Expressive Power of XSLT*

Geert Jan Bex[1] Sebastian Maneth[2] Frank Neven[1,†]

[1]Limburgs Universitair Centrum,
Universitaire Campus,
Dept. WNI, Infolab,
B-3590 Diepenbeek, Belgium,
E-mail: {gjb,frank.neven}@luc.ac.be

[2]Leiden University, LIACS,
PO Box 9512, 2300 RA Leiden,
The Netherlands
E-mail: maneth@liacs.nl

Abstract

The recent extension of XSL by variables and passing of data values between template rules has generated a powerful XML query language: XSLT (eXtensible Style sheet Language Transformations). Indeed, the expressive power of XSLT is beyond that of most other XML query languages. This is shown by first defining a powerful fragment of XSLT (together with formal syntax and semantics) and then investigating the expressive power of this language. Furthermore, important properties such as termination and closure under composition are considered for this fragment of XSLT.

1 Introduction

XSL [CD] is a current W3C [Con] proposal for an XML extensible stylesheet language. Its original primary role was to allow users to write transformations of XML to HTML, thus describing the presentation of XML documents. Nowadays, many people use XSL as their basic tool for XML to XML transformations which renders XSL into an XML query language. It has been noted by the database community [DFF+99a, ABS99], though, that the transformations XSL can express are rather limited. For instance, XSL does not have joins or skolem-functions (and, hence cannot do sophisticated grouping of output data). In other words, XSL lacks the most basic property any query language should have: it is not relationally complete.

However, as the language is still under development, some features have changed over time. Recently various extensions were added to the language [Cla99]. The most apparent ones being the addition of *variables* and *parameter passing* between template rules. We show that these additions, together with the use of modes (which are actually states as used in finite state machines and which were already defined in earlier versions of XSL) render XSL into a powerful query language. Indeed, XSL not only becomes relationally complete, but it can do explicit grouping (with or without skolem functions), it can simulate regular path expressions, and it can simulate most other current XML query languages.

*A preliminary version of this paper was presented at the 1st International Conference on Computational Logic, London, July, 2000.

†Post-doctoral researcher of the Fund for Scientific Research, Flanders

Together with the addition of the new features, XSL was (essentially) split into three parts: XSL Transformations (XSLT) [Cla99], XPath [Cla] and an XML vocabulary for specifying formatting semantics. In this paper we will ignore the latter as it is only concerned with presentation of the output. XPath contains the description of XSL's associated pattern language, while the transformation section defines the real transformation language. To emphasize that we are focusing on the *transformation* part, with the new features, we refer to XSL by XSLT in the rest of this paper.

The main source for the definition of XSLT is its specification [Cla99] which is a long technical document rather difficult to read, especially when one only wants an impression of how the language works or what it is capable of. To remedy this, we define an abstract formal model of XSLT incorporating most of its features, but all of those which are necessary to simulate, say, XML-QL. The purpose of this model is two-fold:

(*i*) the clean and formal semantics provides the necessary mathematical model for studying properties of XSLT;

(*ii*) our formal model abstracts from the actual syntax of XSLT and emphasizes its transformational features in such a way that the interested reader can get a feeling of the language and what it is capable of.

We want to emphasize that we do *not* provide a model for all of XSLT. For instance, we excluded for-loops, and variables can only be instantiated by data values (not by result tree fragments or node-sets). The idea is that we want to use variables as a look-ahead or to fetch data values occurring 'far' from the current node. The resulting language is, hence, not Turing complete. The most important observation will be that the defined language is much more expressive than the previous version of XSL [DFF$^+$99a, ABS99].[1]

We use this model to gain some insight in XSLT. With respect to its expressiveness, we consider the k-pebble tree transducer of Milo, Suciu, and Vianu [MSV00]. These were defined, in the context of type checking, to capture the expressiveness of most current XML query languages including XML-QL [DFF$^+$99a], XQL [Rob99], Lorel [AQM$^+$97], StruQL [FFK$^+$98], UnQL [BDHD96] and the previous version of XSL. Their model does not take value equation into account (needed for joins, for instance) but can easily be modified to do so. By showing that XSLT can simulate this model, we already informally indicate the increased expressiveness of XSLT. Towards a more concrete simulation of an actual language, we define a valid abstraction of XML-QL and show that it can be simulated in XSLT.

Next, we obtain that XSLT can compute all unary monadic second-order (MSO) structural properties. In brief, MSO logic is first-order (FO) logic extended with set quantification. It is an expressive and versatile logic: on trees, for instance, it captures many robust formalisms, like regular tree languages [Tho97a], query automata [NS99], finite-valued attribute grammars [NVdB98, Nev99b], etcetera. By structural patterns we mean MSO without joins, that is, we cannot check whether the values of two attributes are the same (see Section 4 for details). In fact, Neven and Schwentick [NS00] showed that, already w.r.t. structural patterns, MSO logic is more expressive than FO logic extended with various kinds of regular path expressions. Thus, as most current XML

[1] In previous work we defined a formal model for the version of XSL not incorporating data values [MN99].

query languages are based on FO logic extended with regular path expressions, this already indicates that XSLT cannot be simulated by, say, XML-QL.

On the negative side, we show that the new additions make the termination problem undecidable. In fact, deciding termination for XSLT without data values is complete for EXPTIME [MN99]. Further, we show that XSLT programs are not closed under composition.

Although the present paper shows that the recent additions to XSLT render it into a powerful transformation language, we are rather hesitant, however, to accept or promote XSLT as the standard XML *query* language. Our main objection is that XSLT is much too procedural for a query language and therefore maybe too difficult for the average user. On the other hand, as indicated by its widespread use, XSLT is highly adequate for the simple transformations it was intended for (recall that XSL was originally intended just for XML to HTML transformations). These simple XSLT programs are typical one pass transformations from the root to the leaves of the document. Performing joins and doing complicated grouping operations seem to require XSLT programs to traverse the input document many times in several directions and are therefore more difficult to write, especially for people with little programming experience.

The remainder of this paper is structured as follows. In Section 2, we introduce the important features of XSLT by means of two examples. In Section 3, we define our formal model. Finally, in Section 4, we study the properties of our model.

We refer the interested reader to [Bex], were it is shown how the XML-QL queries of [DFF$^+$99a] can be expressed in actual XSLT.

2 New Features of XSLT by Example

A basic XSLT program is a collection of *template rules* where each such rule consists of

- a *matching pattern*,
- a *mode* (which indicates the (finite) state the computation is in), and
- a *template* (see, for example, the program in Figure 2).

Input and output documents of an XSLT program P are unranked trees, with the notational convention that a left bracket is represented by `<xxx>` (where `xxx` is a label) and the corresponding right bracket is represented by `</xxx>`; thus, `<a>b</a>` represents the tree $a(b)$. The computation of P on an input document t starts at the root node of t in the starting mode[2] and proceeds roughly as follows. When the computation arrives at a node, say u, in a certain mode, say q, the program tries to find a template rule with mode q whose matching pattern matches u.[3] If it finds such a template rule, the program executes the corresponding template. The latter usually instructs XSLT to produce some XML output and at various positions in this XML output to select lists of nodes for further processing (we refer to patterns that select nodes for further processing as *selecting patterns*). Each of these selected nodes are then processed independently as

[2]Actually, modes are optional, but for convenience we assume every template has one and there is a start mode.

