Lazy XSL Transformations

Steffen Schott
Universität Karlsruhe
Program Structures Group
Adenauerring 20a
76133 Karlsruhe, Germany
info@steffen-schott.de

Markus L. Noga
Universität Karlsruhe
Program Structures Group
Adenauerring 20a
76133 Karlsruhe, Germany
markus@noga.de

ABSTRACT

We introduce a lazy XSLT interpreter that provides random access to the transformation result. This allows efficient pipelining of transformation sequences. Nodes of the result tree are computed only upon initial access. As these computations have limited fan-in, sparse output coverage propagates backwards through the pipeline.

In comparative measurements with traditional eager implementations, our approach is on par for complete coverage and excels as coverage becomes sparser. In contrast to eager evaluation, lazy evaluation also admits infinite intermediate results, thus extending the design space for transformation sequences.

To demonstrate that lazy evaluation preserves the semantics of XSLT, we reduce XSLT to the lambda calculus via a functional language. While this is possible for all languages, most imperative languages cannot profit from the confluence of lambda as only one reduction applies at a time.

Categories and Subject Descriptors

D.3.4 [**SOFTWARE ENGINEERING**]: Processors—*Interpreters*; D.1.1 [**SOFTWARE ENGINEERING**]: Applicative (Functional) Programming; I.1.3 [**SYMBOLIC AND ALGEBRAIC MANIPULATION**]: Languages and Systems—*Evaluation strategies*

General Terms

Languages, Theory, Measurement, Performance

1. INTRODUCTION

Document transformation is a staple of information technology. As early as 1973, special-purpose transformation languages like `sed` and `awk` processed line-oriented text documents with regular expressions. In the 1990s, such texts were increasingly replaced by semi-structured, or hypertext documents. As traditional HTML is hard to parse and navigate, corresponding transformation languages like `php` and `JSP` are usually embedded in a source document to generate variants of it. The advent of the fixed and unambiguous XML syntax in 1998 made stand-alone transformation languages feasible again. Today, *Extensible Stylesheet Language Transformations*, or XSLT, is the predominant XML transformation language.

Every transformation is both a consumer and a producer of documents. The line-oriented transformation languages both consume input and produce output sequentially. A sequence of transformations can be pipelined using fixed-size buffers. Individual pipeline stages may execute in parallel. Lags are small: the assembly can start producing output before its entire input has been consumed. Fig. 1 illustrates this using a list of species annotated by size and as being real or mythical.

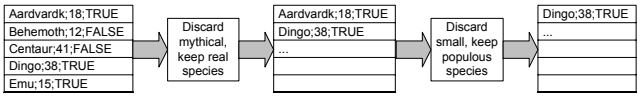

Figure 1: Pipelining line-oriented transformations

Because the XSLT data model contains a plethora of navigation steps on XML trees, XSLT transformations effectively consume the input document using random access. In contrast, the output document is produced sequentially in depth-first order, also called document order.

Traditional implementations follow the XSLT data model literally. Because they cannot produce output out of order, every intermediate result in a sequence of transformations must be produced completely into an unbounded buffer before the subsequent transformation may begin random-access consumption. These barriers eliminate coarse-grained parallelism. As all intermediate steps must complete before the first output is produced, large lags are introduced (see fig. 2).

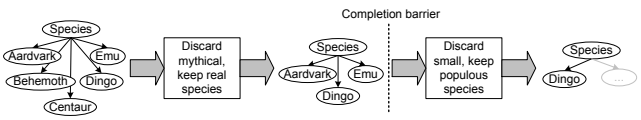

Figure 2: Eager XSLT prevents pipelining

To admit pipelining, a novel XSLT implementation must be capable of producing partial outputs in reply to random-access requests from consumers. This mode of operation

DocEng'03, November 20–22, 2003, Grenoble, France.

accrues an additional benefit: partial coverage of a transformation output can propagate backwards to its input, and thus along an entire transformation sequence. Because they cannot skip input, traditional line-oriented tools are incapable of this (see fig. 3).

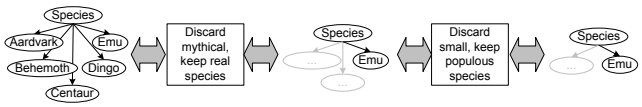

Figure 3: Random-access XSLT allows pipelining

How to provide such random access? Essentially, document trees are nested lists, a well-known concept in functional programming. Functional languages achieve the goal of omitting unnecessary intermediate results by lazy evaluation.

In contrast to eager evaluation, lazy evaluation computes the value of an expression not as it occurs, but when it is used. A necessary and sufficient condition for lazy evaluation is confluence of the programming language — i.e., it must be possible to reduce expressions in arbitrary order without changing their result.

According to the Church-Turing thesis, all languages can ultimately be reduced to the confluent lambda calculus. For lazy evaluation to be profitable, however, multiple alternative reductions must actually exist. Consider the following example in Java:

```
class A {
 public A l,r;
 public A(A l,A r) { this.l=l; this.r=r; }
 public A()        { System.out.println(num++); }
 static int num=0;
}
```

Evaluating the expression `new A(new A(),new A()).r` yields an instance of `A` whose fields are both `null`. However, because the leaf constructor has side effects, it also produces the output `"0\n1\n"` and increments `num` to a value of 2.

Java, and thus the example, can be reduced to the lambda calculus. To preserve semantics in such a model, all method invocations must pass and return a global state parameter encapsulating memory updates and I/O. This imposes a strict sequential order on method invocations. While lazy evaluation is possible, it must produce the same method invocation sequence as eager evaluation. Alternative reduction paths do not exist.

Where reduction alternatives abound, benefits from lazy evaluation are threefold: Due to its adaptive nature, lazy evaluation routinely surpasses static optimizations in scenarios with stochastic access profiles. Lazy evaluation of a composition of programs jointly optimizes them for the given usage scenario by propagating the effects of low coverage. And finally, lazy evaluation extends the design space for programs by allowing the use of infinite intermediate structures.

We reap these benefits for XSLT. After reviewing the state of the art in XSLT processing and optimization in section 2, we give denotational semantics for XSLT in terms of a functional language and examine the constraints on reductions in section 3. For readability, we refrain from breaking things down to lambda. In section 4, we introduce our architecture for a lazy XSLT interpreter that allows random-access on partial results and propagates sparse coverage. Section 5 presents and discusses performance comparisons between lazy and traditional XSLT processors. The final section, 6, summarizes our results and outlines directions for future work.

