XML Transformation Language Based on Monadic Second-order Logic

Kazuhiro Inaba and Haruo Hosoya

The University of Tokyo
{kinaba,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 can express (1) all regular queries, (2) deep matching without explicit recursion, (3) queries that "don't care" 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 have some of these properties, none satisfies all. In this work, we have designed and implemented a practical XML transformation language, *MTran*, fully exploiting MSO's expressiveness. Based on XSLT-like "select-and-transform" paradigm, we design transformation templates specially suitable for expressing structure-preserving transformation, eliminating the need for explicit recursive calls. Also, we allow nesting of templates for making use of an n-ary query that depends on previously selected $n-1$ nodes. For the implementation, we have developed an efficient evaluation strategy for n-ary MSO queries, consisting of (a) an exploitation of MONA system for the translation from MSO to tree automata and (b) a linear time query evaluation algorithm for tree automata. The latter is similar to Flum-Frick-Grohe algorithm locating n-tuples of *sets of* nodes, except that our query is specialized to querying tuples of *nodes* and employs *partially lazy set operations* for attaining a simpler implementation with a fewer number of tree traversals. Our preliminary experiments confirm 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 [1, 2] and for theoretical models for XML transformation with MSO as a sublanguage for node selection [3, 4]. 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 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 needed 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 [5–7], pattern matches [8, 9], and monadic datalog queries [2] 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 [10], while most of existing path-based node selection languages (including XPath [5], 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 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 [11], XML Schema [12], and RELAX NG [13] 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

	Regularity	No Recursion	Don't care	N-ary
Pattern	$\surd$			$\surd$
Path		$\surd$	$\surd$	
Datalog	$\surd$		$\surd$	
MSO	$\surd$	$\surd$	$\surd$	$\surd$

Table 1. Comparisons between query languages

> "select, from an XHTML document, every `<h2>` node that appears between the current and the next `<h1>` 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 retrieving 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 every `<ul>` element with a `<ul>` parent by the MSO formula "`<ul>/x & x in <ul>`" and then transform each selected `<ul>` element 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 [15]:

```
<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>
```

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

Fig. 1. A transformation using binary queries

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 (`<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 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 generalizes this approach 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 outer templates. For example, see the program in Figure 1. This program converts a document representing a one-to-many mapping, e.g.,

```
<o2m><map><name>Hello</name><value>1</value><value>2</value></map>
     <map><name>World</name><value>3</value><value>4</value></map></o2m>
```

to another representing a many-to-many mapping:

```
<m2m><P>Hello,1</P> <P>Hello,2</P> <P>World,3</P> <P>World,4</P></m2m>
```

Notably, in the inner-most selection condition for the variable `v`, which appears inside the scope that selects 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 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 of nodes 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 [16] 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 MSO formulae by tree automata with binary decision diagrams and is experimentally shown to work quickly for large formulae even of dozens of kilobytes. 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 [18]. 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 [3] and its generalization TL [4] 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.

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 [19]. Then, Flum, Frick, and Grohe have solved the general case by showing a linear-time, three-pass algorithm [18]. 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 [2, 16, 20], though each of these has made orthogonal efforts in either implementation or theory. 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.

Outline In the rest of this paper, Section 2 gives some slightly bigger example programs for illustrating the strength of MSO and MTran. Then, Section 3 introduces the syntax and semantics. Section 4 briefly overviews our evaluation strategy and Section 5 shows the results of our preliminary performance evaluation.

2 Examples

As it is stated in the introduction, MSO queries has four strength, namely, regularity, no-recursion, don't-care semantics, and natural N-ary queries. In this section, we give several example templates written in MTran and demonstrate its expressiveness.

```
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[
   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} ]}
```

Fig. 2. Example: Table of Contents

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>
  <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 appears after the node `a` and before any nodes `x` in the set `A` that occur 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.

The main template expression has the following structure. Let us 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

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/>
  <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>
```

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:

```
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 <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}} }]
```

Fig. 3. Example: MathML Conversion

```
 <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>
```

The first macro `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 macro `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 `need_paren(x)` is 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 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 markups in the input document.

2.3 Linguistic Queries

This application is taken from a motivating example of LPath language developed by Bird, Chen, Davidson, Lee, and Zheng [21]. 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 [22], 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 4. 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.

```
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} ] ]
```

Fig. 4. Example: Linguistic Queries

2.4 Relax NG Simplification

RELAX NG specification [13] 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` 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 5 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} ] }
```

Fig. 5. `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`.

