre.is.s.u-tokyo.ac.jp

Haruo Hosoya
The University of Tokyo
hahosoya@arbre.is.s.u-tokyo.ac.jp

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 paper, 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 design our templates specially suitable for expressing structure-preserving transformation, eliminating the need for explicit recursive calls to 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) a linear time query evaluation algorithm for tree automata. For the latter, our algorithm is similar to Flum-Frick-Grohe algorithm for MSO queries locating n-tuples of *sets of* nodes, except that ours is specialized to queries for tuples of *nodes* and employs a *partially lazy set operations* for attaining a simpler implementation with a fewer number of tree traversals. We have made experiments and confirmed that our strategy yields a practical performance.

1. Introduction

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 of XML processing. For example, there have been proposals for XML query languages whose expressivenesses are provably MSO-equivalent [28, 20] and for theoretical models for XML transformation with MSO as a sublanguage for node selection [22, 23]. However, little attempt has been made for bringing MSO logic formulae themselves into an actual language system for XML processing.

Proceedings of the 5th ACM SIGPLAN Workshop on Programming Language Technologies for XML (PLAN-X 2007).
January 20, 2007, Nice, France.

The goal of our work is to design and implement a practical XML transformation language called *MTran* based on MSO queries, in particular, addressing 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.

Our implementation of MTran is publicly available in
`http://arbre.is.s.u-tokyo.ac.jp/~kinaba/MTran`.

1.1 Why MSO?

MSO is first-order logic extended with second-order variables ranging over sets of domain elements in addition to first-order variables ranging over domain elements themselves. Among various variants, WS2S (Weak Second-order logic with two Successors) is a kind of MSO specialized to express propositions over finite binary tree structures. 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 [9, 1, 7, 25], pattern matches [15, 2], and monadic datalog queries [20] have some of these properties, MSO is the only language that has all of them, as we argue below (the summary is in Table 1).

Regularity A query over trees is called regular when there is an equivalent tree automaton with an appropriate alphabet (Section 4.1). MSO is known to be able to express all regular queries [31], while most of existing path-based node selection languages (including XPath [9], currently the most popular path language) do not have this property. This lack of regularity does not only indicate theoretical weakness, but also has a practical impact since it fails to represent even slightly complicated conditions. An obvious exam-

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

Table 1. Comparisons between query languages

ple 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 [6], XML Schema [11], and RELAX NG [10] heavily rely on regular expressions for trees (in particular, RELAX NG schemas can represent any regular tree 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 2.1, whereas it is not in most path languages.

No recursion The way that MSO formulae express retrieval conditions is, in a sense, "logically direct." 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 `<img>` 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 <img>
```

("node `x` that has label `img`") in MSO.

Don't-care semantics The directness of MSO 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, consider writing a query that retrieves the set of nodes `x` containing at least one child node labeled `<date>`. In regular expression patterns, we would write as follows

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

where we have to "mention" the siblings and the content of the `<date>` 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 <date>
```

("node `x` where some node `y` is a child of `x` and has label `date`") where we only refer to the nodes of our interest: the node `x` itself and the child `<date>` 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. (Although the MSO formula in the above example is not much smaller than the pattern, we will later see plenty of MSO examples that express various complicated queries with remarkably small formulae. Theoretically, it is known that MSO formulae can in general be *hyper-exponentially* smaller than their equivalent regular expressions [14].)

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<y & y<z & y in <item>
```

expresses the condition for three nodes `x`, `y`, and `z` that they share a common parent node `p`, that they appear in this order, and that the node `y` is tagged with `<item>`. Although path-based query languages like XPath suit to express binary queries (that is, relations between a previously selected node, i.e., context node, and another node), they cannot represent general n-ary queries. Similarly, monadic datalog can express arbitrary unary MSO formulae but not any higher-arity queries.

1.2 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. Suppose we want to enclose with a `<li>` every `<ul>` element whose parent is also an `<ul>` element. (Direct nesting of `<ul>` elements is a common mistake in representing nested lists in XHTML. The transformation is intended to correct it and emits a valid XHTML document.) In our language, this transformation can be written by the following one line:

```
{visit x :: <ul>/x & x in <ul> :: li[x]}
```

Here, we first select all `<ul>` elements with `<ul>` parent by the MSO formula "`<ul>/x & x in <ul>`" and then transform each of these elements accordingly to the associated rule, i.e., enclose the element by the `<li>` tag. The whole output is the reconstruction of the input tree where each selected element is replaced by the result of its local transformation.

Compare the above program in our language with the same transformation written in XSLT [8]:

```
<xsl:stylesheet version="1.0" ...>
  <xsl:template match="ul[parent::ul]">
    <li>
     <ul>
      <xsl:apply-templates select="@*|node()"/>
     </ul>
    </li>
  </xsl:template>
  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

In this, after selecting a `<ul>` element, we create a `<li>` element containing a `<ul>` element and *then* explicitly make a recursive application of the template to the child nodes (using an instruction `<xsl:apply-templates/>`) for computing the content of the `<ul>` element. Our design principle is to eliminate such explicit recursion and thus avoid the necessity to follow the data flow for understanding the program, which makes transformation more intuitively readable and writable for naive programmers.

Also, in XSLT, we need an explicit template that recursively copies all unremarked nodes. However, as discussed in the preceding section, one of the benefits of MSO is its *don't-care semantics* that allows us to avoid mentioning irrelevant nodes. We further push this merit to our transformation language: our `visit` expressions implicitly copy all irrelevant nodes so as not to bother programmers with writing recursion.