2. STATE OF THE ART

In this section, we briefly cover the relevant XML standards, then we focus on optimizations for XSLT transformations, which can be subclassed into static and dynamic ones.

2.1 XML Processing

The *World Wide Web Consortium* (W3C) is a forum developing specifications, guidelines and tools to facilitate interoperability on the web. One of the W3C's main contributions is the *Extensible Markup Language* (XML) [27, 28], a simple yet flexible markup language derived from SGML. Since its standardization as a *W3C Recommendation* in 1998, XML has become the predominant data exchange and document format on the web.

Well-formed XML documents are depth-first serializations of unranked, ordered, labelled trees. Inner tree nodes are called *elements*. They can be decorated with *attributes* and *namespace* information. Leaf nodes come in greater variety. Besides elements, *text* nodes, *comments* or *processing instructions* are also admissible. All of these are essentially atomic. Although initially specified on character sequences, the *XML Information Set* (Infoset) [7] recommendation formally models the information content of an XML document as a tree consisting of so-called *information items*.

Subsets of the well-formed XML documents can be defined using document type languages, like *XML Document Type Declaration* (DTD) [27] or *XML Schema* [30]. A document belonging to the subset defined by a specification is called *valid* according to that specification. In XML parlance, the process of type checking against a specification is called *validation*.

Actual programs can access well-formed or valid XML documents via one of several APIs, each of which slightly enriches the underlying information set model. The *Document Object Model* (DOM) [26] is a *tree-based* interface, providing a standard set of node objects for representing the document tree as well as a model of how these objects can be combined and methods for accessing and manipulating them.

However, there is no requirement that the XML Information Set be made available through a tree structure. Whereas *text-based* interfaces abandon the document's hierarchical structure and operate on its plain XML source only, *event-based* interfaces like *Simple API for XML Processing* (SAX) [17] decompose the structure into a stream of parse events.

Query-based interfaces form the highest level of abstraction. They interpret query strings to select arbitrary fragments of the document tree. *XML Path Language* (XPath) [29] is a regular path language that has become the acknowledged query language for addressing and selecting parts of an XML document. Besides its primary purpose, it also provides basic facilities for manipulation of strings, numbers and booleans. The underlying *XPath data model* represents an XML document as a tree of nodes and allows random access via navigation steps.

XPath was primarily designed to provide a data model and a query sublanguage for node sets and primitive values to be used by other W3C languages such as *XQuery* [3] or *XSL Transformations* (XSLT) [31]. The latter is a declarative programming language for the transformation of XML documents. An XSLT program is also called a *stylesheet*. It consists of rules mapping a *source tree* to a *result tree*. As rule bodies can contain mathematical variables, recursive rule invocations and case differentiations, XSLT is a Turing-complete language.

2.2 Static Optimization of XSLT

Currently, more than a dozen implementations of XSLT are available. Several of them aim at optimizing XSLT programs to decrease their execution time. All of them rely on static preprocessing.

2.2.1 Lowering

A common approach to language implementation is compilation to a lower-level intermediate code for a *virtual machine*. The benefits from this approach over direct interpretation of high-level language instructions are well known. The VM need not perform semantic analysis, which is precomputed at compile-time. Because only relatively few low-level instructions must be supported, instead of a language defined to ease the work of the programmer, the VM decreases in complexity and possibly exhibits better caching behavior.

XSLT Virtual Machine (XSLTVM) [20] is the software implementation of a stack-based "CPU" for compiled XSLT code. Both instruction set and data types are tailored to processing XSLT instructions and XPath expressions. Unfortunately, the authors confine themselves to presenting the architecture. They fail to mention whether it has been implemented, and which practical results could be obtained from this approach.

However, compilation to *Java Virtual Machine* (JVM) bytecode [9, 2] tends to achieve promising performance [14].

2.2.2 Static evaluation

In [16], one of us introduced *static evaluation* of XSLT programs over known input types. Similar to *abstract interpretation*, this approach evaluates transformations over a set of input trees rather than one specific tree.

Static evaluation can simplify cascades of XSLT steps by taking constraints due to later steps into account early on. It can resolve apply-template calls into call-template calls, effectively removing polymorphism. And finally, it can detect dead rules as well as dead portions of the input.

Work to validate these theoretical results in practice is approaching completion.

2.2.3 Static type checking

Static type checking also exploits known document types. Given a transformation and a type specification for its inputs, this optimization statically verifies the possible output types. When compliance with the input type of the next stage can be statically verified, the respective run-time check can be omitted.

This approach aims to eliminate validation steps rather than to speed up interpretation itself. Although the general problem is undecidable, several authors have succeeded at defining subsets of XSLT for which static type checking is feasible [18, 1, 25].

2.2.4 Query optimization

Recent research in the database community concentrates on *XPath query optimization*. They have been motivated by the observation that querying the source tree tends to be the most time-critical part in XML processing standards such as XSLT or XQuery.

By experimental analysis, [11, 12] reveal query evaluation in several popular XPath processors to require a worst-case execution time that is exponential in the size of the query. The authors propose a main-memory evaluation algorithm for XPath 1.0 with guaranteed polynomial-time combined complexity, i.e., its execution time is polynomial in both the input data size and the query size. Moreover, they identify some practically relevant fragments of XPath for which query evaluation can be optimized even further, obtaining linear-time processing algorithms in the best case.

XPath takes a navigational approach to node selection. Its navigational axes basically allow random access to XML trees. This requires an in-memory representation of the entire XML document, which poses problems when document sizes exceed main memory capacity.

This issue can be evaded by embracing a stream-based strategy for XML processing, which avoids in-memory storage of the data as much as possible. Of particular concern for SAX-like processing are the *reverse axes* of XPath, e.g., `parent::` and `preceding::`, which select nodes occurring before the context node in document order. [21, 22] developed a rewriting algorithm which transforms XPath expressions with reverse axes into equivalent ones containing forward axes only.

Another approach is the optimization of XPath queries using document type information [15]. The paper makes several restrictions to enable classical relational database query optimization techniques. These restrictions and the omission of an XSLT context for the XPath expressions make our symbolic execution approach the more general one.

2.3 Dynamic Optimization and Laziness

All optimization methods discussed so far share one common property: they are static, i.e., they are performed prior to runtime. This means that transformations and type specifications must always be known a priori. With dynamic approaches, all optimizations must first and foremost repay the time consumed by performing them.