3 Language Definition

3.1 Binary Trees and XML representation

Throughout this paper, we assume a fixed, finite alphabet Σ where each $\sigma \in \Sigma$ is called symbol. 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 symbol $t(p) \in \Sigma$ assigned to p the *label* of p. The empty sequence node ε is called the *root* of t.

We handle input XML as binary trees, using a well-known encoding of unranked trees (trees whose each node has an arbitrary number of child nodes) 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 children of the corresponding node in the binary tree. In addition, a real XML document has three types of node—element, 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 their belonging element node. [1]

3.2 Query Expressions

This section describes MTran's query sublanguage based on MSO. We first present the core syntax and semantics and then introduce some syntax sugars.

Assume a set of first-order variables, ranged over by x, and a set of second-order variables, ranged over by X. The following defines the syntax for first-order terms p, second-order terms S, and MSO formulae φ.

$$p ::= x \mid \mathtt{root} \qquad\qquad S ::= X \mid \sigma$$

$$\begin{aligned}\varphi ::= {} & p \ \mathtt{in}\ S \mid p\ \mathtt{=}\ p \mid S\ \mathtt{=}\ S \mid \mathtt{\tilde{}}\varphi \mid \varphi\mathtt{\&}\varphi \mid \varphi\mathtt{|}\varphi \mid \varphi\mathtt{=>}\varphi \mid \mathtt{ex1}\ x\mathtt{:}\varphi \mid \mathtt{all1}\ x\mathtt{:}\varphi \\ & \mid \mathtt{ex2}\ X\mathtt{:}\varphi \mid \mathtt{all2}\ X\mathtt{:}\varphi \mid \mathtt{firstChild(}p\mathtt{,}p\mathtt{)} \mid \mathtt{nextSibling(}p\mathtt{,}p\mathtt{)}\end{aligned}$$

A 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, and 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 formulae p `in` S means that the node denoted by p belongs to the set denoted by S. The operators `=`, `~`, `&`, `|`, and `=>` are the standard operators for equality, negation, conjunction, disjunction, and implication. The constructs `ex1` and

[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`.

`all1` (`ex2` and `all2`) are existential and universal quantifications over first-order (second-order, respectively) variables. The last two constructs are the primitive predicates to relate tree nodes. That is, `firstChild(`p`,`q`)` (`secondChild(`p`,`q`)`) holds if and only if q is the first child (the next sibling, respectively) of node p.

Formally, the semantics of our query expression is defined as follows:

Definition 1. 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:

$$\begin{aligned} \Gamma[X] &= \Gamma(X) \\ \Gamma[\sigma] &= \{p \mid t(p) = \sigma\} \end{aligned}$$

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:

$$\begin{array}{ll}
t, \gamma, \Gamma \models p_1\texttt{=}p_2 & \iff \gamma[p_1] = \gamma[p_2] \\
t, \gamma, \Gamma \models S_1\texttt{=}S_2 & \iff \Gamma[S_1] = \Gamma[S_2] \\
t, \gamma, \Gamma \models p \texttt{ in } S & \iff \gamma[p] \in \Gamma[S] \\
t, \gamma, \Gamma \models \texttt{~}\varphi & \iff t, \gamma, \Gamma \not\models \varphi \\
t, \gamma, \Gamma \models \varphi_1\texttt{\&}\varphi_2 & \iff t, \gamma, \Gamma \models \varphi_1 \text{ and } t, \gamma, \Gamma \models \varphi_2 \\
t, \gamma, \Gamma \models \varphi_1\texttt{|}\varphi_2 & \iff t, \gamma, \Gamma \models \varphi_1 \text{ or } t, \gamma, \Gamma \models \varphi_2 \\
t, \gamma, \Gamma \models \varphi_1\texttt{=>}\varphi_2 & \iff t, \gamma, \Gamma \models \varphi_1 \text{ implies } t, \gamma, \Gamma \models \varphi_2 \\
t, \gamma, \Gamma \models \texttt{ex1 } x\texttt{:}\varphi & \iff \text{for some } a \in \mathit{Pos}(t) \;\; t, \gamma_{x:=a}, \Gamma \models \varphi \\
t, \gamma, \Gamma \models \texttt{all1 } x\texttt{:}\varphi & \iff \text{for all } a \in \mathit{Pos}(t) \;\; t, \gamma_{x:=a}, \Gamma \models \varphi \\
t, \gamma, \Gamma \models \texttt{ex2 } X\texttt{:}\varphi & \iff \text{for some } A \subseteq \mathit{Pos}(t) \;\; t, \gamma, \Gamma_{X:=A} \models \varphi \\
t, \gamma, \Gamma \models \texttt{all2 } X\texttt{:}\varphi & \iff \text{for all } A \subseteq \mathit{Pos}(t) \;\; t, \gamma, \Gamma_{X:=A} \models \varphi \\
t, \gamma, \Gamma \models \texttt{firstChild(}p_1\texttt{,}p_2\texttt{)} & \iff \gamma[p_1].\texttt{l} = \gamma[p_2] \\
t, \gamma, \Gamma \models \texttt{nextSibling(}p_1\texttt{,}p_2\texttt{)} & \iff \gamma[p_1].\texttt{r} = \gamma[p_2]
\end{array}$$