Choice of `visit` and `gather` While a `visit` expression retains in the result the nodes that are *not* matched, we provide another

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

Figure 1. A transformation using binary queries

choice of treating such unmatched nodes, namely, dropping them by using a `gather` expression. It is important to be able specify which to use in each use of an MSO formula since real transformations almost always need a fine control on the structure of the output. As an example, the following

```
<ul> {gather x :: x in <a> :: li[x]} </ul>
```

is similar to the previous example except that it uses `gather` instead of `visit`. The result is a `ul` element containing the list of all `<a>` elements appearing in the input XML, each wrapped by a `<li>` element.

Nested templates XSLT uses XPath binary queries for selecting a node with respect to a single previously selected 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 program in Figure 1. 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>
```

Notably, in the inner-most selection condition for the variable `v`, which is inside the scope where we have already selected a node for `n`, we 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. Note also that this example only uses binary queries, but it is clear that we can also specify higher-arity queries in the same framework.

1.3 MSO Evaluation Algorithm

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

A slightly more detailed explanation is needed on the motivation. Programs in MTran actually do *not directly* select tuples of nodes that simultaneously satisfy a given condition, but select nodes that satisfy the condition *relative to* nodes already selected by previous queries. An n-ary query algorithm is still useful for this purpose and indeed crucial. To illustrate this, let us see again the example in Figure 1. First of all, notice that we could process each query 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 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 appear 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. Fortunately, if there is an efficient n-ary query algorithm, this can be improved: evaluate the above formula only once for locating all pairs of an element and a `<name>` element that are in the parent-child relation. This observation has first been made by Berlea and Seidl [3] in the context of their language based on binary queries, and can easily be extended to our case with n-ary queries.

We have therefore 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 [17], which has an established reputation in its compact and efficient representation of WS2S MSO formulae by tree automata with binary decision diagrams and is experimentally shown to work quickly for large formulae even of dozens of kilobytes [18]. Our preliminary experiments confirm that, for many typical examples of XML queries, MONA yields adequate performance (Section 5).

For the second step, we have developed an efficient linear-time algorithm for n-ary MSO queries. This algorithm is similar to the one developed by Flum, Frick, and Grohe [12]. However, while they treat general MSO queries with second-order free variables, our language only needs queries with first-order free variables and therefore we specialize their algorithm to our simpler case. In addition, we employ a novel implementation technique called *partially lazy operations on sets of nodes*, by which we obtain a simpler implementation with a fewer number of traversals on the input tree.

1.4 Related Work

DTL [22] and its generalization TL [23] are theoretical models for XML transformation that use MSO formulae for node selection. However, their goals are to find theoretical properties of transformation models (such as decidability of precise typechecking, which is not known for our language) whereas ours is to obtain a concrete design and an efficient implementation technique for a transformation language leveraging the full strength of MSO. Indeed, 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 theirs is limited to binary queries.

Nakano has proposed an XML transformation language XTiSP [26] 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.

Finding a fast algorithm for MSO queries has been a topic attracting numerous researchers. Early work by Neven and van den Bussche has described a linear-time, two-pass algorithm for unary queries based on boolean attribute grammars [27]. Then, Flum, Frick, and Grohe have solved the general case by showing a linear-

```
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 :: b in <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 2. Example: Table of Contents

time, three-pass algorithm [12]. Notably, their algorithm's time complexity is linear both in the sizes of the input and the output. Afterward (apparently not noticing the last work), several other algorithms have been published that either have higher complexity or have linear-time complexity with only restricted cases treated [20, 3, 29], though each of these has made orthogonal efforts in either implementation techniques or theoretical properties. As already mentioned, our contribution with respect to Flum-Frick-Grohe algorithm is the technique using partially lazy set operations for a simpler and quicker (by constant factor) implementation with concrete experimental results.

1.5 Outline

The rest of this paper is organized as follows. Section 2 gives slightly bigger examples in MTran to emphasize the usefulness of the expressiveness of MSO in XML. Section 3 introduces the syntax and semantics of our MTran language, first describing the query part of MTran and then the whole transformation language. Section 4 gives a brief overview of our evaluation strategy. Section 5 shows the results of our preliminary performance evaluation of MTran. Section 6 suggests possible directions for future work.

2. Examples

In this section, we show two examples that expose the expressiveness of MTran language. To save space, we postpone other two examples showing the usefulness of second-order variables in XML to Appendix B.

2.1 XHTML Table of Contents

The example shown in Figure 2 is a template to add a table of contents to a given input XHTML document. It retrieves the heading elements from the input document, constructs a tree of itemized lists that reflect the hierarchical structure of the input, and prepends it to the original document. Note that the original document contains the flat structure of h2, h3, h4, and h5 and we turn this implicit hierarchy to an explicit one using nested ul itemizations. For example, the template transforms the input

```
<html><head><title>Title</title></head><body>
  <h1>Title</h1>
  <h2>Chapter 1</h2>
  <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>
</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> </ul>
   (... the same content as the input follows ...)
</body></head>
```

The template begins with the macro subheading(a,b,B,A) intuitively meaning the following:

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

For example, we use this macro as a query subheading(h2,h3, <h3>,<h2>) to collect all nodes h3 that are labeled <h3> and appears between the current node h2 and the next node labeled <h2> if such <h2> exists, or otherwise all <h3> nodes that appear after the current h2. (Here, <h2> and <h3> are constants respectively denoting the sets of nodes labeled <h2> and <h3>.) In other words, 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 standard in usual XML query languages, their combination as in the subheading predicate is not commonly expressible; in particular, XPath is incapable of this since, essentially, XPath cannot express universal quantification.