[3]Usually, and in all our examples, such a matching pattern only refers to the label of the current node. In fact, we show in Section 4 that such patterns suffice.

before. Finally, the documents that are constructed by these subprocesses are inserted at the positions where the subprocesses were generated.

```
<!DOCTYPE organization [
    <!ELEMENT organization  group+ topmgrs>
    <!ELEMENT topmgrs       employee+>
    <!ELEMENT group         (mgr group+) | employee+>
    <!ELEMENT mgr           employee>
    <!ATTLIST employee      id  ID  #REQUIRED>
    <!ATTLIST group         id  ID  #REQUIRED>
]>
```

Figure 1: A DTD describing an organization.

To illustrate the new features of XSLT we use the DTD in Figure 1. It describes an organization as a sequence of groups together with a list of top managers. Each group has an ID, consists of a manager and a list of other groups, or just consists of a list of employees. For simplicity we identify employees by their ID. The XSLT program in Figure 2 computes pairs (e_1, e_2) of employees, where e_1 is a top manager different from `Bill` and is a direct or indirect manager of e_2. Pairs are encoded simply by a `pair` element with attributes `topmgrID` and `employeeID` (cf. Figure 4).

On the face of it, the program just makes a join between the list of top managers and the group managers, that is, the ones occurring in the top manager list and the ones occurring as a manager of a group. However, it does so in a rather direct and procedural way. In brief, the XSLT program starts by applying the first template rule at the root in mode start. This rule selects each top manager (in mode `selecttopmgr`). In particular, the pattern `/organization/topmgr/employee` is matched against the current node which is labeled by `organization` and then selects all `employee` children of all `topmgr` children (the symbol / means 'child of'). For each selected employee (in mode `selecttopmgr`), the second template rule is applied which stores the employee's ID, say e_1, in the variable `varID` and verifies, by using the latter, whether e_1 is different from Bill. If so, it selects all the descendants of the group manager who have an ID e_1 (in mode `display`). In particular, the selection pattern in the second template rule says 'select all employees that are descendants of a group that itself is a child of a group whose manager has the same ID as the one stored in the variable `varID` (the symbol // means 'descendant of'; the expression between the brackets '[...]' is a filter on group elements). In this latter selection the XSLT program passes the ID e_1 along as a parameter. Next, for each employee e_2 selected by the latter selection, the program outputs an element `pair` with attribute values e_1 and e_2 for the attributes `topmgrID` and `employeeID`, respectively.

The program in Figure 2 is not the most compact way to specify the desired query in XSLT, but it nicely illustrates three important new features of XSLT: modes, variables, and passing of data values. We discuss these briefly:

(i) Modes enable XSLT to act differently upon arrival at the same element types. For instance, as described above, when our program arrives at an employee element, its action depends on the actual mode, `select` or `display`, this element was selected in.

```
<xsl:template match="organization" mode="start">
    <result>
        <xsl:apply-templates select="/organization/topmgr/employee"
                    mode="selecttopmgr"/>
    </result>
</xsl:template>

<xsl:template match="employee" mode="selecttopmgr">
    <xsl:variable name="varID">
        <xsl:value-of select="@id"/>
    </xsl:variable>
    <xsl:if test="$varID != 'Bill'">
        <xsl:apply-templates mode="display"
             select="//group[mgr/employee[@id=$varID]]/group//employee">
             <xsl:with-param name="varID" select="$varID"/>
        </xsl:apply-templates>
   </xsl:if>
</xsl:template>

<xsl:template match="employee" mode="display">
    <xsl:param name="varID"/>
    <pair>
        <xsl:attribute name="topmgrID">
            <xsl:value-of select="$varID"/>
        </xsl:attribute>
        <xsl:attribute name="employeeID">
            <xsl:value-of select="@id"/>
        </xsl:attribute>
    </pair>
</xsl:template>
```

Figure 2: An XSLT program computing the query of Section 2.

```
<organization>
    <group id="HR">
        <mgr><employee id="Bill"/></mgr>
        <group id="HR-prod">
            <mgr><employee id="Edna"/></mgr>
            <group id="HR-prod-empl">
                <employee id="Kate"/>
                <employee id="Ronald"/>
            </group>
        </group>
        <group id="HR-QA">
            <mgr><employee id="John"/></mgr>
            <group id="HR-QA-empl">
                <employee id="Jane"/>
                <employee id="Jake"/>
            </group>
        </group>
    </group>
    <topmgr>
        <employee id="Bill"/>
        <employee id="John"/>
    </topmgr>
</organization>
```

Figure 3: An XML document conforming to the DTD of Figure 1.

```
<result>
    <pair topmgrID="John" employeeID="Jane"/>
    <pair topmgrID="John" employeeID="Jake"/>
</result>
```

Figure 4: The output of the XSLT program of Figure 2 on the document of Figure 3.

(*ii*) Variables can be used for two purposes. The most apparent one, which is illustrated by the above query, is that they allow to perform joins between data values. A less apparent application is to use them as a 'look-ahead'. In Figure 5, we give a fragment of an XSLT program evaluating a binary tree, representing a Boolean circuit, to its truth value. Essentially, the use of variables allows for a bottom-up computation. In brief, when arriving at an or-labeled node, the program returns the correct truth value based upon the truth values of the first and second subtree. The restriction to binary trees is just for expository purposes. In fact, we show in Section 4 how to simulate arbitrary bottom-up tree automata over unranked trees.

(*iii*) Passing of data values to other template rules can be crucial for performing joins if the items to be joined are 'far' apart. Moreover, when node IDs are present in the XML document,[4] we can use this mechanism to place 'pebbles' on the input document which enables us to do complicated grouping operations.

(*iv*) A feature not highlighted by the example in Figure 2 is that XPath expressions are no longer restricted to select descendants of the current node. That is, one can also select siblings and ancestors.

It are exactly these four features which render XSLT into a quite powerful transformation language.

If we would consider XSLT from a formal language point of view, we could say that XSLT corresponds to *tree-walking tree transducers with registers* (working over unranked trees). In fact, the distinction between ranked and unranked trees can be dropped if the transducer model is capable of realizing the encoding of unranked trees by ranked ones, and the corresponding decoding. For example, it is straightforward to show that the MSO transducer of [Cou94] can realize the standard encoding of unranked trees (see, e.g., [PQ68], or the proof of Theorem 18 in [MN99]) and the corresponding decoding. (For similar reasons the authors of [MSV00] have drawn their attention to the ranked case only.). Indeed, modes and the possibility to select all neighbors of the current node allows to explore the whole tree as a tree-walking automaton; the output facility renders it into a transducer; and, the variables together with parameter passing can be seen as registers for data values. This connection will become clearer in Section 4.1.