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 $\{\texttt{l},\texttt{r}\}^*$.

Programmers can define macros in the form

$$\texttt{pred } m\texttt{(}V\texttt{,}\cdots\texttt{,}V\texttt{) = }\varphi\texttt{;}$$

where m is a *macro name* and each V has either the form "`var1` x" or "`var2` X", i.e., a first- or a second-order *parameter* declaration. Accordingly, we augment the syntax of formulae with the macro-call form m`(`T`,` $\cdots$ `,`T`)` where each T is a variable whose order matches the corresponding variable declaration in m's definition. A macro call is expanded to its definition whose each parameter variable is replaced by the corresponding supplied argument. 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.

Our core syntax has only two primitive relations on tree nodes, `firstChild` and `nextSibling`, reflecting our binary-tree encoding of XML. For convenience, however, we provide more compact notations based on the *unranked* view of the input XML tree and an XPath-like syntax sugar. We extend the syntax of formulae with the form $UDUD \cdots DU$, where D is *path delimiters* either of the form `/` or `//`, and U is either a first-order term p or a second-order term S. 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. 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>`. Connecting three or more terms by path delimiters means conjunction. For example, `x/y/z` stands for `x/y & y/z`.

Another frequently used primitive is the pre-order relation (often called *document order* relation) among tree nodes. This relation is also MSO-expressible and provided as a short-hand: $p < p$.

3.3 Transformation Templates

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

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

is evaluated to the list of nodes:

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

Note that we gather all nodes that match the query regardless of their inclusion relations. When we want only the outermost nodes, we explicitly specify so:

```
{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. 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 details. 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 `X` tag:

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

By this template, the same input document used in the above example of `gather` is transformed to the result:

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

Each `A` or `C` node is unmatched, for which we simply copy it and proceed to its child nodes. Then, each `B` node is matched, for which we evaluate the subtemplate `X[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 `X` node; we simply copy 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 `X[x]` in the same way as above.

To justify the above design choices, the reason for copying 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—this design is again for our intention to avoid any recursion. Finally, copying 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.*

Syntax and Semantics A program consists of a list of user-defined macros and a (template) expression, where *expressions E* and *expression lists EL* are defined as follows.

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

The phrase "`from` y" can be omitted from a `visit` expression when y is `root`.

MTran internally handles two different forms of XML representation. The first is the binary encoding of XML trees described in Section 3.1, which is used in our query algorithm to represent input trees. The other 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 non-text nodes inside attributes `@x[@y[...]]`. 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

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

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 $p_k \in \mathit{Pos}(t)$.

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:

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

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:

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

where

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

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 $[f(x) \mid x \in \mathit{List}$ (, 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 remaining element and the result list $[f(x_{i_1}) \cdots f(x_{i_n})]$ is yielded. The set $\mathit{Pos}(t)$ in `gather`'s semantics is treated as a list of nodes ordered by the document order. All formulae 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 denotes the node currently visited. If the node is matched by one of the query formulae and has not yet been processed, then it 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 describes our evaluation strategy for MTran. In this, efficient evaluation of MSO query expressions is particularly critical and therefore explained in detail. Our MSO evaluation consists of usual two steps: (1) compilation of MSO formulae to tree automata and (2) evaluation of n-ary queries represented by those tree automata. Below, we first review known facts on tree automata, then formalize the above two steps, and finally briefly explain how to integrate the query algorithm into our evaluation strategy for the whole language.

4.1 From MSO to Tree Automata

First of all, we formalize the notion of queries. An n-ary query for binary trees 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. 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

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

where the product $t \times s$ of trees is 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. In our treatment, such a language defines exactly the same query as the language with all ill-marked trees removed.

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. A binary tree t is *accepted* by a bottom-up deterministic tree automaton when there is a mapping $\rho : \mathit{Pos}(t) \to Q$ such that $\rho(\varepsilon) \in F$ and $\rho(v) = \delta(\rho(v.\mathtt{l}), \rho(v.\mathtt{r}), t(v))$ 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 $\rho(v.\mathtt{r})$.

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. 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 $\varphi(\boldsymbol{x})$ whose free variables are $\boldsymbol{x} = (x_1, \ldots, x_n)$ defines a query $q(t) = \{\boldsymbol{v} \in \mathit{Pos}(t)^n \mid t \vDash \varphi(\boldsymbol{v})\}$. It is well-known [10] that there is an exact correspondence between MSO and tree automata. That is, for every MSO formula with n free variables, there exists a bottom-up deterministic tree automaton over $\Sigma \times \mathbb{B}^n$ that defines the equivalent query. Also, for every bottom-up deterministic tree automaton 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 MONA [17]. Section 5 shows our experimental results supporting our claim.