Then, the main template expression has the following structure. Let us first focus on the subexpression treating h3 elements:

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

By the query subheading(h2,h3,<h3>,<h2>), we collect all h3 elements that are subheadings of already selected h2 elements. For each selected h3, we generate a list item with its content to be the copies of all child elements ({gather t::h3/t::t}) of the h3 element. Inside each list item, we nest the result of a similar transformation on h4 and so on, constructing the whole hierarchy.

2.2 MathML Conversion

The second example is to convert a MathML [24] document to another. MathML is a standard XML format to markup mathematical expressions, whose elements fall into two categories, namely, *content-markup* elements for representing syntactic structure of expressions and *presentation-markup* elements for encoding their visual rendering. For example, a mathematical expression $(2 + 3) \times (4 + (5 + 6))$ is written in content markup as follows:

```
<apply>
 <times/>
 <cn>1</cn>
 <apply> <plus/> <cn>2</cn> <cn>3</cn> </apply>
 <apply> <plus/>
   <cn>4</cn>
   <apply> <plus/> <cn>5</cn> <cn>6</cn> </apply>
 </apply>
</apply>
```

```
pred follows( var1 x, var1 y ) =
  ex1 p: (p/x & p/y & x<y);
pred need_paren( var1 ap ) =
  ap/<plus>
   & ex1 op: (follows(op,ap) & op in <times>);

mrow[ {visit x
  :: x in <ci> ::
    mi[ {gather y :: x/y :: y} ]
  :: x in <cn> ::
    mn[ {gather y :: x/y :: y} ]
  :: x in <apply> & need_paren(x) ::
    mo["("] {gather y::firstChild(x,y)::y} mo[")"]
  :: x in <apply> ::
    {gather y :: firstChild(x,y) :: y}
  :: x in <plus>  ::
    {gather y :: nextSibling(x,y) ::
        y {gather z :: follows(y,z) :: mo["+"] z}
  :: x in <times> ::
    {gather y :: nextSibling(x,y) ::
        y {gather z :: follows(y,z) :: mo["*"] z}
}]
```

Figure 3. Example: MathML Conversion

The MTran program shown in Figure 3 converts a content markup containing only <plus/> and <times/> as operators to a presentation markup, where we minimize the number of occurrences of parentheses, based on the standard priority rules for operators. For instance, the above XML is converted to the following XML in presentation markup with no redundant parentheses:

```
<mrow>
 <mo>(</mo><mn>2</mn><mo>+</mo><mn>3</mn><mo>)</mo>
 <mo>*</mo><mo>(</mo><mn>4</mn><mo>+</mo><mn>5</mn>
 <mo>+</mo><mn>6</mn><mo>)</mo>
</mrow>
```

Two macro predicates are defined. The first one `follows(x,y)` means that the nodes `x` and `y` share the same parent (`p`) and `x` appears before `y` in the document order. That is, the node `x` is one of the preceding siblings of the node `y`. The second one `need_paren(ap)` takes an <apply> node as the parameter `ap` and determines whether it is required to enclose the expression with parentheses. The rule here is that we need parentheses only when the operation used in the `ap` node is <plus> and the outer operation (`op`) is <times>.

The template generates a <mrow> element, in which we use a `visit` expression to visit all elements in the input document, apply the transformation with associated sub-templates, and glue them up into the output document. When an <apply> element is found (`x in <apply>`), we emit parentheses if the `need_paren` predicate returns true and then extract, by a `gather` expression, the first child, which is either a <plus> or a <times> node. At a <plus> node, we construct a sequence of addition expressions. In this, we first emit the first operand (which is obtained by `nextSibling(x,y)`) and then each remaining operand (which is extracted by `follows(y,z)`) prepended with the operator symbol `mo["+"]`. We process a <times> node in a similar way.

The example shows the power of `visit` expressions. That is, we have only specified a local transformation on each node in the input without involving any explicit recursive traversal. Nevertheless, the program can perform a whole-document conversion where the presentation markups in the output document preserve the original structure of the content mark-ups in the input document.

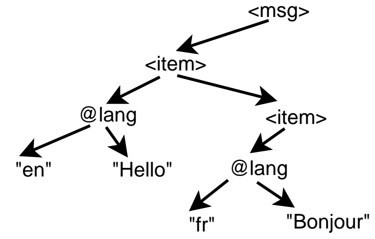

Figure 4. The binary tree representation of the example XML

3. Language

3.1 Binary Trees and XML representation

Our formalization begins with defining binary trees. Throughout this paper, we assume a fixed, finite alphabet Σ.

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

Our surface language needs to handle XML documents, which are in general not binary trees but are unranked trees (trees whose each node has an arbitrary number of child nodes). For this, we use a well-known encoding of unranked trees by binary trees. That is, the first child and the right neighboring sibling of each node in the unranked tree are, respectively, encoded by the left and the right child of the corresponding node in the binary tree. In addition, a real XML document has three types of node—element nodes, attribute nodes, and text nodes. For this, we first assume the alphabet Σ to consist of element names written <e>, attribute names written @a, and texts written "s". Then, we insert attribute nodes before the other element or text nodes of its belonging element node. For example, the following XML document

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

is modeled by the binary tree in Figure 4.[1] Our current implementation ignores other XML features like comments, processing instructions, and namespaces.