In addition, static optimizations fail to take dynamic access profiles into account, which have been employed to great success in other application domains.

2.3.1 Functional Programming

Functional programming languages are generally classified according to their evaluation strategy. *Strict* languages perform *call by value* function application, i.e., argument expressions are evaluated in advance and the resulting values are passed to the function. In contrast to this, *lazy* languages evaluate argument expressions on demand, i.e., they pass references to yet unevaluated argument expressions (*call by name*) and evaluation is delayed until the value of the expression is required. This strategy avoids the evaluation of argument expressions which are irrelevant to the function result.

In theory, all functional languages can be mapped to expressions in the lambda calculus. Program execution then corresponds to a reduction of the lambda expression to normal form by graph reduction. This process is confluent, so reductions can be applied in arbitrary order.

The main difference between lazy and strict functional languages is the selection strategy for the next subexpression to reduce (*reducible expression*,*redex*). *Normal order reduction* selects the leftmost outermost redex first. This results in lazy evaluation and guarantees termination, if a normal form exists. Strict evaluation is performed by *applicative order reduction*, selecting the rightmost redex for reduction. In this case, termination cannot be guaranteed.

Implementations of lazy evaluation can be subclassed into direct and caching approaches. The latter aim to increase performance further by tracking shared expressions and results.

2.3.2 Just-in-time Compilation

High-performance virtual machine implementations usually rely on *just-in-time compilation* (JIT), where source or intermediate code is translated into native machine code on the fly, removing the interpretation overhead. Like all dynamic approaches, JIT compilation increases the total execution time of the program. Because compilation time is added to execution time, the savings in execution time must outweigh the compilation overhead. Therefore, all JITs emphasize compilation speed at the expense of complex optimizations.

Consequently, it is more reasonable to compile methods as they are executed, rather than compiling the complete program in advance. More sophisticated JITs gather profiling information during the program run and base the decision whether to interpret or compile a method upon these data. Hence, just-in-time compilers embrace the concept of lazy processing by dynamically adapting to execution profiles in order to perform as little work as possible.

2.3.3 Lazy XML Parsing

In our previous work [19, 23], we applied lazy processing to XML parsing. Our lazy parser extracts the basic document structure during a preprocessing phase. Actual parsing and document tree construction are deferred until the nodes are queried by the consumer. To him, our parser presents a normal DOM interface.

Our implementation outperforms eager DOM parsers for partial traversals with 0% to 80% document coverage. For full coverage, the overhead imposed upon by the lazy processing strategy remains acceptable.

All in all, the lazy processing strategy has been successfully applied to a variety of domains where dynamic access or execution profiles can be profited by. Considering this, it seems promising to adopt this strategy for partial XSLT transformations as well.

3. MODEL

We introduce a model for XSLT using denotational semantics, then we establish a map from standard XSLT to the model. Then we examine its potential for lazy evaluation.

Several authors give denotational semantics for XPath expressions [32, 10]. As path expressions are peripheral to this paper, any of these semantics is sufficient. We refer to the semantics of set and primitive path expressions e in processor state S_P by $\mathcal{P}[\![\, e \,]\!]\, S_P$. Set expressions return node sets, whereas primitive expressions return numbers, strings or booleans.

3.1 Formal Processing Model

The model consists of three parts: structures, state and statements. We address each of them in turn.

Let $\mathcal{U}$ be the Unicode alphabet [6] and $\mathcal{I} \subset \mathcal{U}^*$ be a set of identifiers. Let $\mathcal{V}$ be the set of admissible variable values in the XPath model used, with at least $\mathcal{U}^* \subset \mathcal{V}$.

Processor *structures* consist of documents, transformations, modi and template rules. A *document* D is a node, $D = (n, A, C)$, where $n \in \mathcal{I}$ is an identifier, the name, $A : \mathcal{I} \to \mathcal{U}^*$ is a partial finite map of attributes and C is a finite sequence of nodes, the children.

A *transformation* $X = \{M_0, \ldots, M_n\}$ is a set of modi accessible by their index. A *mode* $M = \{R_0, \ldots, R_n\}$ is an ordered set of template rules R_i. A *template rule* $R = (n, e, b)$ consists of a name n, a primitive path expression e, the *guard*, and a statement b, the rule body.

The processor *execution state* $S_P = (S_G, S_L)$ consists of invariant global state S_G and variant local state S_L.

Invariant global state forms a tuple $S_G = (X, D, V_G)$, where X is a transformation, D is the source document and $V_G : \mathcal{B} \to \mathcal{V}$ is partial finite map of global variable assignments.

Variant local state consists of a tuple $S_L = (m, V_L)$, where m is the current node (a reference into the source document D), and $V_L : \mathcal{B} \to \mathcal{V}$ is a partial finite map of local variable assignments, with $\operatorname{def} V_G \cap \operatorname{def} V_L = \emptyset$.

We now introduce statements x and their denotational semantics $\mathcal{S}[\![\, x \,]\!]\, (S_G, S_L)$. As S_G is constant, we write $\mathcal{S}[\![\, x \,]\!]\, S_L$ and assume S_G to be implicitly available.

Statements either have side effects, or they do not. Only variable definitions have side effects. They come in two flavors, set and primitive variables. A set variable definition $\$n = b$ consists of an identifier n, the variable name, and a set path expression, the initializer. Primitive variable definitions $\#n = b$ are similar, except that the initializer b is a statement.

$$\mathcal{S}[\![\, \$n = b \,]\!]\, S_L = \begin{cases} n \notin \operatorname{def} V_L \cup \operatorname{def} V_G & \mathcal{S}[\![\, \$n := b \,]\!]\, S_L \\ n \in \operatorname{def} V_L \cup \operatorname{def} V_G & \bot \end{cases}$$

$$\mathcal{S}[\![\, \$n := b \,]\!]\, S_L = V_L \cup \{(n, \mathcal{P}[\![\, b \,]\!]\, S_P)\}$$

$$\mathcal{S}[\![\, \#n = b \,]\!]\, S_L = \begin{cases} n \notin \operatorname{def} V_L \cup \operatorname{def} V_G & \mathcal{S}[\![\, \#n := b \,]\!]\, S_L \\ n \in \operatorname{def} V_L \cup \operatorname{def} V_G & \bot \end{cases}$$

$$\mathcal{S}[\![\, \#n := b \,]\!]\, S_L = V_L \cup \{(n, \mathcal{S}[\![\, b \,]\!]\, S_L)\}$$

Every definition adds exactly one variable to the set. It is an error to overwrite already assigned variables (*single static assignment* (SSA) within a given scope). Statements free of side effects are either groupings, local flow control, global flow control or output constructors. Groupings are either blocks or renamings.