3 A Formal Model for XSLT

We provide XSLT with an abstract formal syntax. First, we restrict matching patterns to test only the label of the current node (as is already the case in Figure 2). This is no

[4]If not, XSL is capable of generating them itself (see Section 4).

```
<xsl:template match="or">
  <xsl:variable name="arg1">
    <xsl:apply-templates select="./*[1]"/>
  </xsl:variable>
  <xsl:variable name="arg2">
    <xsl:apply-templates select="./*[2]"/>
  </xsl:variable>
  <xsl:choose>
    <xsl:when test="($arg1 = 'false') and ($arg2 = 'false')">
      <xsl:value-of select="'false'"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="'true'"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Figure 5: The fragment of an XSLT program evaluating tree-structured Boolean circuits that takes care of *or*-nodes.

restriction, as Theorem 4.3 shows that we can test many properties of the current node in the body of a template rule. Second, we divide a template rule into two parts: the *variable definition part* and the *construction part*. Variables can only be assigned with data values. We assume an infinite set $\mathbf{D}$ and the only allowed operation on elements of $\mathbf{D}$ is the equality test. Further, a variable can only be defined as the value of some attribute of the current node or by an XSLT apply-templates statement that will return exactly one data value. We will refer to such special templates as *selection template rules*. In the construction part of the template rule, the actual output is defined relative to some conditions on the values of the variables, the parameters, the attribute values of the current node, and possibly whether the current node is the root, a leaf, or the first or last child of its parent.

3.1 Trees and forests

We start with the necessary definitions regarding trees and forests over a finite alphabet Σ (the symbols in Σ correspond to the element names of the XML document seen as a tree). To use these trees as adequate abstractions of actual XML documents, we extend them with attributes that take values from an infinite domain $\mathbf{D} = \{d_1, d_2, \dots\}$.

The set of Σ-trees, denoted by $\mathcal{T}_\Sigma$, is inductively defined as follows:

- every $\sigma \in \Sigma$ is a Σ-tree
- if $\sigma \in \Sigma$ and $f_1, \dots, f_n \in \mathcal{T}_\Sigma$, $n \geq 1$ then $\sigma(f_1 \cdots f_n)$ is a Σ-tree.

A Σ-*forest* is a (possibly empty) sequence $f_1, \dots, f_n$, $n \geq 0$ of Σ-trees. The empty Σ-forest is denoted by ε. The set of all Σ-forests is denoted by $\mathcal{F}_\Sigma$. Note that there is no a priori bound on the number of children of a node in a Σ-tree; such trees are therefore *unranked*. In the following, whenever we say tree or forest, we always mean Σ-tree and Σ-forest.

The reason that we consider forests is that even if XSLT is used for tree-to-tree transformations only, we sometimes need to specify template rules which construct forests (see, e.g., Example 3.15).

For every forest $f \in \mathcal{F}_\Sigma$, the *set of nodes of* f, denoted by $\mathrm{Nodes}(f)$, is the subset of $\mathbb{N}^*$ inductively defined as follows:

- if $f = \sigma(t_1 \cdots t_n)$ with $\sigma \in \Sigma$, $n \geq 0$, and $t_1, \ldots, t_n \in \mathcal{T}_\Sigma$, then $\mathrm{Nodes}(f) = \{\varepsilon\} \cup \{iu \mid i \in \{1, \ldots, n\}, u \in \mathrm{Nodes}(t_i)\}$; and
- if $f = t_1 \cdots t_n$, then $\mathrm{Nodes}(f) = \{iu \mid i \in \{1, \ldots, n\}, u \in \mathrm{Nodes}(t_i)\}$.

Thus, for a tree the node ε represents its root and ui represents the i-th child of u. Further, for a forest the node iu represents the node u of the i-th tree in the forest. For every forest $f \in \mathcal{F}_\Sigma$ and every node u of f, the *label* of f at u is denoted by $f[u]$. The *substitution* of the forest f' at node u in f is denoted by $f[u \leftarrow f']$.

Next, we add XML attributes to the above defined attributes. To this end, for the rest of the paper, we fix a finite set of attributes A.

Definition 3.1 An *attributed forest with domain* S is a pair $(f, (\lambda_a^f)_{a \in A})$ with $f \in \mathcal{F}_\Sigma$ and where for each $a \in A$, $\lambda_a^f : \mathrm{Nodes}(f) \mapsto S$ is a (partial) function assigning a value in S to each node of f. The set of all attributed forests with domain S, is denoted by $\mathcal{F}_\Sigma^S$.

The notion of label and substitution carry over in the straightforward manner.

For S we will usually take $\mathbf{D}$. However, to create output in template rules we will use attributed forests over $\mathbf{D} \cup \{x_1, \ldots, x_n\}$ where the variables refer to those defined by the variable defining part of the template. Of course, in real XML documents, usually, not all element types have the same set of attributes. Obviously, this is just a convenient and not a necessary restriction. In an analogous way one can define the set of attributed trees, denoted by $\mathcal{T}_\Sigma^S$. For a set B, $\mathcal{F}_\Sigma^S(B)$ denotes the set of attributed forests f over $\Sigma \cup B$ such that symbols of B may only appear at the leaves of f. Below, B will be the set of apply template expressions.

In the next example we elaborate a bit on the connection between actual XML documents and attributed trees.

Example 3.2 Clearly, in an XML document all text elements can be replaced by appropriate attributes. In terms of the corresponding tree this means that all nodes (including the leaves) are labeled by bracket labels (such as `beer` or `color` in this example). Thus, the use of attributes to replace text elements solves the technicality of how to represent `<a>bcd</a>` as an unranked tree (either $a(bcd)$ or $a(b, c, d)$).

For instance, for our purpose, the XML document

```
<beer name="Leffe trippel">
    <alc> 8.7 </alc>
    <description>
      <color> blonde </color>
      <taste> bitter </taste>
    </description>
</beer>
```

can be represented by, e.g.,

```
<beer name="Leffe trippel", alc="8.7">
    <description color="blonde", taste="bitter">
</beer>.
```

The latter can easily be represented by the attributed tree

$$t := \text{beer}(\text{description}),$$

where $\lambda^t_{\text{name}}(\varepsilon) =$ "Leffe trippel", $\lambda^t_{\text{alc}}(\varepsilon) =$ "8.7", $\lambda^t_{\text{color}}(1) =$ "blonde", and $\lambda^t_{\text{taste}}(1) =$ "bitter".

Here, $\mathbf{D}$ is assumed to contain the subset {"Leffe trippel", "8.7", "blonde", "bitter"}. $\square$

In our formal model we abstract away from a particular selection pattern language. Recall that XSLT uses the pattern language described in XPath [Cla] (see [Wad99], for a formal semantics). Patterns can be rather involved as illustrated by the second template rule in Figure 2 where the pattern depends on the value of the variable `varID`. In addition, patterns can also be moving instructions like select parent, left sibling, right sibling, or first child. Actually, the proof of Theorem 4.1 indicates that such local selections are already enough to simulate many existing XML query languages.

In the following, we assume an infinite set of variables $\mathcal{X}$. We define a pattern over the variables $X \subseteq \mathcal{X}$ as a function

$$(\mathcal{T}^{\mathbf{D}}_{\Sigma} \times (X \mapsto \mathbf{D})) \mapsto (\mathbb{N}^* \mapsto 2^{\mathbb{N}^*})$$

and denote the set of all patterns over X by $\mathcal{P}^X_{\Sigma}$. The idea is as follows. Let p be a pattern, t be a tree, and γ be a variable assignment (for the variables in p). Then $p(t,\gamma)(u)$ is the set of selected nodes when the pattern is applied at node u.

Example 3.3 Let p be the XPath pattern

$$\text{beer[name[@val=}x\text{]]/alc},$$

where x is a variable. Let γ map x to "Leffe trippel" and let t be the tree of Example 3.2. Then $p(t,\gamma)(\varepsilon) = \{2\}$, because the node 2 of t is an alc-labeled son of a beer-labeled node with attribute name="Leffe trippel". $\square$

3.2 Syntax

Definition 3.4 An *XSLT program* is a tuple $P = (\Sigma, \Delta, M, \text{start}, R)$, where

- Σ is a finite alphabet of *input symbols*;
- Δ is a finite alphabet of *output symbols*;
- M is a finite set of *modes*, partitioned into the sets M^c and M^s of *construction* and *selection modes*, respectively;
- start $\in M^c$ is the *start* mode; and

- R is a finite set of *rules*, partitioned into the sets R^c and R^s of *construction* and *selection template rules* (to be defined below), respectively.

As mentioned at the beginning of Section 3, we distinguish between two types of template rules: *constructing* and *selecting* ones. The former are used to create output. So, the result of applying these is a forest. The latter are used to fetch data values. So, each one returns exactly one domain element. The mode will determine the nature of the template: constructing or selecting.

Definition 3.5 A *construction apply-templates-expression* r (at-expression for short) is of the form $q(p, \bar{z})$, where $q \in M^c$, p is a pattern and $\bar{z}$ is a possibly empty sequence of variables in $\mathcal{X}$ and domain elements in $\mathbf{D}$. A *selection at-expression* is defined as a constructing one with the restriction that $q \in M^s$ and p only selects single nodes, that is, for every tree t, each assignment of variables γ, and each node u of t, $p(t,\gamma)(u)$ is a singleton set. We denote the set of construction (selection) at-expression by $\mathcal{AT}^{\rm c}$ ($\mathcal{AT}^{\rm s}$).

For instance, the apply-templates expression in the second template of Figure 2 is a constructing one and corresponds in our model to the expression

$$\text{display}(p, \text{varID}),$$

with p the pattern