3.2 Query Expressions

This section describes MTran's query sublanguage based on MSO. We first present the core syntax and semantics of query expressions and then introduce syntax sugars for convenience and reusability of queries.

3.2.1 Core Syntax and Semantics

As we mentioned in the introduction, we adopt MSO formulae as our query language. MSO is a kind of logic that allows second-order variables ranging over sets of domain elements in addition to first-order variables ranging over domain elements, where domain elements here are tree nodes in our context. Formally, we assume a set of first-order variables, ranged over by x, and a set of second-order variables, ranged over by X. The BNF below defines the core syntax for first-order terms p, second-order terms S, and MSO formulae φ.

$p ::= x \mid \mathtt{root}$

[1] Although we formalize binary trees on top of the fixed alphabet Σ, actual input XML documents may have arbitrary labels possibly not belonging to Σ. To treat this, we always add an extra, distinguished symbol `others` to Σ, and rename any label in input trees not belonging to Σ as an `others`.

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

Let us informally explain the semantics of MSO. An MSO formula φ is interpreted in terms of a binary tree t over Σ, a first-order assignment γ (mapping each first-order variable to an element of $Pos(t)$), and a second-order assignment Γ (mapping each second-order variable to a subset of $Pos(t)$). A first-order term denotes a node in the tree t. In particular, the constant `root` denotes the root node ε of t. A second-order term denotes a set of nodes. Each symbol $\sigma \in \Sigma$ works as a constant denoting the set of nodes labeled with σ. The other three kinds of second-order constant terms `<*>`, `@*`, and `#` are called *wild-card* terms denoting the set of any element nodes, the set of any attribute nodes, and the set of any text nodes, respectively.

The formulae p `=` p (and S `=` S) means that the terms on both side denote the same node (and the same set of nodes, respectively). The formulae p `in` S means that the node denoted by p belongs to the set denoted by S. The operators `~`, `&`, and `|` are the standard logical operators for negation, conjunction, and disjunction. The constructs `ex1` and `all1` are quantification over first-over variables. Similarly, `ex2` and `all2` are quantification over second-over variables. The last two constructs of φ are the primitive predicates to relate tree nodes. That is, `firstChild(`p`,`q`)` holds if and only if q is the first child of node p. Similarly, `nextSibling(`p`,`q`)` holds if and only if q is the next sibling of node p. These two primitives correspond to the edge relations in the binary tree encoding in Section 3.1.

Formally, the semantics of terms and formulae is defined as follows:

Definition 2. Let t be a binary tree over Σ. Under a first-order assignment γ, first-order terms are interpreted as follows:

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

Also, under a second-order assignment Γ, second-order terms are interpreted as follows:

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

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:

$$
\begin{aligned}
t,\gamma,\Gamma &\vDash p_1\texttt{=}p_2 && \iff \gamma[p_1] = \gamma[p_2] \\
t,\gamma,\Gamma &\vDash S_1\texttt{=}S_2 && \iff \Gamma[S_1] = \Gamma[S_2] \\
t,\gamma,\Gamma &\vDash p \;\texttt{in}\; S && \iff \gamma[p] \in \Gamma[S] \\
t,\gamma,\Gamma &\vDash \texttt{\~{}}\varphi && \iff t,\gamma,\Gamma \nvDash \varphi \\
t,\gamma,\Gamma &\vDash \varphi_1\&\varphi_2 && \iff t,\gamma,\Gamma \vDash \varphi_1 \text{ and } t,\gamma,\Gamma \vDash \varphi_2 \\
t,\gamma,\Gamma &\vDash \varphi_1|\varphi_2 && \iff t,\gamma,\Gamma \vDash \varphi_1 \text{ or } t,\gamma,\Gamma \vDash \varphi_2 \\
t,\gamma,\Gamma &\vDash \texttt{ex1}\; x\texttt{:}\varphi && \iff \text{for some } a \in Pos(t) \quad t,\gamma_{x:=a},\Gamma \vDash \varphi \\
t,\gamma,\Gamma &\vDash \texttt{all1}\; x\texttt{:}\varphi && \iff \text{for all } a \in Pos(t) \quad t,\gamma_{x:=a},\Gamma \vDash \varphi \\
t,\gamma,\Gamma &\vDash \texttt{ex2}\; X\texttt{:}\varphi && \iff \text{for some } A \in 2^{Pos(t)} \quad t,\gamma,\Gamma_{X:=A} \vDash \varphi \\
t,\gamma,\Gamma &\vDash \texttt{all2}\; X\texttt{:}\varphi && \iff \text{for all } A \in 2^{Pos(t)} \quad t,\gamma,\Gamma_{X:=A} \vDash \varphi \\
t,\gamma,\Gamma &\vDash \texttt{firstChild(}p_1\texttt{,}p_2\texttt{)} && \iff \gamma[p_1].\texttt{l} = \gamma[p_2] \\
t,\gamma,\Gamma &\vDash \texttt{nextSibling(}p_1\texttt{,}p_2\texttt{)} && \iff \gamma[p_1].\texttt{r} = \gamma[p_2]
\end{aligned}
$$