A block $\{x_1; x_2; \ldots; x_n;\}$ establishes sequential order on its constituent statements x_i.

$$\mathcal{S}[\![\,\{\}\,]\!]\,S_L = \epsilon$$

$$\mathcal{S}[\![\,\{x_1; \ldots; x_n;\}\,]\!]\,S_L = \begin{cases} x_1 \in = & \mathcal{S}[\![\,\{x_2; \ldots; x_n;\}\,]\!]\,S'_L \\ & \text{where } S'_L = (m, V'_L) \\ & \quad V'_L = \mathcal{S}[\![\,x_1\,]\!]\,S_L \\ x_1 \notin = & \mathcal{S}[\![\,x_1\,]\!]\,S_L \\ & \mathcal{S}[\![\,\{x_2; \ldots; x_n;\}\,]\!]\,S_L \end{cases}$$

A variable renaming $\texttt{map}\,A\,b$; consists of a partial function $A : (\mathcal{I} \setminus \operatorname{def} V_G) \to \operatorname{def} V_L$, and a statement b.

$$\begin{aligned}\mathcal{S}[\![\,\texttt{map}\,A\,b\,]\!]\,S_L =& \mathcal{S}[\![\,b\,]\!]\,(m, V'_L) \\ & \text{where } V'_L = \{n \mapsto V_L(A(n)) \mid n \in \operatorname{def} A\}\end{aligned}$$

It changes the names of local variables for the purpose of evaluating b. Because this has no effect on preceding or following statements, it preserves the SSA property in the statically surrounding block.

Local flow control comprises case differentiations or loops.

An if statement $\texttt{if}(e)\,b_1\,\texttt{else}\,b_2$ consists of a primitive path expression e, the condition and two statements $b_{1,2}$, the mutually exclusive bodies. If the condition applies, the first body is executed, otherwise the second.

$$\mathcal{S}[\![\,\texttt{if}(e)\,b_1\,\texttt{else}\,b_2\,]\!]\,S_L = \begin{cases} e' & \mathcal{S}[\![\,b_1\,]\!]\,S_L \\ \neg e' & \mathcal{S}[\![\,b_2\,]\!]\,S_L \end{cases} \quad \text{where } e' = \mathcal{P}[\![\,e\,]\!]\,S_P$$

A for-each statement $\texttt{foreach}(e)\,b$ consists of a set path expression e, the selection, and a statement b, the body. The body is applied to every node in the set M' obtained by evaluating the expression e. $\leq$ denotes document order.

$$\begin{aligned}\mathcal{S}[\![\,\texttt{foreach}(e)\,b\,]\!]\,S_L =& \mathcal{S}[\![\,\texttt{for}\,b\,M'\,]\!]\,S_L \\ & \text{where } M' = \mathcal{P}[\![\,e\,]\!]\,S_P \\ \mathcal{S}[\![\,\texttt{for}\,b\,\emptyset\,]\!]\,S_L =& \epsilon \\ \mathcal{S}[\![\,\texttt{for}\,b\,M'\,]\!]\,S_L =& \mathcal{S}[\![\,b\,]\!]\,(m', V_L) \\ & \mathcal{S}[\![\,\texttt{for}\,b\;M'\backslash\{m'\}\,]\!]\,S_L \\ & \text{where } m' = \min_{\leq} M'\end{aligned}$$

Global flow control consists of monomorphic and polymorphic rule invocations, which differ only in the selection of the invoked rule.

A monomorphic rule invocation $n :: i()$ consists of an identifier n, the template name, and a mode index i. The first template named n in mode M_i is executed.

$$\begin{aligned}\mathcal{S}[\![\,n :: i()\,]\!]\,S_L =& \mathcal{S}[\![\,X.M_i.R_j.b\,]\!]\,S_L \\ & \text{where } j = \min\{j \mid X.M_i.R_j.n = n\}\end{aligned}$$

Polymorphic rule invocations $i()$ consist only of a mode index i. They execute the first template in mode M_i whose guard admits the current node.

$$\begin{aligned}\mathcal{S}[\![\,i()\,]\!]\,S_L =& \mathcal{S}[\![\,X.M_i.R_j.b\,]\!]\,S_L \\ & \text{where } j = \min\{j \mid \mathcal{P}[\![\,X.M_i.R_j.e\,]\!]\,S_L \neq \emptyset\}\end{aligned}$$

There are two kinds of output constructors, node and attribute constructors.

Node constructors $\texttt{new}\,n(b_a, b_c)$ consist of an identifier n, the node name, and two statements b_a, b_c, the initializers for node attributes and node children. They create exactly one node, whose attributes and children are determined by the respective constructors dynamically contained in b_a, b_c.

$$\begin{aligned}\mathcal{S}[\![\,\texttt{new}\,n(b_a, b_c)\,]\!]\,S_L =& (n, A, C) \\ & \text{where } A = \bigcup \mathcal{S}[\![\,b_a\,]\!]\,S_L \\ & \quad C = \quad \mathcal{S}[\![\,b_c\,]\!]\,S_L\end{aligned}$$

Attribute constructors $\texttt{new}\,@n(b)$ consist of an identifier n, the attribute name, and a statement b, the initializer. They create an attribute map that projects the given name on the evaluated initializer.

$$\mathcal{S}[\![\,\texttt{new}\;@n(b)\,]\!]\,S_L = \{n \mapsto \mathcal{S}[\![\,b\,]\!]\,S_L\}$$

Primitive path expressions e are statements.

$$\mathcal{S}[\![\,e\,]\!]\,S_L = \mathcal{P}[\![\,e\,]\!]\,S_P$$

3.2 Mapping Standard to Model

Our document model consists of named, attributed nodes with possibly empty child sequences. This corresponds exactly to XML elements. XML additionally defines unnamed, unattributed, childless text and comment nodes, as well as named, unattributed, childless processing instructions. Each of these variants can be represented in our model by using reserved identifers as names and encoding the payload as attributes. Thus, our document model captures XML.

Stylesheets, modi, rules and templates in our model correspond directly to their homonyms in XSLT. The only difference is perceptual: XSLT treats modi as rule attributes, whereas we consider them rule containers. Thus, all model structures capture the corresponding standard.