```
//group[mgr/employee[@id=varID]]/group//employee.
```

Note that application of this pattern eventually leads to the generation of a `pair` element. So the expression is constructing in the sense that it eventually produces output.

Definition 3.6 An *attribute expression* is an expression of the form $a(.)$ where a is an attribute. An *atomic test* is one of the following: (i) an expression of the form $x = y$ where x and y are attribute expressions, variables, or domain elements; or, (ii) an expression of the form root, leaf, first-child, or last-child. Finally, a *test* is a Boolean combination of atomic tests.

During a computation the expression $a(.)$ will evaluate to the value of the attribute a of the current node. Further, root, leaf, first-child, last-child evaluate to true whenever the current node is the root, a leaf, the first or the last child of its parent. Selection template rules are defined next. Recall that they output one domain element.

Definition 3.7 *Let $q \in M^s$ and $\sigma \in \Sigma$.* A *(q,σ)-selection template rule* is of the form

template $q(\sigma, x_1, \ldots, x_n)$
    vardef
        $y_1 := r_1; \ldots ; y_m := r_m$
    return
        if c_1 then z_1; $\ldots$; if c_k then z_k
end

where $n, m \geq 0$, $k \geq 1$; and,

- all x's and y's are variables (the former are parameters while the latter are local variables);
- each $r_i \in \mathcal{AT}^s$ or is an attribute expression; further, if $r_i \in \mathcal{AT}^s$ then every variable occurring in it is among $y_1, \ldots, y_{i-1}, x_1, \ldots, x_n$;
- all c_i are tests containing only variables in $X := \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$; and
- every z_i is a domain element, a variable in X, or a selecting at-expression with the restriction that all variables occurring in it should belong to X.

Intuitively, this template is evaluated as follows. First, the values of the variables y_1, $\ldots$, y_n, are defined. Such a value can be an attribute value of the current node or can be defined by invoking an at-rule that will compute the desired data value. The return value then is determined by z_i where c_i is the first test that evaluates to true.

Definition 3.8 Let $q \in M^c$ and $\sigma \in \Sigma$. A (q, σ)-*construction template rule* is of the same form as a selection rule only now each z_i is a forest in $\mathcal{F}_\Sigma^{\mathbf{D}\cup\mathcal{X}}(\mathcal{AT}^c)$ (recall that these are forests where attributes take values in $\mathbf{D} \cup \mathcal{X}$ and where leaves may be labeled with constructing at-expression) with the restriction that each variable occurring in an at-expression occurring at a leaf of z_i should be in X.

The only difference with Definition 3.7 is that the z_i's are forests in $\mathcal{F}_\Sigma^{\mathbf{D}\cup\mathcal{X}}(\mathcal{AT}^c)$. These determine the output and at which node, in which state, and with which parameters the computation process should continue. The precise semantics is given in the next section.

Remark 3.9 To keep the model total and deterministic we require the existence of exactly one (q, σ)-template rule for each mode and each σ. Moreover, by default the empty forest will be outputted whenever none of the test $c_1, \ldots, c_k$ evaluates to true (cf. Definition 3.12). To ensure that an XSLT program generates tree-to-tree translations, we require that each z_i in a (start, σ)-construction template rule is a tree (rather than a forest). $\square$

Remark 3.10 Constructing template rules specify $\mathcal{F}_\Sigma^{\mathbf{D}\cup\mathcal{X}}(\mathcal{AT}^c)$ forests. Recall that such a forest is actually a pair $(f, (\lambda_a^f)_{a\in A})$. We will use the following more convenient arrow notation. By

$$\sigma(\sigma_1[a \to d_1, b \to d_2]\ \sigma_2[a \to d_1]),$$

we mean the tree where the root has no attributes (more precisely, all attributes are undefined); the attributes a and b of the first child get the values d_1 and d_2, respectively; and, the attribute a of the second child gets the value d_1.

When we want to copy the current node together with the values of all its attributes we use the keyword *self*. This could be simulated by first defining the variables $x_a := a(.)$ for all $a \in A$ and then writing $\sigma[(a \leftarrow x_a)_{a\in A}]$, where σ is the label of the current node. $\square$

Example 3.11 We illustrate the above by translating the program in Figure 2 into our syntax. The patterns p_1 and p_2 refer to the patterns in the first and second template rule, respectively. In the second template rule display(p_2,varID) is the tree consisting of one node labeled with display(p_2,varID); recall that ε denotes the empty forest. In the last rule, `pair`[`topmgr`$\rightarrow$varID; `employeeID`$\rightarrow$myID] denotes the tree t consisting of one node where $\lambda^t_{\texttt{topmgrID}}(\varepsilon) :=$ varID and $\lambda^t_{\texttt{employeeID}}(\varepsilon) :=$ myID. For readability, we omitted the test 'if true then'. All modes are constructing. For clarity, we use a box to indicate at-expressions.

template start(organization)
    return
        `result`($\boxed{\text{selecttopmgr}(p_1)}$)
end;

template selecttopmgr(employee)
    vardef
        varID := id(.);
    return
        if varID $\neq$ Bill then $\boxed{\text{display}(p_2,\text{varID})}$;
        if varID = Bill then ε
end;

template display(employee,varID)
    vardef
        myID := id(.);
    return
        `pair`[`topmgrID`$\rightarrow$varID; `employeeID`$\rightarrow$myID]
end;

Note that the rule "if varID = Bill then ε" is not needed, according to Remark 3.9. $\square$

3.3 Semantics

Let $\bar{w}$ consist of a sequence of variables of $\mathcal{X}$ and domain elements. For a function $\gamma : \mathcal{X} \mapsto \mathbf{D}$, we denote by $\bar{w}[\gamma]$ the sequence of domain elements obtained from $\bar{w}$ by replacing each occurrence of the variable x in $\bar{w}$ by $\gamma(x)$. By $x_1\backslash d_1, \ldots, x_n\backslash d_n$, or $\bar{x}\backslash\bar{d}$ for short, we denote the function that maps each x_i to d_i.

We next define the semantics of an XSLT program P on a tree t. A *local configuration* is an element of

$$LC(t) := \text{Nodes}(t) \times (M^c \cup M^s) \times \mathbf{D}^*.$$