4.2 *N*-ary Query Algorithm

This section is devoted to our n-ary query algorithm. First, Section 4.2 introduces a basic algorithm that is essentially the same as the binary query algorithm proposed by Berlea and Seidl, except that we reformalize it for general n-ary queries. Section 4.2 presents our improvements to the algorithm.

Basic Algorithm Definition 4.1 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 $O(|t|^{n+1})$.

This high time complexity can be improved by sharing intermediate results among calculations of the runs corresponding to the n-tuples. Specifically, we assign a set $m(v,q)$ of n-tuples of nodes to each pair of a node v of the tree t and a state q of the automaton. Intuitively, $m(v,q)$ is the set of tuples satisfying 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 the automaton. For example, in a ternary query, the fact that $(\mathtt{rl},\mathtt{r},\perp) \in m(\mathtt{r},q_1)$ 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. Thus, a tuple here may have a $\perp$ component to mean "no selection yet." A marking run can be calculated in a bottom-up fashion where each $m(v,q)$ is calculated from that of each child of v. For example, suppose $\delta^{-1}(q) = \{((t(v),00),q_{L1},q_{R1}),((t(v),01),q_{L2},q_{R2}))\}$. We can then calculate $m(v,q)$ as follows.

$$\begin{gathered} m(v.\mathtt{l},q_{L1}) * m(v.\mathtt{r},q_{R1}) * \{(\perp,\perp)\} \\ \cup m(v.\mathtt{l},q_{L2}) * m(v.\mathtt{r},q_{R2}) * \{(\perp,v)\} \end{gathered}$$

Here, the operator $*$ is a kind of "product" operation that combines two sets of tuples, defined as follows:

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

For example:

$$\{(\mathtt{l},\perp),(\perp,\mathtt{l})\} * \{(\mathtt{r},\perp)\} = \{(\mathtt{r},\mathtt{l})\}$$

The left operand means that "(the node l is selected as the first element) or (the node l is selected as the second element)," while the right operand means that "r is selected as the second element." There is only one possible way to collect up these two conditions, namely, "the node r is selected as the first element and the node l is selected as the second element." This is what the asterisk product $*$ calculates.

According to the definition, we can calculate $S * T$, by picking up each pair of tuples (s,t) where $s \in S$ and $t \in T$, and constructing the new tuple u. This strategy, however, has an inefficiency. Every resulting tuple $u \in S * T$ comes from a pair (s,t) such that non-bottom elements of s and t do not overlap (for example, $(\mathtt{r},\mathtt{l})$ is derived from $((\perp,\mathtt{l}),(\mathtt{r},\perp))$). Other pairs like $((\mathtt{l},\perp),(\mathtt{r},\perp))$ are just discarded. This inefficiency is cured by introducing *types* over tuples and classifying the tuples in $m(v,q)$ by the types. This saving is crucial for guaranteeing the linear-time complexity of our algorithm, as we discuss later.

Formally, we define marking runs as follows. We define each $m(v,q)$ as a disjoint union of sets $m_s(v,q)$, where $s \in \mathbb{B}^n$. A sequence s of bits indicates non-$\perp$ elements of the tuples in $m_s(v,q)$. That is, for $(u_1,\ldots,u_n) \in m_{s_1\ldots s_n}(v,q)$, we have $u_i \neq \perp$ if and only if $s_i = 1$ for every i. In other words, s indicates the *type* of tuples contained in $m_s(v,q)$.

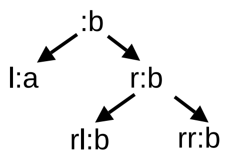

Fig. 6. Example Tree

Definition 2. 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 $m = \{m_s | \, s \in \mathbb{B}^n\}$. 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 $v \notin Pos(t)$,

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

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

$$\begin{aligned} m_s(v,q) = \bigcup \{ m_l(v.\mathtt{l}, q_L) * m_r(v.\mathtt{r}, q_R) * \{sing(v,c)\} \mid \\ q_L, q_R \in Q, \quad l, r, c \in \mathbb{B}^n, \\ \delta(q_L, q_R, (t(v), c)) = q, \\ (l, r, c) \in sub(s) \} \end{aligned} \tag{1}$$

Here, the tuple $sing(v, c)$ is defined as the tuple $(u_1, \ldots, u_n)$ where $u_i = v$ 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 triples (l, r, c) such that for all i, if $s_i = 1$ then exactly one of l_i, r_i, and c_i is 1, and otherwise $l_i = r_i = c_i = 0$.

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

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

Example Let us illustrate the evaluation algorithm by the following example. Consider the tree t over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ depicted in Figure 6. (For 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:

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

$$\delta(q_i, q_j, x) = q_3 \quad \text{for any other cases}$$