State additionally encompasses variable assignments V_L and V_G and the context node m. The variable assignments and the context node correspond directly to their homonyms in XSLT. XSLT additionally knows context positions and sizes, which can be modelled with variables.

Note that XSLT operates on a context node list instead of a context node set. This list is ordered either in document order or in reverse document order, depending on the direction of the preceding XPath step. Only two operations exploit this order: determining the `position`() of a node in an XPath expression, and iterating over results in a `<for-each>` statement. We express this by composing simpler model operations to form static replacements.

Given a `position`() implementation that implicitly assumes document order and the standard `count`() operation, we can implement the original `position`() behavior. Replace all `position`() calls preceded by an inverse document order step with (`count`() − `position`()).

Now we model `<for-each>` statements whose final XPath step has inverse document order. Replace the model b of the original `<for-each>` body with $n :: i(\$cur = ".")$, where i, n are a previously unused mode index and identifier, respectively. Then introduce a new rule $R = (n, e, b)$ in M_i:

$$\begin{aligned}(n, *, \{&\texttt{foreach}\,(\$cur/[\texttt{position}() = \texttt{last}()])\,b; \\ &\$next = \$cur/[\texttt{position}()! = \texttt{last}()]; \\ &\texttt{map}\,\{cur \mapsto next\}\,n :: i(); \\ \})&\end{aligned}$$

In sum, our model structures capture the main XSLT structures[1]. Now, we consider XSLT statements.

Variable definitions in XSLT carry both a path expression e and a statement body b. If e is present, it takes precedence over b. We can construct this behavior from atomic operations: $\$n = \texttt{if}(e)\, e\, \texttt{else}\, b;$.

XSLT knows another form of variable definition, the parameter definition. It may supply default values v for parameters not passed to a rule. For all such parameters n, we introduce new variables $\$n' = \texttt{if}(\$n)\, \$n\, \texttt{else}\, \$v$; and replace all occurrences of $\$n$ inside the rule body with $\$n'$.

The `<call-template>` statement and its `<with-param>` children in XSLT differ from our monomorphic rule invocations in allowing parameter assignments. These can be modelled with surrounding variable `map`s.

The other rule invocation variant in XSLT is dynamic. `<apply-templates select="s" mode="i"/>`, constructs the node set matching `s`, then invokes matching rules for each member of that set. This translates to `foreach` $s\ i(A);$.

`<if>` statements in XSLT are devoid of an else branch. We simply fill the else branch in our model with an empty block {}. XSLT additionally allows cascaded case differentiations via `<choose>` and `<when>` statements. We represent them as `if` cascades in the model.

The XSLT `<value-of>` statement is absent from our model, because primitive expressions are statements.

For any known input type, the `<copy-of>` and `<copy>` statements can be expanded to an `if` cascade, constructor invocations and recursion. We cannot represent these statements for the rare case of unknown input schemata.

Our model thus captures the main XSLT statements[2]. Summing up, most of XSLT has direct correspondences in the model or can be constructed from simpler parts.

3.3 Model Evaluation Strategies

In a confluent language, reducible expressions, or *redexes*, can be reduced in any order without changing the normal form. Normal forms consist only of irreducible expressions.

Eager evaluation always reduces the leftmost innermost redex. Lazy evaluation always reduces the leftmost outermost redex. As lazy evaluation is guaranteed to terminate if any sequence of reductions does, it is also called *normal order* evaluation. This does not hold for eager evaluation.

Our model expresses all statements as pure functions free of side effects, which have direct correspondences in the lambda calculus. Lambda is confluent due to the Church-Rosser-Property [4], therefore so is our model.

Everything that occurs on a left hand side $\mathcal{S}[\![\, x \,]\!]\, S_L$ of a semantics definition, that is, every pair of statement and state, is a redex. Only nodes occur solely on right hand sides. Therefore, nodes and node trees are in normal form.

The XSLT standard proscribes eager evaluation: it proceeds in document order by applying leftmost reductions. Due to confluence, our interpreter can employ lazy evaluation without changing the result.

[1]XSLT admits several other top-level declarations, e.g., `<xsl:key>`, `<xsl:output>`, `<xsl:preserve-space>` and `<xsl:skip-space>`. Because the global nature of keys inhibits lazy evaluation, we exclude them from consideration. The other declarations are peripheral to our approach.

[2]Some additional XSLT statements, like the `<xsl:message>` debugging facility, make the evaluation order externally visible. As they inhibit lazy evaluation, we exclude them from consideration.

Do multiple alternative reductions exist? Yes, and they offer high potential for optimization. E.g., statements in a block without variables can be reduced in any order:

$$\begin{aligned}\mathcal{S}[\![\, \{x_1; \ldots; x_n;\} \,]\!] &= \mathcal{S}[\![\, x_1 \,]\!]\, S_L \mathcal{S}[\![\, \{x_2; \ldots; x_n;\} \,]\!]\, S_L = \ldots \\ &= \mathcal{S}[\![\, x_1 \,]\!]\, S_L \ldots \mathcal{S}[\![\, x_n \,]\!]\, S_L\end{aligned}$$

If variables are present, they must be evaluated first, but the remaining statements can also be reduced in any order. Similarly, in node constructors, the attribute initializer can be reduced independently of the child initializer.

This allows us to provide random access to the transformation result via a standard result tree interface. The actual tree is initially virtual. When a node is first accessed, we partially reduce the creating constructor $\mathcal{S}[\![\, \texttt{new}\, n(b_a, b_c) \,]\!]\, S_L$ to a local node $(n, A, \mathcal{S}[\![\, b_c \,]\!]\, S_L)$. When node children are first accessed, we partially reduce the child initializer $\mathcal{S}[\![\, b_c \,]\!]\, S_L$ to a sequence of constructor calls $\mathcal{S}[\![\, \texttt{new}\, n_i(b_{ai}, b_{ci}) \,]\!]\, S_{Li}$. As constructors create exactly one node, child names and order are now known, and navigation is possible.

4. DESIGN

Here, we describe our implementation of lazy XML transformations in more detail. We call our approach *demand-driven* interpretation.

The idea behind demand-driven interpretation is based on the observation that the consumer has to access the result tree via interface operations anyway. This allows to mask a *virtual result tree* behind that interface and to pretend that the complete result tree exists in reality. Besides some inner fragments of the final result tree, the virtual result tree may contain *bound instructions* at its leaves. Bound instructions are pairs of instruction and execution state, thereby carrying all information necessary to restart the transformation process for yet virtual fragments of the result tree.