Note that here we will use the notation $d_1, \ldots, d_n$ for an element in $\mathbf{D}^*$, that is, a comma-separated list. Intuitively, $\theta := (u, q, d_1, \ldots, d_n) \in LC(t)$ means that the program has selected node u in mode q with values $\bar{d}$ as parameters. A local configuration is selecting (constructing) if its associated state is selecting (constructing). Define $LC^s(t)$ ($LC^c(t)$) as the set of selecting (constructing) local configurations. We define two rewrite relations. The first one determines the rewriting w.r.t. selection, the second one w.r.t. construction. For a binary relation R, we denote by R^* its transitive closure.

Definition 3.12 The *selecting rewriting relation induced by P on t*, denoted by $\rightarrow_{P,t}$, is the binary relation on $LC^s(t) \cup \mathbf{D}$ defined as follows. For $\theta = (u, q, d_1, \ldots, d_n) \in LC^s(t)$ and $\alpha \in LC^s(t) \cup \mathbf{D}$,

$$\theta \rightarrow_{P,t} \alpha$$

if

- the (q, σ)-selection template rule has n local variables for $t[u] = \sigma$; further, let the (q, σ)-template rule be of the form as specified in Definition 3.7, where each r_i is $q_i(p_i, \bar{x}'_i)$ or the attribute expression $a_i(.)$;
- there are $e_1, \ldots, e_m \in \mathbf{D}$ such that
  - if r_i is an at-expression, then $(v_i, q_i, \bar{x}'_i[\gamma_i]) \rightarrow^*_{P,t} e_i$, where γ_i maps each x_j to d_j, for $j = 1, \ldots, n$, and y_j to e_j for $j = 1, \ldots, i-1$, and v_i is the node selected by p_i, that is, $p_i(t, \gamma_i)(u) = \{v_i\}$; or
  - if r_i is an attribute expression, then $e_i := \lambda^t_{a_i}(u)$;
- c_i is the first condition that evaluates to true by interpreting each y_j by e_j, x_j by d_j, $a(.)$ by $\lambda^t_a(u)$, and root, leaf, first-child, last-child by true iff u is the root, a leaf, the first or the last child of its parent, respectively;
  - if z_i is a constant, a variable, or an attribute expression then $\alpha \in \mathbf{D}$ and it equals the corresponding value;
  - if z_i is a selecting at-expression $q'(p, \bar{w})$, then $\alpha \in LC^s(t)$ and $\alpha = (v, q', \bar{w}[\bar{x}\backslash\bar{d}, \bar{y}\backslash\bar{e}])$ where v is the node selected by p, that is, $p(t, [\bar{x}\backslash\bar{d}, \bar{y}\backslash\bar{e}])(u) := \{v\}$.

Definition 3.13 The *constructing rewriting relation induced by P on t*, denoted by $\Rightarrow_{P,t}$, is the binary relation on $\mathcal{F}_\Sigma(LC^c(t))$ (recall that these are the Σ-forests where leaves may be labeled with constructing local configurations) defined as follows. For $\xi, \xi' \in \mathcal{F}_\Sigma(LC^c(t))$,

$$\xi \Rightarrow \xi',$$

if there is a leaf node $v \in \text{Nodes}(\xi)$ with $\xi[v] = (u, q, d_1, \ldots, d_n) \in LC^c(t)$ such that

- the (q, σ)-construction template rule has n local variables for $t[u] = \sigma$; further, let the (q, σ)-template rule be of the form as specified in Definition 3.8; recall that there each z_j is forest in $\mathcal{F}^{\mathbf{D}\cup\mathcal{X}}_\Sigma(\mathcal{AT}^c)$, that is, a forest where attributes take values in $\mathbf{D} \cup \mathcal{X}$ and where leaves may be labeled with constructing at-expressions;
- there are $e_1, \ldots, e_m \in \mathbf{D}$ and a c_i that fulfill the same conditions as in Definition 3.12;
- $\xi' = \xi[v \leftarrow f]$, where f is the forest obtained from z_i by replacing
  - every occurrence of a y_j and a x_j as the value of some attribute by the data values e_j and d_j, respectively;
  - every occurrence $q'(p', \bar{w})$ of an at-expression at a leave of z_i by the forest

    $$(u_1, q', \bar{w}[\bar{x}\backslash\bar{d}, \bar{y}\backslash\bar{e}]) \cdots (u_\ell, q', \bar{w}[\bar{x}\backslash\bar{d}, \bar{y}\backslash\bar{e}]),$$

    where $p'(t, [\bar{x}\backslash\bar{d}, \bar{y}\backslash\bar{e}])(u) = \{u_1, \ldots, u_\ell\}$ and $u_1 \prec \ldots \prec u_\ell$ (here $\prec$ denotes pre-order).

The *initial* local configuration of t is defined as $\theta_{\text{start}}(t) := (\varepsilon, \text{start}, \varepsilon)$, i.e., it is the tree consisting of the single (root) node labeled $(\varepsilon, \text{start}, \varepsilon)$.

Definition 3.14 The transformation *realized* by P, is the (partial) function $\tau_P : \mathcal{T}_\Sigma^{\mathbf{D}} \to \mathcal{T}_\Delta^{\mathbf{D}}$ with $\tau_P(t) = s$ if $t \in \mathcal{T}_\Sigma^{\mathbf{D}}$ and there is an $s \in \mathcal{T}_\Delta^{\mathbf{D}}$ such that $\theta_{\text{start}}(t) \Rightarrow_{P,t}^{*} s$.

Note that the above is well-defined. By Remark 3.9 and the definition of $\to$ and $\Rightarrow$, the transformation is deterministic and generates only trees.

We illustrate the above definitions by an example. In the above we did not instantiate the concrete pattern language. In the examples we assume that it at least contains the functionality of XPath, the pattern language of XSLT. Therefore, we will assume that the following selection patterns are available: self, first, last, right, left, and parent, which select the current node itself, the first child, the last child, the right sibling, the left sibling, and the parent of the current node, respectively.

Example 3.15 Let $P_1 = (\Sigma_1, \Delta_1, M_1^c, \text{start}, R_1)$ be the XSLT program where $\Sigma_1 = \{a\}$, $\Delta_1 = \{b, c\}$, $M_1^c = \{\text{double}, \text{start}\}$, and R_1 consists of the two rules

template start(a)
    return
        $b(\boxed{\text{double(self), double(self)}})$
end;

and

template double(a)
    return
        if leaf then c;
        if $\neg$leaf then $\boxed{\text{double(first), double(first)}}$
end;

Note that in this example we made use of constructing a forest in a rule (namely the forest "double(first), double(first)"), even though the translation will be a tree-to-tree translation. This program transforms a monadic tree consisting of n a's (that is a tree where every node has rank at most one) into the tree $b(c, \cdots, c)$ with 2^n c's. A part of the derivation is illustrated in Figure 6. Essentially, every a is doubled at each transition. Only when reaching a leaf, a c is being output.

Let $P_2 = (\Sigma_2, \Delta_2, M_2^c, \text{start}, R_2)$ be the XSLT program where $\Sigma_2 = \{b, c\}$, $\Delta_2 = \{a\}$, $M_2^c = \{\text{copy}, \text{start}\}$, and R_2 consists of the two rules

template start(b)
    return
        $\boxed{\text{copy(first)}}$
end;

template copy(c)
    return
        if $\neg$last-child then $a(\boxed{\text{copy(right)}})$;
        if last-child then a