Here, $\gamma_{x:=a}$ is an assignment that is identical to γ except that it maps the variable x to a; similarly for $\Gamma_{X:=A}$. The dot operator . used in the definition of `firstChild` and `nextSibling` denotes the concatenation of sequences from $\{\texttt{l},\texttt{r}\}^*$.

```
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 5. Unranked view in binary MSO

3.2.2 Auxiliary Syntax

Macro Expressions Programmers can define their own macros for frequently used formulae. Assuming a set of macro names m, the syntax for macro definitions is as follows:

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

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

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

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

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 3.1. Although, in principle, these two primitives would have sufficient expressiveness, it is much more convenient to have more compact notations based on the *unranked* view of the input XML tree. For this reason, we introduce an XPath-like syntax sugar.

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

Here, D stands for *path delimiters* and U stands for *path units*. The expression `p/q` (and `p//q`) means that the node denoted by `p` is the parent (and an ancestor, respectively) of the node denoted by `q` in the original unranked tree. Since the unranked parent-child relation and the ancestor-descendant relation are indeed expressible in MSO, our implementation internally converts those syntax-sugars to equivalent plain MSO formulae. Namely, the expression `p/q` is converted to `p_c(p,q)` and the expression `p//q` is converted to `a_d(p,q)`, where the macros `p_c` and `a_d` are defined as in Figure 5.

```
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 6. Document Orders

When a second-order term is postfixed to a path expression, it has an existential meaning. For example, `p/<a>` is a shorthand for `ex1 x: p/x & x in <a>`. When a pair `p:S` of a first- and a second-order terms is postfixed to a path expression, it simultaneously declares that `p in S` and that the term `p` satisfies the present path expression. For instance, the expression `x:<a>/y` is equivalent to `x in <a> & x/y`. When three or more units are connected by path delimiters, the expression has a conjunctive meaning. For example, `x/y/z` stands for `x/y & y/z`. A delimiter-prefixed expression $/U_1 D_1 \dots D_n U_n$ represents an *absolute path* from the root node. The expression is interpreted as `(root=`U_1`)` & $U_1 D_1 \dots D_n U_n$ when U_1 is a first-order term, and interpreted as `(root in` U_1`)` &$U_1 D_2 \dots D_n U_n$ when U_1 is a second-order term.

Document Order Relations Another frequently used primitive is the pre-order relation (often called *document order* relation) among tree nodes. This relation is also MSO-expressible (`doc_order` macro in Figure 6) and provided as a short-hand syntax as follows:

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

3.3 Transformation Templates

This section defines the MTran language itself, which embeds our MSO-based query sublanguage given in the last section.

3.3.1 Overview

The most important constructs in MTran are `gather` and `visit` expressions. A `gather` expression *gathers* all nodes in the input tree that satisfy 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 we have gathered all nodes that match the query regardless of their inclusion relations. When we want to obtain only the outermost nodes, we need to explicitly specify so 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. A query to retrieve only the innermost nodes can also be written similarly.

A `visit` expression, on the other hand, *visits* every node that satisfies the associated query formulae and appears in the subtree specified by the `from` clause (or in 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. This is roughly the semantics of our `visit` expressions. However, there are subtle behaviors in the detail. When we evaluate each `visit` expression, we actually distinguish four kinds of nodes that are encountered during the traversal—matched nodes, unmatched nodes, newly generated nodes, and already-processed nodes—and take a different action for each. To illustrate this, let us consider the following template for enclosing each `B` node in a `Mark` tag:

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

This template transforms the input document

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

to the result:

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

Each `A` or `C` node is unmatched, for which we simply skip and proceed to its child nodes. Then, each `B` node is matched, for which we evaluate the subtemplate `Mark[x]` with `x` bound to the node itself, and *then* traverse again the generated tree. During the traversal, we will encounter the `B` node that has already been processed as well as the newly generated `Mark` node; we simply skip both of them. In the same traversal, we will also encounter another `B` node that has not yet been seen, for which we apply the subtemplate `Mark[x]` in the same way as above.

To justify the above design choices, the reason for skipping unmatched nodes is clear: we can release the programmer from explicitly writing recursion for searching and transforming deeply located nodes. The reason for retraversing generated trees is that, otherwise, we cannot transform matched nodes that appear inside another matched node, unless we explicitly write a recursive application—that is, this design is again for our intention to avoid any recursion. Finally, skipping already-processed nodes and newly generated nodes is for simplifying the language design. In particular, this can make the MTran language *terminating* and *transformable in one pass*. In Appendix A, we prove that the evaluation of transformation templates always terminates.

3.3.2 Syntax and Semantics

We formally define the syntax and semantics of the language in the rest of this section. A whole program consists of a list of user-defined macros and a (template) expression, where *expressions* E and *expression lists* EL are defined by the following syntax.

$$EL ::= E^*$$
$$E ::= x \mid \sigma[EL] \mid \{\texttt{gather}\ x :: \varphi :: EL\} \mid \{\texttt{visit}\ x\ \texttt{from}\ y\ (:: \varphi :: EL)^*\}$$

The phrase "`from` y" can be omitted from a `visit` expression when y is `root`. Also, when the σ in an expression $\sigma[\cdots]$ is an element label <e>, the angle brackets can be omitted, i.e., the expression can be written as $e[\cdots]$.