This automaton defines a binary query that selects all pairs of a node and its right child both labeled `b`, i.e. $(\varepsilon, \texttt{r})$ and $(\texttt{r}, \texttt{rr})$ 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 recognizes 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 node is selected for the same component 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. However, this is actually not necessary since the query defined by an automaton is not affected by whether or not the automaton accepts ill-marked trees, as we already stated.

A marking run on the above automaton consists of four functions m_{00}, m_{01}, m_{10}, and m_{11} of type $\{\texttt{l}, \texttt{r}\}^* \times Q \to 2^{(\{Pos(t) \cup \{\perp\}\})^2}$. We first calculate those functions for the leaf nodes $\{\texttt{l}, \texttt{rl}, \texttt{rr}\}$. Since for $p \notin Pos(t)$ the set $m_{00}(p, q_0)$ is defined to be $\{(\perp, \perp)\}$ and $m_s(p, q) = \emptyset$ for any other s, q, we only need to calculate the formula (1) with $q_L = q_0$ and $q_R = q_0$. Thus, for each leaf node e we have:

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

The concrete values of $m_s(e, q)$ are shown in the following tables:

m_{00}	q_0	q_1	q_2	q_3
`l`:a	$\{(\perp,\perp)\}$	$\emptyset$	$\emptyset$	$\emptyset$
`rl`:b	$\{(\perp,\perp)\}$	$\emptyset$	$\emptyset$	$\emptyset$
`rr`:b	$\{(\perp,\perp)\}$	$\emptyset$	$\emptyset$	$\emptyset$

m_{01}	q_0	q_1	q_2	q_3
`l`:a	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\perp,\texttt{l})\}$
`rl`:b	$\emptyset$	$\{(\perp,\texttt{rl})\}$	$\emptyset$	$\emptyset$
`rr`:b	$\emptyset$	$\{(\perp,\texttt{rr})\}$	$\emptyset$	$\emptyset$

m_{10}	q_0	q_1	q_2	q_3
`l`:a	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\texttt{l},\perp)\}$
`rl`:b	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\texttt{rl},\perp)\}$
`rr`:b	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\texttt{rr},\perp)\}$

m_{11}	q_0	q_1	q_2	q_3
`l`:a	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\texttt{l},\texttt{l})\}$
`rl`:b	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\texttt{rl},\texttt{rl})\}$
`rr`:b	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\texttt{rr},\texttt{rr})\}$

For example, the entry $m_{01}(\texttt{rr}, q_1) = \{(\perp, \texttt{rr})\}$ 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 sets that are actually required for obtaining the result of the query. The first set is $m_{11}(\texttt{r}, q_2)$:

$$m_{11}(\texttt{r}, q_2) = \bigcup\{m_l(\texttt{rl}, q_L) * m_r(\texttt{rr}, q_R) * \{sing(\texttt{r}, c)\} \mid$$

$$\delta(q_L, q_R, (\texttt{b}, c)) = q_2, \quad (l, r, c) = sub(11)\}$$

Seeing the definition of the transition function δ, there are three choice for q_L, q_R, and c to satisfy the condition $\delta(q_L, q_R, (\texttt{b}, c)) = q_2$. 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 $m_l(\texttt{rl}, q_2)$ and $m_r(\texttt{rr}, q_2)$ are empty for any l and r (see the tables). Thus, we have:

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

The second interesting set is $m_{01}(\texttt{r}, q_1)$. We can calculate it in the same way. This time, there is only one choice for q_L, q_R, and c:

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

Finally, we show the calculation of $m_{11}(\varepsilon, q_2)$, which is the result of the query. Just like $m_{11}(\texttt{r}, q_2)$, we have three choice for q_L, q_R, and c, of which two are actually significant.

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

The parts "$\ldots$" involve products with $\emptyset$ and therefore are omitted.

Complexity A direct bottom-up calculation of m takes $O(|t|^{n+1})$ time, since we need to calculate $m_{1\ldots1}(v, q)$, which can grow up to $O(|t|^n)$ in the worst case, for all $|t|$ nodes.

This complexity can be improved if we reduce the set of selection marks to be tested by pre-calculating the exact set of states relevant to accepting runs. Flum, Frick, and Grohe [18] 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. The last pass is essentially the same algorithm as we have shown in the preceding subsection, except that it calculates $m(v, q)$ only at the state that are determined to affects the query result by the preceding two passes. 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 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, 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. The advantage over Flum-Frick-Grohe is that partially lazy set operations are extremely easy to implement, yet we only need to traverse the input tree once (enumeration of the final results traverses the tree of lazy set operations, whose size is proportional to the output size).

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 written in O'Caml [23]. Only two operations, union and product, are necessary. The type of non-empty sets is:

```
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 `node option` type is a type that can be either `None` ($\bot$) 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. The type of (possibly empty) sets of nodes is:

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

To deal with emptiness, empty and unit sets are strictly distinguished from non-empty sets. Union and product operations are:

```
  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)
```

An empty set and a unit set are dealt as special cases. That is, when one of the arguments is an empty set (a unit set, respectively), the set operation is explicitly performed at that point. 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.

```
(* returns: node 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"

(* returns: node 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` calculates the asterisk product of two sets of tuples exactly as in Definition 2, where `prod_one` takes the product of two singleton sets. Each function appends the result to the accumulator argument `acc`. Since we introduced the *types* of tuples in the definition, we do not need to consider the case that both `a` and `b` in `prod_one` has a non-bottom head. The function `to_list` is the main routine for converting a symbolic operation tree to an actual set of tuples.

```
(* node_set -> node 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 [])
```

Since this function is assumed to be applied to a tree representing the result set of a more-than-zero-ary query, it does not consider the case of `UnitSet`, which means that no node is selected. For a `Singleton`, we just create a singleton list. For a `Product`, we use the `prod` function already defined. For a `Union`, we simply concatenate the two lists. To see why this is correct, let us review the Definition 2. Our query algorithm always takes the union of two sets in the form

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

where $m_l(v.\mathtt{l}, q_L)$ 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 those 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 our simple method with concatenation is correct.

To see why eager computation of emptiness is required, let us see the case of `Product` in the `to_list` function. If we do not distinguish empty sets during the construction of operation trees, either `a` or `b` may be an operation tree representing an empty set. If that happens, for example, `a` is evaluated to an empty set, then the result of the computation `(f b [])` is just disposed by the `prod` function, and does not affect the final query result. By the eager evaluation of emptiness, we eliminate all such emergence of unnecessary computations. This is essential for obtaining a linear-time complexity, as shown in the sequel.

Complexity Using these lazy set operations, we can calculate all operation trees corresponding to m_s including $m_{1\ldots1}$, by simply following Definition 2. 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 $O(3^n|Q|^2t)$ 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 1. *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 number of `Product` nodes on each path is bounded by 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 constant factors, and thus we obtain the complexity $O(t+|s|)$. The proof of Lemma 1 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 $O(3^{k-1}n|a| + 3^{k-1}n|b| + n|a||b|) < O(3^k n|a||b|) = O(3^k n|s|)$

Example Running our algorithm on the previous example in Section 4.2, 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, we get the desired result: `[[r; rr]; [ε; r]]`.

Two features of the algorithm has eliminated unnecessary calculation. The first point is its laziness. By delaying the actual construction of the sets, we have avoided the computation for the sets like $m_{10}(\mathtt{l}, q_3)$ or $m_{01}(\mathtt{r}, q_3)$ that are not contained in the query result. The second point is its eager evaluation for empty sets. By this we have avoided the computation for the sets like $m_{01}(\mathtt{rl}, q_1)$ or $m_{00}(\mathtt{rr}, q_1)$ that are taken products with an empty set.

4.3 Transformation Template Evaluation

How we can evaluate `gather` or `visit` expressions themselves is clear from the definition of their semantics. The issue is how to integrate the query algorithm introduced in Section 4.2 to the evaluation strategy for whole transformation templates. A naive implementation of the semantics in Section 3.3 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). 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 as stated in Section 4.3, and obtain a list $[(v_1^{\mathtt{x}}, v_1^{\mathtt{y}}) \ldots (v_s^{\mathtt{x}}, v_s^{\mathtt{y}})]$ of 2-tuples satisfying the expression. This list is referred to by a 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 specifically for querying will be the sum of the times spent by all queries each evaluated once.

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 $O(3^n(|Q|^2 t + n|s|))$ 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 the result from inner binary query. Only the pairs $(v_i^{\mathtt{x}}, v_i^{\mathtt{y}})$ satisfying $\phi(v_i^{\mathtt{x}})$ are 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 the compiled automaton may increase. The parameter $|Q|$ affects the total time complexity in a quadratic order and therefore cannot be ignored.

We have developed a slightly modified version of n-ary query algorithm to remedy this problem. In this, we user the results of outer queries during the evaluation of the inner query. Using this, we can reduce the query result like the rewriting approach while keeping the size of the automaton unchanged. Concretely, we first execute the outermost unary query ϕ(x) in the same way as explained in the previous subsection, and obtain a list of results $R_{\mathtt{x}} = [u_1^{\mathtt{x}} \ldots u_k^{\mathtt{x}}]$. We then execute the next inner query ψ(x,y) with one modification to the definition of a marking run:

$$m_s(v,q) = \bigcup \{ m_l(v.\mathtt{l}, q_L) * m_r(v.\mathtt{r}, q_R) * \{sing(v,c)\} \mid q_L, q_R \in Q, \quad l, r, c \in \mathbb{B}^n,$$