end;

This program transforms a tree $b(c \cdots c)$ with $m \geq 1$ c's into a monadic tree of height m consisting only of a's. In brief, the program jumps to the first c without outputting anything. Then if visits every sibling from left to right, each time outputting an a.

The above programs will be used in the proof of Theorem 4.5 to show that XSLT is not closed under composition. □

$$
\begin{array}{lll}
\Theta_{\text{start}(a(a))} & = & (\varepsilon, \text{start}, \varepsilon) \\
(\varepsilon, \text{start}, \varepsilon) & \Rightarrow_{P_1, a(a)} & b((\varepsilon, \text{double}, \varepsilon), (\varepsilon, \text{double}, \varepsilon)) \\
 & \Rightarrow^*_{P_1, a(a)} & b((1, \text{double}, \varepsilon), (1, \text{double}, \varepsilon), (1, \text{double}, \varepsilon), (1, \text{double}, \varepsilon)) \\
 & \Rightarrow^*_{P_1, a(a)} & b(c, c, c, c)
\end{array}
$$

Figure 6: Derivation of the program P_1 on input $a(a)$

We conclude this section with some remarks. We note that XSLT does not make the explicit distinction between constructing and selecting template rules, or even, between the variable definition part and the constructing part of a template rule. However, we feel that by making this explicit, programming becomes more structured. On the other hand, we did not incorporate everything XSLT has to offer. For instance, we refrained from including for-loops. Nevertheless, we indicate in the next section that we have captured a powerful fragment capable of simulating most existing XML query languages and even more. With the aid of the example in Section 2, we invite the reader to check that any program in our abstract syntax can be readily translated into actual XSLT.

4 Properties

We show that XSLT is quite expressive in the sense that it can simulate most other current XML query languages and that it can express all join-free MSO patterns. The latter even implies that XSLT is strictly more expressive than other XML query languages. On the negative side we obtain that deciding termination is undecidable and that XSLT is not closed under composition.

4.1 k-pebble transducers

To study decidability of type checking, Milo, Suciu, and Vianu [MSV00] defined the k-pebble tree transducer as a formalism capturing the expressiveness of most existing XML query languages. Such transducers transform binary trees into other binary trees.[5]

[5]When proving properties of XML transformations, restricting to binary trees is usually sufficient as unranked trees can be coded by ranked ones; cf. the remark at the end of Section 2. However, we seek a formal model for XSLT which works directly on unranked trees, that is, on the original input.

Next, we informally describe such deterministic transducers with *equality tests on data values.*

The k-pebble deterministic tree transducer uses up to k pebbles to mark certain nodes in the tree. Transitions are determined in a unique way by the current node symbol, the current state (or mode), the presence/absence of the various pebbles on the current node, and equality tests on the attribute values of the nodes the pebbles are located on. Pebbles are ordered and numbered from 1 to k. The machine can place pebbles on the root, move them around, and remove them (actually, the use of the pebbles is restricted by a stack discipline which ensures that the model does not become too powerful, that is, accepts non-tree-regular languages). There are move transitions and output transitions. Move transitions are of the following kind: move-to-parent, move-to-first-child, move-to-last-child, move-to-left-sibling, move-to-right-sibling, remain-and-change-state, place-new-pebble, and pick-current-pebble. There are two kinds of output transitions. A binary output symbol σ, possibly with attributes defined by an attribute value of a pebbled node, and spawns two computation branches that compute, independently of each other, the left and the right subtree of σ. Both branches inherit the positions of all pebbles on the input and do not communicate with each other, that is, each branch moves the pebbles independently of the other. A nullary output α generates an α-labeled leaf and the computation halts.

It should be clear that, apart from the pebbles, the above described model is extremely close to XSLT: XSLT is equipped with modes (states), can do local movements and the simple output transitions. Under the assumption that each node has a unique id, XSLT can also simulate pebbles. Indeed, we just use k variables x_1 up to x_k, where at each time instance x_i contains the id of the node on which the i-th pebble is located.

Actually, this assumption is not necessary as XSLT is equipped with the function generate-id(.) which generates a unique id for the current node. Furthermore, this id only depends on the current node, that is, when invoked for the same node several times it will return the same value. So there is no need to store the node id's; they can be computed on demand.

The above discussion immediately leads to the next theorem.

Theorem 4.1 *XSLT can simulate k-pebble deterministic tree transducers with equality tests on data values.*

We point out that non-deterministic tree transducers can be simulated by giving a non-deterministic semantics to XSLT in the obvious way.

Although Theorem 4.1 provides a better understanding of the expressiveness of XSLT, it does so in an indirect way. Therefore, we define next an abstraction of an XML query language and show how to simulate it in XSLT.

4.2 XML-QL

Although there is no generally accepted transformation language for structured documents, several ones have emerged during the last years, including XML-QL [DFF$^+$99a], XSLT [Cla99], and XQL [Rob99], which are specifically for XML, and Lorel [AQM$^+$97], StruQL [FFK$^+$98], and UnQL [BDHD96], for the semi-structured data model. As argued by Fernandez, Siméon, and Wadler [FSW99], XML queries roughly consists of a *pattern* clause and a *constructor* clause. The purpose of the pattern language is to identify the different parts of the document that have to be combined to obtain the output document. The constructing part, on the other hand, indicates how the selected parts should be assembled.

Such queries can, for instance, be written as

$$\texttt{WHERE}\ \varphi(x_1, \ldots, x_n),\ \texttt{CONSTRUCT}\ \mathrm{result}(s), \qquad (*)$$

