

# Size Relationships in XML Type Checking

Zhendong Su

<su@cs.ucdavis.edu>

Gary Wassermann

<wassermg@cs.ucdavis.edu>

Department Computer Science  
University of California, Davis  
Davis, CA 95616-8562 USA

## ABSTRACT

XML transformation languages (*e.g.*, XSLT) take an XML document as input and produce another XML document as output. It is useful to know *statically* that such transformations always produce valid documents, for static debugging of the transformation program or for eliminating dynamic checks on the output documents. This gives rise to the *XML type checking problem*. Type- and automata-theoretic techniques have been proposed to address this type checking problem, exploiting XML's tree structure. However, existing approaches are not capable of reasoning about *size* information of produced XML documents, such as that two locations in the output documents always have the same number of elements. This paper presents a type-based inference system to discover size relationships in output documents from XML transformation programs through refined type checking. For example, our system can identify program fragments producing the same number of elements for all input documents. The novel aspects of our system are techniques to deal with the rich tree structure of XML types (*i.e.*, schemas), whereas array analyses (*e.g.*, bounds checking) for languages such as C deal with flat arrays. In this paper, we present our type system and give a sketch of its soundness.

## 1. INTRODUCTION

Since XML [9] became a W3C recommendation in 1998, XML has been increasingly accepted as the standard format for electronic data exchange. Two parties who wish to exchange data generally organize their data differently. Thus, one or both of the parties must transform their data so that it is suitable for the other to use. In the context of XML, "schemas" (*e.g.*, XML Schema) [21] are used to specify data organization. When data is exchanged using XML, the recipient specifies a schema to which all received XML documents must conform. The sender must write a transformation program to convert data from his own schema to the recipient's schema. If the sender can determine that his program performs the transformation correctly, no runtime

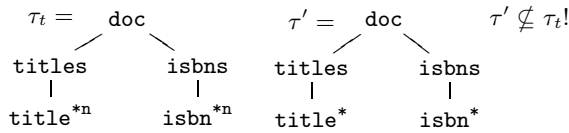
checks are necessary.

This gives rise to the XML type checking problem. Let  $T$  be the set of all XML documents ( $T$  is a mnemonic for "trees"; all XML documents have a tree structure). An XML type is a subset of all documents:  $\tau \subseteq T$ , often called a *schema*. The XML type checking problem asks, for source and target types  $\tau_s$  and  $\tau_t$  respectively, and transformation program  $P$ , is it true that  $\forall x \in \tau_s. P(x) \in \tau_t$  [25]? One common approach to answer this question is based on type inference: an output type  $\tau'$  is conservatively inferred based on the program and the source type:  $P(\tau_s) \subseteq \tau'$ . If the inferred type is a subtype of the target type,  $\tau' \subseteq \tau_t$ , then the program successfully type checks.

We introduce the notion of sizes in XML documents and types: a *size* denotes the number of XML elements and/or scalars in a consecutive sequence under a common parent. For a particular XML document, sizes are always known constants. However, sizes may not remain constant across all documents conforming to a single type. In this case, the sizes of the type are represented by variables, which may be constrained to allow only values valid for some document within the type. When some sizes of a type are constrained in terms of other sizes (currently not supported in XML Schema), we call those *size relations*. Because of the common use of Kleene stars in types, it is generally impossible to discover the values of sizes. Rather, we aim at discovering relationships among sizes in output documents.

Some practical settings require size information. For example, in a document with parallel lists of movie titles and the years those movies were made, the length of those lists can vary provided that they are equal to each other. Alternatively, consider a specification manual that must include the same information in multiple languages. The number of headings in one linguistic section can vary provided that it equals the number of headings in every other linguistic section. Size relationships arise in settings that include parallel or repeated data. The ability to infer size relationships may find application in ensuring the correct composition of Web services [7].

No previous technique for XML type checking can accurately type check a program when the target type has size relations and the program output is not confined to a regular subtype of the output type. The decidability of XML type checking has been established using  $k$ -pebble tree transducers when



**Figure 1: A target type with size relations. Conservatively inferred types using existing techniques cause correct programs to fail to type check.**

no size relations are present [20]. It is unclear how well these automata-based techniques would work in practice because of their high computational complexity, and more fundamentally, how to incorporate size information into these formalisms to retain decidability of type checking. Existing type-based approaches [8] may provide more practical, if less precise, solutions. However, currently these approaches are unable to infer types with size relations. The main contribution of our paper is a type-based inference system to discover size relations for XML transformation programs. To the best of our knowledge, ours is the first system capable of reasoning about size relations for XML transformations.

Several languages have been proposed for XML transformations, including XSLT [6], XQuery [8], XDuce [14], CDuce [2], HaXml [27], and Relaxer [11]. The XML transformation language in this paper used to explain our technique has much of the expressive power of these languages. It includes iterations over subtrees, pattern matching based on tag or type, conditional expressions, etc. Our language is introduced in Sect. 2.