$$\begin{aligned}&\delta(q_L, q_R, (t(v), c)) = q,\\&(l, r, c) \in sub(s),\\&\underline{c_1 = 1 \implies v \in R_{\mathtt{x}}}\}\end{aligned}$$

assuming that the first bit corresponds to the variable x (If there are further inner queries, we repeat the above recursively.) In this modified definition, we select each node v as a marked node for variable `x` only when v is contained in $R_{\mathtt{x}}$. Therefore, the final result obtained in this way 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 (the underlined part) does not incur any additional time complexity blowup, since by processing outer queries earlier than inner ones, we can always have each $R_{\mathtt{x}}$ already constructed and its inclusion can be determined in constant time using any appropriate data structure such as a hash-table.

Complexity Using the fact that each MSO query can be evaluated in linear time, we have proved the following theorem, assuring that the evaluation of a fixed MTran transformation template terminates in polynomial time with respect to the size of the input.

Theorem 1. *An MTran program is evaluated in $O(d|t|^d + |s|)$ time, where d is the nesting-depth of the program, and $|t|$ and $|s|$ is the size of the input and the output, respectively.*

In order to achieve this complexity order, we internally represent the output tree as a DAG (directed acyclic graph). Since a sub-template evaluated under the same variable binding always returns the same output tree, our representation shares such output trees to avoid redundant calculation. The proof of the theorem is done by induction on the structure of transformation templates, showing both the evaluation time according to the semantics and the size of the output DAG of each subtemplate fit in the complexity order. The details are omitted.

5 Preliminary Performance Evaluation

We show results from our experiments using four examples from Section 2. All benchmarks are run on Windows XP SP2 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++ and compiled it by GNU C++ Compiler. We have measured the execution time using the `time` command, and have taken the average of 10 runs.

We separately experiment on two steps of our implementation strategy, 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 examples of XML queries, our strategy yields enough performance. We have then measured the performance of our evaluation algorithm using randomly generated

	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)

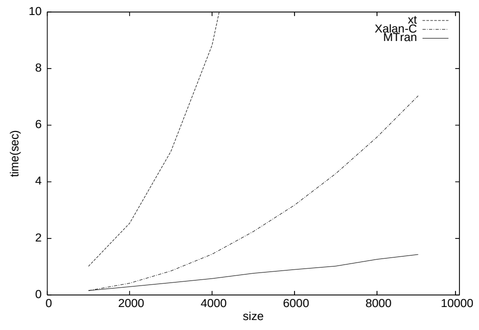

Fig. 7. Evaluation Time

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 [24] and Xalan-C [25]). 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 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.

References

1. Neven, F., Schwentick, T.: Query automata over finite trees. Theoretical Computer Science **275**(1-2) (2002) 633–674
2. Koch, C.: Efficient processing of expressive node-selecting queries on xml data in secondary storage: A tree automata-based approach. In: VLDB. (2003) 249–260

3. Maneth, S., Neven, F.: Structured document transformations based on XSL. In: DBPL. (1999) 80–98
4. Maneth, S., Perst, T., Berlea, A., Seidl, H.: XML type checking with macro tree transducers. In: Proceedings of Symposium on Principles of Database Systems (PODS). (2005)
5. Clark, J., DeRose, S.: XML path language (XPath). `http://www.w3.org/TR/xpath` (1999)
6. Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J.L.: The Lorel query language for semistructured data. International Journal on Digital Libraries **1**(1) (1997) 68–88
7. Brüggemann-Klein, A., Wood, D.: Caterpillars: A context specification technique. Markup Languages **2**(1) (2000) 81–106
8. Hosoya, H., Pierce, B.C.: Regular expression pattern matching for XML. Journal of Functional Programming **13**(6) (2002) 961–1004 Short version appeared in Proceedings of The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 67–80, 2001.
9. Benzaken, V., Castagna, G., Frisch, A.: CDuce: An XML-centric general-purpose language. In: Proceedings of the International Conference on Functional Programming (ICFP). (2003) 51–63
10. Thatcher, J.W., Wright, J.B.: Generalized finite automata with an application to a decision problem of second-order logic (1968)
11. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E.: Extensible markup language (XML$^{\text{TM}}$). `http://www.w3.org/XML/` (2000)
12. Fallside, D.C.: XML Schema Part 0: Primer, W3C Recommendation. `http://www.w3.org/TR/xmlschema-0/` (2001)
13. Clark, J., Murata, M.: RELAX NG. `http://www.relaxng.org` (2001)
14. Henriksen, J.G., Jensen, J.L., Jørgensen, M.E., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 1019., Springer (1995) 89–110
15. Clark, J.: XSL Transformations (XSLT) (1999) `http://www.w3.org/TR/xslt`.
16. Berlea, A., Seidl, H.: Binary queries. In: Extreme Markup Languages 2002. (August 2002)
17. Klarlund, N., Møller, A., Schwartzbach, M.I.: MONA 1.4. `http://www.brics.dk/mona/` (1995)
18. Flum, J., Frick, M., Grohe, M.: Query evaluation via tree-decompositions. Lecture Notes in Computer Science **1973** (2001) 22–??
19. Neven, F., den Bussche, J.V.: 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, ACM press (1998) 11–17
20. Niehren, J., Planque, L., Talbot, J.M., Tison, S.: N-ary queries by tree automata. In: DBPL. (Aug 2005)
21. Bird, S., Chen, Y., Davidson, S., Lee, H., Zheng, Y.: Extending XPath to support linguistic queries. In: Informal Proceedings of the Workshop on Programming Language Technologies for XML PLAN-X 2005. (2005) 35 – 46
22. Halevy, A., Ives, Z., Mork, P., Tatarinov, I.: Piazza: Data management infrastructure for semantic web applications (2003)
23. Leroy, X., Doligez, D., Garrigue, J., Vouillon, J., Rémy, D.: The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/` (1996)

24. Lindsey, B., Clark, J.: XT version 20051206. `http://www.blnz.com/xt/` (Dec 2005)
25. Foundation, T.A.S.: Xalan C++ 1.10. `http://xml.apache.org/xalan-c/` (Feb 2004)

Acknowledgements

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 and by the Okawa Foundation for Information and Telecommunications.