where φ is a pattern selecting vertices and s is a tree in $\mathcal{T}_\Sigma^{\mathbf{D}\cup\{\bar{x}\}}$ containing at leaves special constructs like $\mathrm{lab}(x)$ and $\mathrm{subtree}(x)$ indicating that at this position the label or the subtree rooted at the matched vertex for x should be plugged in.

More precisely, let $\bar{u}^1 \prec \ldots \prec u^\ell$ be the tuples of nodes of t selected by φ (again, $\prec$ denotes pre-order extended to tuples of nodes). Then the output of the above query on t is

$$\mathrm{result}(s[\bar{u}^1] \cdots s[\bar{u}^\ell]),$$

where $s[\bar{u}^j]$, $\bar{u}^j = u_1^j, \ldots, u_n^j$, denotes the tree obtained from s by replacing each $\mathrm{lab}(x_i)$ by $t[u_i^j]$, each $\mathrm{subtree}(x_i)$ by $t_{u_i^j}$, and each x_i occurring in an attribute definition of a by $\lambda_a^t(u_i^j)$.

As a pattern language, most of these XML query languages, combine the query primitives that are used in relational databases, hence SQL, with constructs that allow to navigate along paths, often by means of regular expressions. Clearly, they are all contained in first-order logic (FO) extended with vertical and horizontal regular path expressions as defined next.

First we say how we view attributed trees as logical structures (in the sense of mathematical logic [EF95]) over the binary relation symbols E and $<$, and the unary relation symbols $(O_\sigma)_{\sigma\in\Sigma}$. We denote this vocabulary by τ_Σ. The domain of t, viewed as a structure, equals the set of nodes of t, i.e., $\mathrm{Nodes}(t)$. E is the edge relation and equals the set of pairs $(v, v\cdot i)$ for every $v, v\cdot i \in \mathrm{Nodes}(t)$. The relation $<$ specifies the ordering of the children of a node, and equals the set of pairs $(v\cdot i, v\cdot j)$, where $i < j$ and $v\cdot j \in \mathrm{Nodes}(t)$. For each σ, O_σ is the set of nodes that are labeled with a σ.

An atomic FOREG$^+$ formula is an atomic formula over τ_Σ or is of the form

1. $a(x) = b(y)$,
2. $a(x) = d$,
3. $[r]^{\rightarrow}(x, y)$, or
4. $[r]^{\downarrow}(x, y)$,

where a and b are attributes, $d \in \mathbf{D}$, and r is a regular expression over Σ. These formulas hold for nodes u and v in t, if

1. $\lambda_a^t(u) = \lambda_t^b(v)$,
2. $\lambda_a^t(u) = d$,
3. u is a left sibling of v and the labels on the path from u to v (both included) match r, or

4. u is an ancestor of v and the labels on the path from u to v (both included) match r.

A FOREG$^+$ formula is the usual closure of FOREG$^+$ atomic formulas under conjunction, negation, and first-order quantification over nodes. Note that there is no quantification over elements from $\mathbf{D}$.

We define XML-QL as queries of the form (*) where φ is an FOREG$^+$ formula.

Theorem 4.2 *XSLT can express all XML-QL queries.*

Proof. Consider an XML-QL query of the form (*). Essentially, on a tree t, we have to (1) enumerate all tuples $\bar{u}$ in pre-order, (2) check whether $t \models \varphi(\bar{u})$, and (3) if so, output $s[\bar{u}]$.

- Enumerating all tuples in pre-order is easy. We explained in Section 4.1 how to use registers together with unique ids to simulate pebbles. So, initially, we place all pebbles on the root. Whenever we have to go to the next tuple we just move the corresponding pebble.

- Suppose the pebbles are placed on nodes $u_1, \ldots, u_n$. We show how to test whether $t \models \varphi(u_1, \ldots, u_n)$ by induction on the structure of FOREG$^+$ formulas in a straightforward manner. We can assume that φ is in prenex normal form with the quantifier-free part in disjunctive normal form.

  Atomic formulas of the form $E(u_i, u_j)$, $u_i < u_j$, $a(u_i) = b(u_j)$, $a(u_i) = d$ are tested in a straightforward manner. Slightly more difficult are atomic formulas of the form $[r]^{\downarrow}(u_i, u_j)$ and $[r]^{\rightarrow}(u_i, u_j)$. To test the former, for instance, an XSLT program does the following. First it locates u_j by doing a depth-first search of the tree. Then it walks towards the root simulating the deterministic automaton for the reverse of the language defined by r. If the program arrives at u_i in a final state it accepts, otherwise when a non-final state is reached or when u_i is not an ancestor it rejects. To test a disjunction of conjunctions of atomic formulas it suffices to find a disjunct for which all atomic formulas hold.

  Suppose we have an XSLT program to test a formula $\psi(v, \bar{u})$ which is in prenex normal form. Then $\exists x \psi(x, \bar{u})$ can be tested by placing a pebble on every node v and testing whether $\psi(v, \bar{u})$ holds. It accepts when at least one such v exists. Testing $\forall x \psi(x, \bar{u})$ is analogous.

- For the output, all $s[\bar{u}]$ have to be listed in pre-order. So, we have a rule of the form

  $$\boxed{\text{output(self), next(self)}},$$

  where "output" generates $s[\bar{u}]$ and "next" calls for the next tuple in pre-order to be checked. The only difference with XSLT is that s contains constructs of the form lab(x_i) and subtree(x_i). The value of the former can be determined by walking to x_i. The latter, can be accomplished by template rules that copy the subtree rooted at x_i.

□

To conclude, we remark that the existence of unique id's or, equivalently, the ability of XSLT to generate unique id's is necessary to simulate our abstraction of XML-QL. Indeed, Neven, Schwentick, and Vianu show that without these, XSLT cannot even express all FO patterns [NSV].

4.3 Join-free MSO

We next consider monadic second-order logic (MSO) as a pattern language. MSO is first-order logic (FO) extended with quantification over set variables. We refer the unfamiliar reader to, e.g., the books by Ebbinghaus and Flum [EF95], Immerman [Imm98], or the chapter by Thomas [Tho97a]. Join-free MSO$^+$ is then defined as MSO over the above vocabulary extended with atomic formulas of the form $a(x) = d$ that are interpreted as explained above. Note that we do not allow atomic formulas of the form $a(x) = b(y)$. In other words, we do not allow joins. Again no quantification over $\mathbf{D}$ is allowed.

Clearly, join-free MSO$^+$ can define all XPath matching patterns. The next theorem says that XSLT, in fact, is capable of expressing all unary MSO$^+$ patterns. In particular, this means that one does not need matching patterns in templates. That is, XSLT actually allows to specify rules like