Internal form MTran internally handles three different forms of XML representation. The first is the normal textual representation as defined in the XML standard [6]. The second is the binary encoding of XML trees described in Section 3.1, which is used in our query algorithm to represent input trees. The last is the *internal form*, which is suitable for handling the above-mentioned subtle behavior of our transformation templates. Internal trees I (trees in the internal form) are defined as follows where $p \in \{0,1\}^* \cup \{\bot\}$:

$$I ::= (\sigma\texttt{[}I^*\texttt{]}, p)$$

The first component of an internal tree I represents a node in the unranked tree structure, where σ is its label and I^* is its children. (Note that this definition may yield an invalid XML, such as repeated attributes `elem[@x[...]@x[...]]` or non-text nodes inside attributes `@x[elem[...]]`. Such ill-formed XMLs are detected at runtime and result in an error.) The second component p maintains the position where the node inhabited in the input tree. In the case of a node that is not from the input tree (i.e. newly constructed by a template), $\bot$ is assigned. We use this positional information only for the evaluation of `visit` expressions, which has to distinguish newly generated nodes from nodes derived from the input tree. In the final output as an XML document, the positional information is dropped.

The conversion function *itl* from a pair of a binary tree t and its node p into the internal form is defined as follows

$$itl(t,p) = (t(p)\texttt{[}itl(t,p_0),\ldots,itl(t,p_k)\texttt{]},p)$$

where $p_i = p.\texttt{l}\overbrace{\texttt{r}\cdots\texttt{r}}^{i}$ (the dot . denotes concatenation) and k is the maximum number such that $p_k \in Pos(t)$.

Interpretation An expression or an expression list is interpreted under an input binary tree t and a first-order variable assignment γ, and denotes a list of internal trees. Concretely, an expression list $E_1 \cdots E_k$ denotes the concatenation of the interpretations of $E_1, E_2, \cdots$, and E_k:

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

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

$$\begin{aligned}
[\![x]\!](t,\gamma) &= [\ itl(t,\gamma(x))\ ]\\
[\![\sigma\texttt{[}EL\texttt{]}]\!](t,\gamma) &= [\ (\sigma\texttt{[}[\![EL]\!](t,\gamma)\texttt{]},\bot)\ ]\\
[\![\{\texttt{gather}\ x\texttt{::}\varphi\texttt{::}EL\}]\!](t,\gamma) &=\\
&concat\ [\ [\![EL]\!](t,\gamma_{x:=p}) \mid p \in Pos(t), \gamma_{x:=p} \vDash \varphi\ ]\\
[\![\{\texttt{visit}\ x\ \texttt{from}\ y\texttt{::}\varphi_1\texttt{::}EL_1\texttt{::}\ldots\texttt{::}\varphi_k\texttt{::}EL_k\}]\!](t,\gamma)&\\
&= vis(Pos(t), itl(t,\gamma(y)))
\end{aligned}$$

where

$$vis(V,(\sigma\texttt{[}IL\texttt{]},p)) =
\begin{cases}
concat\ [vis(V\backslash\{p\},I) \mid I \in [\![EL_1]\!](t,\gamma_{x:=p})] & \text{if } p \in V \text{ and } t,\gamma_{x:=p} \vDash \varphi_1\\
\vdots & \\
concat\ [vis(V\backslash\{p\},I) \mid I \in [\![EL_k]\!](t,\gamma_{x:=p})] & \text{if } p \in V \text{ and } t,\gamma_{x:=p} \vDash \varphi_k\\
[(\sigma\texttt{[}concat\ [vis(V,I) \mid I \in IL]\texttt{]},p)] & \text{otherwise}
\end{cases}$$

In general, more than one case in the definition of vis may be applicable at a time. In that case, the first one is chosen.

The notation $[f(x) \mid x \in List$ (, 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 remaining element and the result list $[f(x_{i_1})\cdots f(x_{i_n})]$ is yielded. The set $Pos(t)$ in `gather`'s semantics 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 transformation templates.

The *vis* function takes two parameters. The first parameter V denotes the set of input nodes that are *not* yet processed in the current evaluation of the `visit` expression. The second parameter (n,p) denotes the node currently visited. If the node is matched by one of the query formulae and has not yet been processed, then the node is replaced by the list of nodes generated from the associated template. Then, again we recursively visit these generated nodes, recording the current node to be already processed. If the node is matched by no query formula, has already been processed, or has newly been generated, then we just recursively go down in the tree.

4. Evaluation Algorithm

This section briefly overviews our evaluation strategy for MTran. We first explain our MSO evaluation consisting of (1) compilation of MSO formulae to tree automata and (2) evaluation of n-ary queries represented by those tree automata. We then sketch how to integrate the query algorithm into our evaluation strategy for the whole language. For lack of space, we only sketch the algorithms, omitting a detailed presentation.

4.1 From MSO to Tree Automata

First of all, we formalize the notion of queries.

Definition 4. An n-ary query for binary tree over Σ is a function q that maps each tree t to a set of n-tuples 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 5. 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

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

where the product $t \times s$ of trees is the 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 each β_i marks the element v_i as 1 and the other elements as 0. 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 where β_i marks no node or more than one node. But such a language defines exactly the same query as the language with all ill-marked trees removed.

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

Definition 6. 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 7. A *run* of a bottom-up deterministic tree automaton on a binary tree t is a mapping $\rho : Pos(t) \to Q$ such that $\rho(v) = \delta(\rho(v.\texttt{l}), \rho(v.\texttt{r}), t(v))$ for each node $v \in Pos(t)$. When $v.\texttt{l}$ or $v.\texttt{r}$ does not belong to the domain $Pos(t)$, we use q_0 instead of $\rho(v.\texttt{l})$ or $\rho(v.\texttt{r})$. The run ρ is called *accepting* if $\rho(\varepsilon) \in F$. Note that a run is uniquely determined from any tree t. A tree is *accepting* when there is an accepting run on it.

Each tree automaton defines the tree language consisting of 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 8. An n-ary query over Σ is *regular* if there exists a bottom-up deterministic tree automaton that defines the query.