Resuming and suspending the transformation process is handled transparently by the interface methods. When the consumer enters yet virtual regions of the result tree, the respective portions have to be materialized on demand. For this purpose, the transformation process is restarted at the bound instruction and continued until the required portions of the result tree have been produced. This can of course only happen by executing output instructions, because flow control instructions and variable definitions do not produce any parts of the result tree by themselves. After creating the required node, computation can be suspended again by storing a bound instruction that represents the remaining fragment.

All of this happens transparently, i.e., the consumer does not notice that yet unvisited portions of the result tree possibly do not exist but are computed on demand instead. For him, the complete result tree appears to exist right from the start.

In our prototype implementation, we decided to adhere strictly to the XPath data model. This means that the internal representation of our tree complies with the abstractions and relationships defined by the XPath model. Our external interface consists of a collection of atomic navigation primitives and access methods, which constitute a qualified basis for the construction of a sophisticated XPath engine on top of it. Adaptation to other standard interfaces like DOM or SAX for example can be accomplished by straightforward wrappers.

More specifically, the transformation process is driven by the navigation steps retrieving the first child, and the subsequent sibling, i.e., `getFirstChild()` and `getNextSibling()`.

Each time one of these methods is called with respect to a context node, they have to check whether the requested node has already been materialized or whether the result tree along this axis is merely virtual and computation is still pending. In the latter case, the transformation process must be reactivated and continued until the requested node has been created by an output instruction. Then, computation can be suspended again.

As lazy transformation has to proceed from the root node of the result tree downward, accessing the parent of a given a node is always safe, i.e., the `getParent()` method is guaranteed to access a real node and thus does not have to check for bound instructions.

The same is true for the `getPrevSibling()` method because construction of child nodes proceeds form the first child node to the following siblings. Needless to say, methods accessing local node properties are inherently safe because one of the navigation methods must have reached that node before.

Our interpreter complies strictly with the W3C recommendation for the XSLT 1.0 language. Except for result tree fragments, which are currently unsupported as variable types, our implementation fully supports all XSLT data types.

5. MEASUREMENTS

How does the performance of our lazy XSLT interpreter compare to traditional interpreters?

Because we avoid to construct parts of the result tree that are never accessed, we expect our performance to be superior for sparse coverage. However, as demand-driven suspension and resumption of the transformation process requires additional bookkeeping, we expect these overheads to worsen performance for increasing coverage.

To validate these expectations, we have to measure execution times for a full range of coverage. In these data, we are particularly looking for the following three figures.

The *setup time* is the execution time for 0% coverage. It captures the interval until the root node of the transformation result becomes available for the first time.

The *break-even point* is characterized by the coverage where the lazy and traditional approaches show equal performance.

The runtime difference for full coverage indicates the *overhead* that is imposed upon by the lazy evaluation strategy.

5.1 Experimental Setup

We want to compare performance for lazy and traditional interpretation over a full range of coverage. As human interaction is unsuitable for exactly repeatable measurement runs, we chose to generate test cases automatically. Fig. 4 illustrates our setup that automates access to the transformation result by employing a second transformation instead. For this purpose, we use a partial identity copy that can be parameterized by the fraction of the source tree that is to form the result tree. The transformation is computationally simple and traverses only the fragment of the source tree that it actually copies. In this way access profiles over a full range of coverage can be simulated.

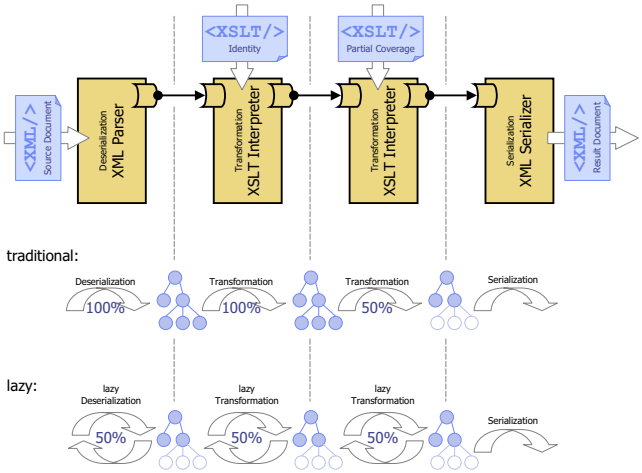

Figure 4: Measurement setup

5.2 Competitors

Benchmarking the lazy approach requires a set of competitors that correspond closely to our implementation with respect to all performance-critical properties except the evaluation strategy itself. For this purpose we are looking for traditional XSLT interpreters written in Java. We confine ourselves to freely available open-source implementations in order to be able to verify their internal workings.

There are three widely known interpreters that meet our criteria. XT [5], written by James Clark, was the first XSLT interpreter available, and is known to be the fastest Java implementation available [14]. Unfortunately, we could not determine whether it supports chaining of transformations without serializing intermediate results to files. As this additional step would unfavorably distort the measurements for XT, we excluded it from consideration as a candidate. Michael Kay's Saxon [13] is known for its strict coverage of the language recommendation and also shows very good performance. Xalan Java [8] is a part of the Apache Software Foundation's XML Project. It also sticks strictly to the language specification, but offers rather poor performance. Both interpreters support event-based chaining of transformations via the Java API for XML Processing (JAXP) [24] interface. Therefore, our reference candidates are Saxon Version 6.5.2 from Michael Kay and Xalan Java Version 2.2.D11, the latter being part of Sun's J2SE 1.4.0 platform distribution.

5.3 Test Cases

Our benchmark comprises three test cases. The first, *identity*, performs an identity copy of a large source document. It is computationally simple with respect to the size of the result tree. We therefore expect a moderate growth in runtime for increasing coverage. On the other hand, parsing a 2.4 MB source document should account for a significant share of the total runtime and for comparatively high setup times.

The second test case, *generated*, contrasts these properties by producing a large result tree from a comparatively small source document. It unfolds the element names found in the source document by constructing a binary tree thereof. In this way the result tree is produced primarily by compu-

tation whereas parsing costs remain negligible. We therefore expect the setup time for lazy interpretation to be near zero and the slope for increasing coverage to be significantly steeper than in the first test case.

The last test case, *infinite*, produces a binary branching result tree with infinite depth, thereby providing for a reference transformation that can be performed by lazy evaluation only whereas traditional depth-first evaluation fails. It merely serves as a proof of concept by demonstrating that lazy evaluation is inherently more powerful due to the potential to operate on infinite data structures.