For illustration purposes, we consider first the following XQuery program:

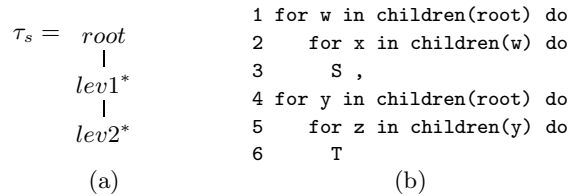
```

<doc>
  <titles>
    for $a in document("cat.xml")//catalog/book/title
      return $a
  </titles>
  <isbns>
    for $b in document("cat.xml")//catalog/book/isbn
      return $b
  </isbns>
</doc>

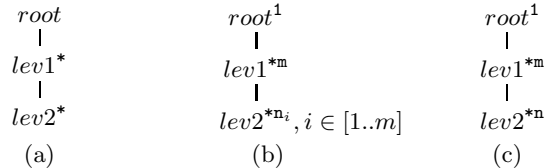
```

The program takes an XML catalog of books and creates an output document with lists of titles and ISBNs, perhaps for easy ISBN lookup by title in a printed listing. Because each book has exactly one title and ISBN, the lists rooted at `titles` and `isbns` must have the same number of elements—that is, they must have the same size. The programmer would like to confirm that this size relationship holds.

Fig. 1 gives the output type  $\tau_t$  (omitting the scalar children of `title` and `isbn`) as the programmer intends it. We portray types as trees to provide a conceptual view of the tree structure of XML types. Using existing type inference methods, the type  $\tau'$  would be inferred. Because  $\tau' \not\subseteq \tau_t$ , this correct program fails to type check. Given the current practice in data transformation, this situation is relatively common.



**Figure 2: A source type with nested repetitions.**



**Figure 3: Two ways to annotate a source type  $\tau_s$ .**

### 1.1 Difficulties with Inference of Size Relations

At first consideration, it may seem as though the use of integer constraints, which enable array analyses in languages such as C, would be sufficient for inferring size relationships. Surprisingly, it is not that simple. The main problem is that because XML transformations operate on trees, a very rich data structure, size relationship inference must interrelate tree sub-structures. Standard array analyses, however, need only reason about how size information for arrays, a flat data structure, flows in C-like programs.

Consider the source type  $\tau_s$  and program shown in Fig. 2. Lines 1–3 and 4–6 of the program in Fig. 2 have the same semantics as `/root/~/~` (where `~` is a wildcard that matches any tag), except line 3 substitutes an `S` for whatever the output would have been, and equivalently with `T` on line 6. In general, the semantics of paths can be achieved through nested `for` and `case` expressions. For example, in Fig. 11, lines 1–10 match the semantics of `/catalog/book/title`.

Clearly this program produces the same number of `S`'s as `T`'s. However, standard type systems perform a modular analysis, using only the types of subexpressions and some global type environments for type checking and inference. Therefore, in discovering a relation, such as size equality between two expressions, the type system is restricted to using information it has available at the time of typing both expressions: the input type. It must discover relations between the expressions and the input type to relate the expressions. Suppose that in hoping to discover the size relationships precisely, we annotate  $\tau_s$  as in Fig. 3b, where  $1$ ,  $m$ , and  $n_i$  are size annotations for the corresponding types.

We argue that the precision aimed at cannot be achieved. The `for` expression has the form `(for x in e1 do e2)`. The expression  $e_1$  evaluates to a list, and for each element `a` in that list,  $x$  gets bound to `a` and  $e_2$  gets executed. There are two approaches to typing the `for` expression. We first look at the approach used in XQuery's type system [8]: first,  $x$  is bound to the union of the unit types in  $e_1$ 's type,  $\tau_1$ , and

$$\begin{array}{ccc}
\tau_1 = lev1^{*m} & \vdots & \tau_u = lev1^1 \\
| & & | \\
lev2^{*n_i}, i \in [1..m] & \vdots & lev2^{*n_1} | \dots | n_m \\
\cdots & & \cdots \\
\tau_{1b} = lev2^{*n_1} | \dots | n_m & & \\
\Rightarrow \tau_{1b} = lev2^{*r}, \{\min(n_i) \leq r \leq \max(n_i)\} & & \\
\cdots & & \cdots \\
\tau_{ub} = lev2^1 & \vdots & \tau_{2b} = S^1 \\
\cdots & & \cdots \\
\tau' = S^{*p}, \{\min(m \times n_i) \leq p \leq \max(m \times n_i)\} & & 
\end{array}$$

**Figure 4: Some inferred types in our first attempt to infer all size relationships precisely.**

$$\begin{array}{cccc}
\tau_{u_1} = lev1^1 & \tau_{u_2} = lev1^1 & \tau_{u_3} = lev1^1 & \dots \\
| & | & | & \\
lev2^{*n_1} & lev2^{*n_2} & lev2^{*n_3} & 
\end{array}$$

**Figure 5: Some inferred types in our next attempt to infer all size relationships precisely.**

$e_2$  is typed once with  $x$  having that binding. Then type constructors are added to the inferred type based on their occurrences in  $\tau_1$ .