```
<xsl:template match="p" mode="q">
```

where p is a matching pattern rather than just a label. It means that a rule can only be applied on nodes that satisfy p. Theorem 4.3 implies that one can test for p in the body of the template rule and, hence, does not need matching patterns.

Theorem 4.3 *Let $\varphi(x)$ be a join-free MSO$^+$ formula. There exists an XSLT program P and a mode q_φ such that $(u, q_\varphi) \rightarrow^*_{P,t}$ **true** iff $t \models \varphi[u]$.*

Proof. Essentially, a join-free MSO$^+$ formula φ using the constants $d_1, \ldots, d_n$ can be translated into an ordinary MSO formula $\tilde{\varphi}$ over an extended (but finite) vocabulary containing, say, the sets $\{O^a_{d_1}, \ldots, O^a_{d_n} \mid a \in A\}$. Here, for a node u and a tree t, $t \models O^a_{d_i}(u)$ iff $\lambda^t_a(u) = d_i$. Hence, it suffices to proof the theorem for MSO rather than MSO$^+$.

There is a well-known connection between sets of trees defined by MSO formulas and (unranked) regular tree languages. Before we state this, we recall the definition of tree automata over unranked trees.

A *deterministic bottom-up tree automaton (DBTA)* is a tuple $B = (Q, \Sigma, F, \delta)$, where Q is a finite set of states, $F \subseteq Q$ is the set of final states, and δ is a function $Q \times \Sigma \to 2^{Q^*}$ such that $\delta(q, a)$ is a regular language for every $a \in \Sigma$ and $q \in Q$. To ensure determinism, we require that $\delta(q, \sigma) \cap \delta(q', \sigma) = \emptyset$ for all $q \neq q' \in Q$ and $\sigma \in \Sigma$. The semantics of B on a tree t, denoted by $\delta^*(t)$, is defined inductively as follows:

- if t consists of only one vertex labeled with σ then $\delta^*(t) = q$ with q such that $\varepsilon \in \delta(q, \sigma)$;
- if t is of the form $\sigma(t_1, \ldots, t_n)$, then $\delta^*(t) = \{q \mid \delta^*(t_1) = q_1, \ldots, \delta^*(t_n) = q_n$ and $q_1 \cdots q_n \in \delta(q, \sigma)\}$.

A tree t over Σ is *accepted* by the automaton B if $\delta^*(t) \in F$. A set of trees is *regular* if it is accepted by a DBTA.

Let $\varphi(x)$ be an MSO formula. It is known that the set $\{(t,u) \mid t \models \varphi[u]\}$ is a regular tree language [NS99, Nev99a, Tho97a]. Here, (t,u) is the tree where a distinguishing * is attached to the label of u (for instance, if $t[u] = \sigma$, then the label of u in (t,u) is $\sigma *$).

Let $B = (Q, \Sigma, F, \delta)$ be the DBTA accepting the tree language $\{(t,u) \mid t \models \varphi[u]\}$. Let for each q, σ, $M^{q,\sigma} = (Q^{q,\sigma}, Q, \delta^{q,\sigma}, q_0^{q,\sigma}, F^{q,\sigma})$ be the DFA accepting $\delta(q,\sigma)$. Note that $M^{q,\sigma}$ accepts strings over Q. To simulate B we start the following procedure at the root of (t,u). We use as states (σ, g) where $\sigma \in \Sigma$ and g is a function from Q to $\bigcup_{q\in Q} Q^{q,\sigma}$. The idea is as follows. If a node $ui \neq \varepsilon$ is called in state (σ, g) then $t[u] = \sigma$ and for each $q \in Q$, $\delta^{q,\sigma *}(\delta^*(t_{u1}) \cdots \delta^*(t_{u(i-1)})) = g(q)$. That is, the parent of ui is labeled with σ and when $M^{q,\sigma}$ processes the states assumed by B at the left siblings of ui it arrives at ui in state $g(q)$. Define g_0^σ as $g_0(q) = q_0^{q,\sigma}$ for all $q \in Q$.

The XSLT program P starts at the root where it defines a variable

$$y := \boxed{(t[\varepsilon], g_0)(\text{first})}.$$

That is, it selects the first child of the root in state $(t[\varepsilon], g_0)$ (we tacitly assume that t consists of more than one node).

Suppose P selects ui in state (σ, g).

- First, P computes $\delta^*(t_{ui})$, the state B assumes at ui. More precisely, if ui is a leaf $\delta^*(t_{ui}) := \tilde{q}$ for which $\varepsilon \in \delta(\tilde{q}, t[ui])$. Otherwise, P defines a variable $y := \boxed{(t[ui], g_0)(\text{first})}$ that selects the first child of ui. Denote the value of this variable by $\tilde{q}$.
- If ui is the right most child of u then P returns the unique q for which $g(q) \in F^{q,\sigma}$. This means that, $M^{q,\sigma}$ accepts $\delta^{q,\sigma *}(\delta^*(t_{u1}) \cdots \delta^*(t_{ui})))$. Otherwise, P selects its right sibling in state (σ, g') where $g'(q) = \delta^{q,\sigma}(g(q), \tilde{q})$, for each $q \in Q$.

□

4.4 Termination

An XSLT program P is said to be *terminating* when τ_P is everywhere defined.

Theorem 4.4 *It is undecidable whether XSLT programs are terminating.*

Proof. It is well known that the halting problem of multi-head automata on strings is undecidable [FS73]. In brief, a multi-head automaton is a two-way deterministic finite automaton over strings with multiple heads. Movements depend on the current state and the symbols below the respective heads. Clearly, a multi-head automaton can be simulated by a one-head automaton with pebbles. Each pebble corresponds to a head. One step of a multi-head automaton can be simulated by several steps of a one-head automaton with registers. Indeed, when the automaton needs to know the symbol under a specific head it just makes a sweep of the string looking for the correct position.

□

4.5 Composition

Theorem 4.5 *XSLT programs are not closed under composition.*

Proof. We first show that the height of an output tree generated by a terminating XSLT program is at most polynomial in the height of the input tree. Indeed, let $P = (\Sigma, \Delta, M, M, \mathrm{start}, R)$ be an XSLT program and let t be a Σ-tree. Further, let D be the number of different attribute values occurring in t and issued as constants in P. Let h be the maximum height of a forest in a construction template rule. Let n be the maximum number of parameters occurring in the head of a template rule. As the number of local configurations is $N := |\mathrm{Nodes}(t)| \times |Q| \times D^n$, this means that the height of the output tree is bounded by $h \times N$, which is polynomial in the size of t.

Consider the programs P_1 and P_2 of Example 3.15. The composition P_2 after P_1 transforms an input tree of height n to one of height 2^n. By the previous argument, no single XSLT program can express this transformation. $\square$

Acknowledgements

We thank Bertram Ludäscher, Dan Suciu, and Victor Vianu for encouraging and helpful discussions.

References

[ABS99] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[AQM+97] 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.

[BDHD96] P. Buneman, S. Davidson, G. G. Hillebrand, and D.Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, volume 25:2 of *SIGMOD Record*, pages 505–516. ACM Press, 1996.

[Bex] G. J. Bex. Examples of translations from XML-QL to XSLT. http://www.luc.ac.be/~gjb/xml-ql2xslt.html.

[CD] J. Clark and S. Deach. Extensible stylesheet language (XSL). http://www.w3.org/TR/WD-xsl.

[Cla] J. Clark. XML Path Language (XPath). http://www.w3.org/TR/xpath.

[Cla99] James Clark. XSL transformations version 1.0. http://www.w3.org/TR/WD-xslt, august 1999.

[Con] World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3.org/XML/.

[Cou94] B. Courcelle. Recognizable sets of graphs: equivalent definitions and closure properties. *Math. Struct. in Comp. Science*, 4:1–32, 1994.

[DFF+99a] A. Deutsch, M. fernandez, D. Florescu, A. Levy, D. Maier, and D. Suciu. Querying XML data. *Data Engineering Bulletin*, 22(3):10–18, 1999.

[EF95] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer, 1995.

[FFK+98] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Catching the boat with strudel: Experiences with a web-site management system. In L. M. Haas and A. Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data*, pages 414–425. ACM Press, 1998.

[FSW99] M. Fernandez, J. Siméon, and P. Wadler, editors. *XML Query languages: Experiences and Exemplars*, 1999. `http://www-db.research.bell-labs.com/user/simeon/xquery.html`.

[FS73] P. Flajolet and J.-M. Steyaert. Decision Problems for Multihead Finite Automata. In *Proceedings of the second International conference on Mathematical Foundations of Computer Science*, pages 225–230, 1973.

[Imm98] N. Immerman. *Descriptive Complexity.* Springer, 1998.

[MN99] S. Maneth and F. Neven. A formalization of tree transformations in XSL. To appear in the proceedings of the Seventh International Workshop on Database Programming Languages, *Lecture Notes in Computer Science*, 1999.

[MSV00] T. Milo, D. Suciu, and V. Vianu. Type checking for XML transformers. In *Proceedings of the Nineteenth ACM Symposium on Principles of Database Systems*, pages 11–22. ACM Press, 2000.

[Nev99a] F. Neven. *Design and Analysis of Query Languages for Structured Documents — A Formal and Logical Approach.* Doctor's thesis, Limburgs Universitair Centrum (LUC), 1999.

[Nev99b] F. Neven. Extensions of attribute grammars for structured document queries. To appear in the proceedings of the Seventh International Workshop on Database Programming Languages, *Lecture Notes in Computer Science*, 1999.

[NS99] F. Neven and T. Schwentick. Query automata. In *Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems*, pages 205–214. ACM Press, 1999.

[NS00] F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proc. 19th Symposium on Principles of Database Systems (PODS 2000), Dallas*, pages 145–156, 2000.

[NSV] F. Neven, T. Schwentick, and V. Vianu. Automata over infinite alphabets. In preparation.

[NVdB98] F. Neven and J. Van den Bussche. Expressiveness of structured document query languages based on attribute grammars. In *Proceedings of the Seventeenth ACM Symposium on Principles of Database Systems*, pages 11–17. ACM Press, 1998.

[PQ68] C. Pair and A. Quere. D'efinition et etude des bilanguages r'eguliers. *Inform. and Control*, 13:565–593, 1968.

[Rob99] J. Robie. The design of XQL. `http://www.texcel.no/whitepapers/xql-design.html`, 1999.

[Tho97a] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 7. Springer, 1997.

[Wad99] P. Wadler. A formal semantics of patterns in XSLT. Markup Technologies, 1999.