5.4 Test Environment

All measurements were taken on a 2 GHz Pentium IV with 512 MB memory, running Java 2 SDK, Standard Edition, Version 1.4.0 under Windows XP Home Edition.

5.5 Results

The runtime figures given here are mean values over five subsequent runs as measured by the system clock. The Java Virtual Machine's just-in-time compilation was triggered by an initial blind run prior to the actual measurements, the coverage could be computed from the lazy processing chain by recording the number of created nodes.

Fig. 5 shows the runtimes for the first test cases, *identity*. The results confirm our expectations. In both cases, the traditional interpreters show the same characteristics with Saxon being substantially faster than Xalan Java. We therefore consider Saxon representative for traditional interpretation in the following. Lazy transformations were performed in combination with both a traditional and a lazy parser respectively. The best setup-times were measured for lazy parsing, but even for traditional parsing, Saxon requires two times the setup time of our lazy interpreter. This advantage declines continuously for increasing coverage, with a break-even point at around 95% coverage and a slight overhead for full coverage.

Fig. 6 shows the runtimes from test case 2, *generated*. Due to the fact that there are nearly no parsing costs in this case, either parsing strategy yields the same results and setup

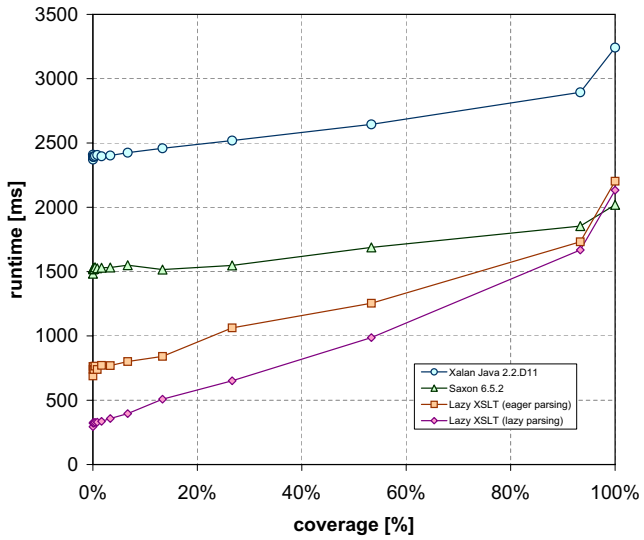

Figure 5: Execution times for the *identity* test case

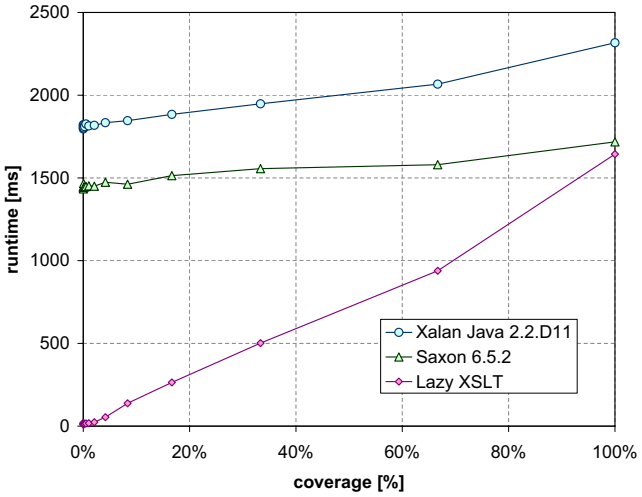

Figure 6: Execution times for the *generated* test case

time is negligible for lazy interpretation. For full coverage, lazy interpretation is roughly on a par with Saxon and there is neither a break-even point nor an overhead. Xalan Java is again the slowest competitor.

Traditional processors enter an infinite loop for the third test case, *infinite*. Although the subsequent transformation only queries a depth-limited fraction of its input, the initial transformation descends forever, spewing out copies of the original tree in depth-first order. In contrast, our lazy processor generates only those nodes actually requested by the second transformation. It therefore terminates.

We do not give runtime figures for the last test case, as the resulting graph is rather uninteresting. Infinite execution time does not fit on the scale, and visiting a finite number of nodes from an infinite tree would yield 0% coverage no matter how many nodes are visited. However, for our lazy processor, plotting execution time over the absolute number of nodes visited instead of percentual coverage yields a figure identical to fig. 6.

5.6 Summary

Lazy interpretation of XSLT is not a purely theoretical concept. Our implementation profits extremely well from low coverages. Especially when paired with a lazy parser, our setup time is negligible. Execution time increases linearly with coverage. Break-even with the baseline established by Saxon occurs near 95% coverage. The overhead at full coverage appears to be small.

6. CONCLUSIONS

Traditional XSLT interpreters consume input documents using random access, but produce output strictly in document order. In cascaded transformations, this means that the previous processing step must terminate completely before the next may begin. To that end, the intermediate result must be stored in an unbounded buffer.

Rooted in a theoretical foundation of confluence and multiple reduction alternatives, our lazy XSLT breaks this mould. We support random access on partial transformation results. In pipeline fashion, this allows later processing steps to produce output before their predecessors have terminated.

Traditional transformation languages for line-oriented text can do the same. However, they cannot skip data because they both consume and produce sequentially. In contrast, our approach propagates partial input coverage backwards through the processing pipeline. That is, if later processing steps disregard parts of an intermediate result, those parts are never produced by the earlier processing steps.

Measurements confirm that this propagation of coverage yields performance benefits. While our implementation is competitive with traditional ones for full coverage, we realize significant performance gains as coverage decreases.

Transitive queries on a transformation result, e.g., `.//p`, remain expensive because the entire subtree must be traversed, which destroys coverage propagation. In many cases, the desired nodes are confined to distinct portions of the subtree. However, lazy evaluation for one input document cannot generate such assertions over all input documents. Propagating and using such assertions to simplify queries lies in the domain of static evaluation for known input types [16], whose implementation is currently approaching completion.

For low-coverage scenarios, purely lazy pipelines seem promising. They would combine lazy transformation with lazy parsing [19], and conceivably with lazy type checking. The latter is a subject for future research.

As human access profiles are probably the most common source of partial coverage, lazy viewers are an interesting continuation of our approach. Current browsers can format and display the available document fragment while downloading the remainder. This is essentially sequential processing, which imposes full coverage. A lazy viewer could propagate partial coverage to the pipeline on the server.