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. *[31] 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.*

This equivalence allows us to compile a given MSO formula to an equivalent automaton as the first step of MSO query evaluation. As mentioned in the introduction, although this compilation step is known to take a non-elementary time in the worst case, we can overcome this difficulty simply by employing the MONA system [17]. Section 5 shows our experimental results supporting our claim.

4.2 N-ary Query Algorithm

Definition 5 yields a naive evaluation algorithm for n-ary queries represented by tree automata. That is, for every n-tuple 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 $O(|t|^{n+1})$.

Actually, this high time complexity can be improved if we reduce the set of selection marks to be tested by doing a pre-calculation that determines the exact set of states relevant to accepting runs. Flum, Frick, and Grohe [12] have shown an algorithm for evaluating an n-ary MSO query locating n-tuples of sets of nodes that runs in $O(|t| + |s|)$ time, where $|s|$ is the size of the output. They use a three-pass algorithm to achieve the linearity. The first bottom-up pass calculates, for each node of the input tree, the set of states where a bottom-up run of the automaton reaches for some selection marks. The second top-down pass determines another set of states for each node, namely, the states that may lead to an accepting state at the root node. Finally, the last bottom-up pass collects the result of the query. In the last pass, all selection marks relevant to accepting runs are simultaneously tested in a single traversal of the input tree, by using operations on sets of selection marks. Here, it is crucial to have efficient *union* and *product* operations on the set data structures for ensuring the linear time complexity. For this, they have exploited a linked list with an additional pointer to the last element, which enables constant-time concatenation, for representing sets with efficient operations.

Our algorithm is based on this linear-time algorithm. However, since we only need to query n-tuples of *nodes* in MTran, as oppose to *sets of nodes* in their case, we can specialize the algorithm so as to use a more concise representation. Our approach is, instead of pre-calculating relevant states, to directly execute the third collection phase in conjunction with our *partially lazy evaluation* of set operations. In this, we basically delay set operations like unions and products until actually enumerating the final result, except that we eagerly compute those operations when one of the operands is an empty set. This technique achieves the same time complexity as Flum-Frick-Grohe algorithm. Roughly speaking, the delaying of the operations corresponds to the second pass, which confines the calculation of concrete n-tuples to the runs that may reach accepting states. Also, the eager computation for empty sets corresponds to the first pass, which eliminates the calculation for the runs that never happen for any selection marks.

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

Table 2. Compilation and Execution Time (sec)

4.3 Transformation Template Evaluation

How we can evaluate `gather` or `visit` expressions themselves is clear from the definition of semantics. The issue is, how to integrate the query algorithm introduced in Section 4.2 into the evaluation strategy for whole transformation templates. A naive implementation of the semantics in Section 3.3.2 will evaluate each query as a unary query, by fixing the binding of the free variables other than the one to be queried. However, as we already discussed in Section 1.3, we do not take this strategy. Instead, we evaluate each query expression as an n-ary query (where n is the number of its free variables) *once and for all*. We have further developed a novel optimization technique for n-ary queries exploiting the context information (i.e., the sets of nodes bound in outer templates). The details are omitted from this abstract.

5. Preliminary Performance Evaluation

We show results from our experiments using four examples (two from Section 2 and two from Appendix B). 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. We have measured the execution time using the `time` command, and have taken the average of 10 runs.

Our implementation strategy of transformation templates consists of two steps: (1) compilation of MSO formulae to tree automata using MONA system (Section 4.1) and (2) query evaluation and transformation with the compiled tree automaton (Sections 4.2 and 4.3). We experiment on each step separately, whose result is shown in Table 2. The second column shows the total time spent for compiling all query expressions in each program. Although this step takes hyper-exponential time in the worst case, the experiment shows that, at least for these three examples of XML queries, our strategy yields enough performance. We have then measured the performance of our evaluation algorithm using randomly generated XML documents of different sizes as inputs. The results confirm that most transformations are executed in reasonably practical time even for relatively large inputs. However, the MathML example spends longer time compared to the other examples. The reason seems the following. Recall that our query algorithm runs in $O(|t| + |s|)$ time, where $|s|$ is the size of the query result. The MathML program selects every node in the input for a whole document transformation, which makes $|s|$ quite large.

To demonstrate the efficiency of our template evaluation strategy (Section 4.3), we compare the performances of MTran and traditional XSLT processors (XT [21] and Xalan-C [32]). For benchmark, we wrote, both in MTran and XSLT, a transformation that appends, to the content of each `h2` element, the content of its preceding `h1` element. Figure 7 shows the execution times for the inputs varying the number of `h2` elements from 1000 to 9000. As we

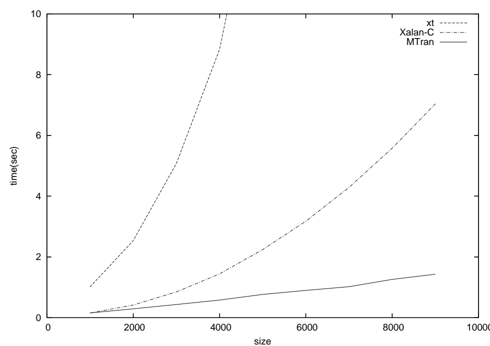

Figure 7. Evaluation Time

explained in Section 4.3, MTran processes the query to select "the `h1` element preceding the current `h2` element" as a binary query, which enables a linear time transformation. On the other hand, XT and Xalan-C evaluate the above query as a unary query and repeat it on each `h2` node, which incur quadratic blow-up as can be seen in the figure.