In inferring a type for the program in Fig. 2b, if  $\tau_s$  is annotated as shown in Fig. 3b, the type of expression `children(root)` on line 1 is  $\tau_1$ , as shown in Fig. 4. Fig. 4 also shows the rest of the types we refer to in this paragraph. Suppose we find the type we will assign to  $w$  within the body of the `for` expression by taking the union of the unit types in  $\tau_1$ , as done in XQuery’s type system. This yields  $\tau_u$ . The type of `children(w)` on line 2 is then  $\tau_{1b}$ , however, it is not clear what this size annotation means. To add clarity, we rewrite it as shown in Fig. 4. We can now find the unit type to which  $x$  gets assigned as  $\tau_{ub}$ , and so the body of the inner `for` expression on line 3 is typed as  $\tau_{2b}$ . We then go back up and compose the type  $\tau_{2b}$  with  $*r$ . When going up again to find the type of the `for` expression on line 1, we compose the type of the nested `for` expression with  $*m$ . The result is  $\tau'$  (for clarity,  $r$  has been simplified out of the constraints).

Unfortunately, all we know about  $p$ ’s value is that it is confined to a given range. When the `for` expression on line 4 is typed, the result will also be a starred type whose size is confined to the same range. Knowing that two numbers are in the same range is usually not sufficient to relate the two numbers concretely (e.g., describing one as a function of the other). Thus, this approach is not suitable for discovering interesting size relations.

The second approach to typing the `for` expression is the one taken by Fernandez *et al.* [10]: find a type for the body of a `for` expression for every named unit type in  $\tau_1$  and combine them based on the type structure of  $\tau_1$ . Given the annotation for  $\tau_s$  in Fig. 3b, the type of `children(root)` is again  $\tau_1$ , as shown in Fig. 4. Because the number of children ( $n_i$ ) may be different for each of the  $m$  `lev1`’s, the first unique unit type here is  $\tau_{u_1}$ , as shown in Fig. 5. The second is  $\tau_{u_2}$ , the third is  $\tau_{u_3}$ , and so on. Trying to infer a type for the `for` expression by inferring a type for  $e_2$  based on  $\tau_{u_1}, \tau_{u_2},$

$\tau_{u_3}, \dots$ , is cumbersome and requires complicated symbolic reasoning about summations such as  $\sum_{i=1}^m n_i$ .

Why cannot we get precise size information through precise source type annotations? When a type element in a tree type has a Kleene star (e.g., `lev1*` in Fig. 2), its children in the type tree (e.g., `lev2*`) represent uniformly all lists of child elements of the starred element in an actual document. Adding precise size annotations to Kleene starred elements (e.g., `lev2^{*n_i}`) of the input type distinguishes within the type tree the concrete lists that the Kleene starred element represents. After distinctions have been added to the input type, either the type system “factors out” the distinctions, resulting in a loss of precision (as shown in Fig. 4), or in addressing the distinctions directly, the type system faces increased complexity (as shown in Fig. 5). In this paper, we present an approach to overcome these problems. Next we briefly discuss our approach.

## 1.2 Our Approach

The key insight of our approach is that the elements of a list are usually treated uniformly, both in the type and in the program, so the only information we need is the *total* number of elements in a concrete tree represented by an element (or more precisely, by a path) in the type tree. In some XML transformation languages, there is no mechanism to access the elements in a list non-uniformly. For example, it is impossible to remove the first element from a given list. In other languages (e.g., CDuce), constructs that handle list-elements non-uniformly can be typed conservatively. We take advantage of this in our analysis. We annotate the source type as shown in Fig. 3c. The annotation  $n$  on `lev2` in Fig. 3c denotes the total number of `lev2` elements in the input document that have as parent a `lev1` element and as grandparent a `root` element. Note the difference between this annotation and a size: the elements may not all have a common parent. For an alternating sequence of `for` and `case` expressions that match the semantics of `/root/lev1/lev2`,<sup>1</sup> the body of the innermost `case` will be executed  $n$  times. Consequently our type system multiplies the size of the innermost `case` expression by  $n$  to find the size of the outermost `for` expression.

Because our annotations do not introduce distinctions into  $\tau_s$ , we avoid the trouble shown in Fig. 5. We therefore leverage the more powerful second approach to typing the `for` expression. The union operation used in the first approach loses information whenever an element type has more than one child type and so cannot achieve the precision necessary to infer size relationships.