Beyond performance, lazy evaluation also extends the design space for XSLT programs. As in functional programming, composed XSLT programs may create conceptually infinite intermediate structures, provided that later processing stages only access a finite part of them. Our implementation already supports this flexible paradigm today.

Currently, composed XSLT transformations are still confined to content management systems, which handle rather finite data. It remains to be seen wether the evolving usage of XSLT will result in as important a role for infinite intermediate structures in XSLT programming as they enjoy in contemporary functional programming.

7. REFERENCES

[1] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. *XML with Data Values: Typechecking Revisited.* ACM Press, Proc. of the 20th ACM Symposium on Principles of Database Systems (PODS '01), May 2001, Santa Barbara, California, USA, `http://citeseer.nj.nec.com/article/alon01xml.html`, 2001.

[2] J. R. Ambrosiak. *Gregor XSLT Compiler.* Ambrosoft, Inc., `http://www.ambrosoft.com/gregor.html`.

[3] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Siméon. *XQuery 1.0: An XML Query Language.* W3C Working Draft 15 November 2002, `http://www.w3.org/TR/2002/WD-xquery-20021115/`, 2002.

[4] A. Church and J. Rosser. *Some properties of conversion.* Transactions of the American Mathematical Society, 36(3):472-482, May 1936, 1936.

[5] J. Clark. *XT.* `http://www.jclark.com/xml/xt/`.

[6] T. U. Consortium, editor. *The Unicode Standard, Version 3.0.* Addison-Wesley, 2000.

[7] J. Cowan and R. Tobin. *XML Information Set.* W3C Recommendation 24 October 2001, `http://www.w3.org/TR/2001/REC-xml-infoset-20011024`, 2001.

[8] A. S. Foundation. *Xalan Java XSLT Processor.* Apache XML Project, `http://xml.apache.org/xalan-j/`.

[9] A. S. Foundation. *XSLTC Compiler.* Apache XML Project, `http://xml.apache.org/xalan-j/xsltc/`.

[10] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing xpath queries. Hong Kong, 2002.

[11] G. Gottlob, C. Koch, and R. Pichler. *Efficient Algorithms for Processing XPath Queries. Extended version.* Proc. of the 28th International Conference on Very Large Data Bases (VLDB 2002), Aug. 2002, Hong Kong, China, `http://www.dbai.tuwien.ac.at/staff/koch/download/vldb2002-post.pdf`, 2002.

[12] G. Gottlob, C. Koch, and R. Pichler. *XPath Query Evaluation: Improving Time and Space Efficiency.* Proc. 19th IEEE International Conference on Data Engineering (ICDE'03), Mar. 2003, Bangalore, India, `http://citeseer.nj.nec.com/gottlob03xpath.html`, 2003.

[13] M. H. Kay. *Saxon: Anatomy of an XSLT processor.* IBM developerWorks, `http://www-106.ibm.com/developerworks/library/x-xslt2/`, 2001.

[14] E. Kuznetsov and C. Dolph. *XSLTMark 2.0.* DataPower Technology, Inc., `http://www.datapower.com/XSLTMark/`, 2001.

[15] A. P. Kwong and M. Gertz. *Schema-based Optimization of XPath Expressions.* Submitted for conference publication, `http://citeseer.nj.nec.com/551000.html`, 2002.

[16] W. Löwe and M. L. Noga. *Scenario-Based Connector Optimization. An XML Approach.* Springer Verlag, LNCS 2370, IFIP/ACM CD 2002, 2002.

[17] D. Megginson. *Simple API for XML Processing (SAX).* SAX Project, `http://www.saxproject.org/`, 1998.

[18] T. Milo, D. Suciu, and V. Vianu. *Typechecking for XML Transformers*. ACM Press, Proc. of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS), pages 11–22, `http://citeseer.nj.nec.com/milo00typechecking.html`, 2000.

[19] M. L. Noga, S. Schott, and W. Löwe. *Lazy XML Processing*. ACM Press, Proc. of the ACM Symposium on Document Engineering (DocEng) 2002, pages 88–94, Nov. 2002, McLean, VA, USA, 2002.

[20] A. Novoselsky and K. Karun. *XSLTVM - an XSLT Virtual Machine*. XML Europe 2000, Paris, France, 12-16 June 2000, `http://www.gca.org/papers/xmleurope2000/papers/s35-03.html`, 2000.

[21] D. Olteanu, H. Meuss, T. Furche, and F. Bry. *Symmetry in XPath*. Universität München, Lehr- und Forschungseinheit für Programmier- und Modellierungssprachen, Forschungsbericht, `http://www.pms.informatik.uni-muenchen.de/publikationen/`, 2001.

[22] D. Olteanu, H. Meuss, T. Furche, and F. Bry. *XPath: Looking Forward*. Springer Verlag, LNCS 2490, Proc. of the EDBT Workshop on XML Data Management (XMLDM) 2002, pages 109–127, `citeseer.nj.nec.com/olteanu02xpath.html`, 2002.

[23] S. Schott. *Optimierung der Zerteilung von XML für Transformationen*. Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, Studienarbeit, 2001.

[24] *The Java^{TM} API for XML Processing. Version 1.0*. Sun Microsystems Inc., `http://java.sun.com/xml/jaxp/`, 2001.

[25] A. Tozawa. *Towards static type checking for XSLT*. ACM Press, Proc. of the ACM Symposium on Document Engineering (DocEng) 2001, pages 18–27, Atlanta, Georgia, USA, `http://doi.acm.org/10.1145/502187.502191`, 2001.

[26] *Document Object Model*. W3C, `http://www.w3.org/DOM/`, 2000.

[27] *Extensible Markup Language (XML) 1.0*. W3C Recommandation, `http://www.w3.org/TR/1998/REC-xml-19980210`, 1998.

[28] *Namespaces in XML*. W3C Recommandation, `http://www.w3.org/TR/1999/REC-xml-names-19990114`, 1999.

[29] *XML Path Language*. W3C Recommandation, `http://www.w3.org/TR/xpath`, 1999.

[30] *XML Schema Part 1: Structures*. W3C Recommendation 2 May 2001, `http://www.w3.org/TR/2001/REC-xmlschema-1-20010502`, 2001.

[31] *XSL Transformations (XSLT)*. W3C Recommandation, `http://www.w3.org/TR/xslt`, 1999.

[32] P. Wadler. A formal semantics of patterns in XSLT, 2000.