6. Future Work

The major direction for future work is an addition of static typechecking. For verifying the correctness of XML transformations, many researches [23, 16, 30, 19] have been done in the area of typechecking for regular tree languages. It is an interesting question how those results are applicable to MTran, in particular, to our non-standard semantics of `visit` expressions.

Static type information is useful not only for verification, but also for query optimizations. 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* [4]. We could think of a further improvement of the current evaluation algorithm, using such optimized automata.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

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

[3] A. Berlea and H. Seidl. Binary queries. In *Extreme Markup Languages 2002*, Aug. 2002.

[4] M. Biehl, N. Klarlund, and T. Rauhe. Algorithms for guided tree automata. In *Workshop on Implementing Automata*, pages 6–25, 1996.

[5] S. Bird, Y. Chen, S. Davidson, H. Lee, and Y. 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.

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML™). `http://www.w3.org/XML/`, 2000.

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

[8] J. Clark. XSL Transformations (XSLT), 1999. `http://www.w3.org/TR/xslt`.

[9] J. Clark and S. DeRose. XML path language (XPath). `http://www.w3.org/TR/xpath`, 1999.

[10] J. Clark and M. Murata. RELAX NG. `http://www.relaxng.org`, 2001.

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

[12] J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. *Lecture Notes in Computer Science*, 1973:22–38, 2001.

[13] A. Halevy, Z. Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semantic web applications. In *the 12th Intl. World Wide Web Conf*, pages 556–567, 2003.

[14] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019, pages 89–110. Springer, 1995.

[15] H. Hosoya and B. 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.

[16] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 11–22, Sept. 2000.

[17] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA 1.4. `http://www.brics.dk/mona/`, 1995.

[18] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002.

[19] N. Klarlund, T. Schwentick, and D. Suciu. Xml: Model, schemas, types, logics, and queries. In *Logics for Emerging Applications of Databases*, pages 1–41, 2003.

[20] C. 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.

[21] B. Lindsey and J. Clark. XT version 20051206. `http://www.blnz.com/xt/`, Dec 2005.

[22] S. Maneth and F. Neven. Structured document transformations based on XSL. In *DBPL*, pages 80–98, 1999.

[23] S. Maneth, T. Perst, A. Berlea, and H. Seidl. XML type checking with macro tree transducers. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, pages 283–294, 2005.

[24] Mathematical markup language MathML version 2.0 (second edition). `http://www.w3.org/Math/`, Oct 2003.

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

[26] K. 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.

[27] F. Neven and J. V. 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.

[28] F. Neven and T. Schwentick. Query automata over finite trees. *Theoretical Computer Science*, 275(1-2):633–674, 2002.

[29] J. Niehren, L. Planque, J.-M. Talbot, and S. Tison. N-ary queries by tree automata. In *DBPL*, Aug 2005.

[30] D. Suciu. Typechecking for semistructured data. In *DBPL*, pages 1–20, 2001.

[31] J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, pages 2:57–82, 1968.

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

A. Termination

This section gives a formal proof of termination of a transformation. The following is a key lemma.

Lemma 1. *In each step of recursion in the vis function, the pair (V, I) of arguments always strictly decreases in its 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 the order. 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 being each element of IL, which is the subpart of I. Thus in the arguments (V, I) strictly decreases also in this case. □

Using this, we can 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 easily derived from the induction hypothesis. For the case of `visit` expressions, we prove the termination of the *vis* function, since the semantics of a `visit` expression is actually a evaluation of *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. □

B. Further Examples

B.1 Linguistic Queries

This application is taken from a motivating example of LPath language developed by Bird, Chen, Davidson, Lee, and Zheng [5]. 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>
 <NP>I</NP>
 <VP>
   <V>saw</V>
   <NP>
     <NP>
       <Det>the</Det><Adj>old</Adj><N>man</N>
     </NP>
     <PP>
       <Prep>with</Prep>
       <NP><Det>a</Det><N>dog</N></NP>
     </PP>
   </NP>
 </VP>
 <N>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 said to immediately follow p when p appears immediately after q in some proper analysis [13], 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 shown in Figure 8. In LPath implementation, "immediately follow" 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(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(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(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.

B.2 Relax NG Simplification

RELAX NG specification [10] 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 more sophisticated approach. We take up their "`empty` element" rule as an example. The `empty` element in RELAX NG 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`

```
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 8. Example: Linguistic Queries

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 sufficiently expressive query language, achieving the desired result requires us to really 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 E) =
  all1 x: (x in E <=>
     x in <empty>
   | x in <group>      & all1 y: (x/y => y in E)
   | x in <interleave> & all1 y: (x/y => y in E)
   | x in <choice>     & all1 y: (x/y => y in E)
   | x in <oneOrMore>  & all1 y: (x/y => y in E);

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 9 shows a template implementing it. We assume that the input is a valid RELAX NG schema, and that each `group` or `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 we bring the `empty` child in the beginning, as stated in the simplification rule. Any other node is kept unchanged (which is ensured by the semantics of `visit`).

```
{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 9. `empty` element simplification

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 have to specify that we need to recursively transform `gather`ed elements `y`.

Acknowledgments

We are grateful to Mizuhito Ogawa, Anders Møller, and members of POPL seminar at the University of Tokyo for valuable discussions. We also thank anonymous referees of POPL'06 for useful and encouraging comments. This work was partly supported by Japan Society for the Promotion of Science.