Conditional expressions are often used to select certain elements of a list and pass over others, so they, too, influence the sizes of output types. A solution that leads to sizes confined to ranges has the same problems as discussed in Sect. 1.1, but unless all parts of the boolean expression are static, we cannot determine statically which branch of the conditional will be executed. To address this we use *pair* types. Like conditional types [1], pair types preserve the relationship between the types of the branches of conditional

<sup>1</sup>The first and second halves of the program in Figure 2 would match this semantics if the appropriate `case` expressions were inserted.

tag	$a$
variable	$x$
constant	$n ::= c_{\text{int}} \mid c_{\text{str}} \mid c_{\text{bool}}$
operator	$op ::= + \mid - \mid \text{and} \mid \text{or}$ $\mid = \mid <$
expression	$e ::= n \mid x \mid a[e] \mid e, e$ $\mid () \mid e \text{ op } e \mid \text{let } x = e \text{ do } e$ $\mid \text{for } x \text{ in } e \text{ do } e \mid \text{children}(e)$ $\mid \text{case } e \text{ of } x:p \Rightarrow e \mid x \Rightarrow e \text{ end}$ $\mid \text{if } e \text{ then } e \text{ else } e$ $\mid f(e; \dots; e)$
pattern	$p ::= a \mid \sim \mid s$
data	$d ::= n \mid a[d] \mid d, d \mid ()$

Figure 6: XML transformation language.

tag	$a$
type name	$x$
size type	$r ::= c \mid n$
scalar type	$s ::= \text{String} \mid \text{Boolean}$ $\mid \text{Integer}$
unit type	$u ::= a[\tau] \mid \sim[\tau] \mid s$
type	$z ::= x \mid \tau, \tau \mid \tau^* \mid ()$ $\mid <\tau, \tau> \mid \tau \mid \tau \mid \emptyset$
annotated type	$\tau ::= u^1 \mid z^r$

Figure 7: Type language.

expressions and **true** and **false** evaluations of the boolean expression. We relate the size of a pair type to the sizes of the conditional expression’s **true** and **false** branches as well as to the identity of the boolean condition. If, in a post-processing phase, two boolean conditions can be found to be equivalent, then it becomes possible to relate the sizes of the corresponding conditional expressions.

## 2. THE SOURCE LANGUAGE

Figure 6 gives the syntax of our XML transformation language. Most of the constructs are standard. The expression  $a[e]$  constructs XML elements. Paths can be expressed through **for** and **case** expressions. We omit the expression to select the parent of an element, which can be typed conservatively, as is done in other XML transformation languages with type systems, such as XQuery. Beyond that, our language does not include, for example, sorting, explicit type casts, and modules. We do not expect much difficulty in extending our technique to cover these language constructs.

Note that the **case** expression matches a value against a  $p$ , defined by the “pattern” derivation (which parallels the “unit type” derivation in Figure 7). Also, as shown by the “data” derivation, we denote XML elements as **tag**[...] rather than  $\langle \text{tag} \rangle \dots \langle / \text{tag} \rangle$  to simplify notation. The dynamic semantics for this language is standard, but the companion technical report gives a complete presentation [24].

## 3. TYPE LANGUAGE FOR SIZE INFERENCE

Fig. 7 gives our type language. The “size type” shows that either a constant or a variable can be used as a size annotation to a type. The size annotation denotes the number

expression	$e ::= c \mid n \mid (\Gamma_\pi(\pi)) \mid (e)$ $\mid e + e \mid e \times e$
path	$\pi ::= \pi/a \mid \varepsilon$
constraint	$C ::= C \cup \{n = e\} \mid C \cup \{\pi\} \mid \emptyset$

Figure 8: Constraint language.

of unit values that may be matched to the annotated type. The wildcard unit type,  $\sim[\tau]$ , is defined such that  $a[\tau]$  is a subtype of  $\sim[\tau]$  for all tags  $a$ , following Fernandez *et al.* [10].

Among the types  $z$ , “ $\tau, \tau$ ” is the type of two values in sequence. The union type is “ $\tau \mid \tau$ ”; a value whose type is either of the choices matches it. We introduce the pair type, “ $<\tau, \tau>$ ,” for typing conditional expressions. Like the choice type, a value of either of the two types may match it. Unlike the choice type, the order of the two types is preserved, *i.e.*,  $(\tau_1 \mid \tau_2) = (\tau_2 \mid \tau_1)$ , but  $<\tau_1, \tau_2> \neq <\tau_2, \tau_1>$ , when  $\tau_1 \neq \tau_2$ . Because the order is preserved, it is possible to reason about the types of different conditional expressions in relation to each other. The “ $\emptyset$ ” is an identity for choice types, and is needed for typing repetition expressions.

We also have a constraint language to capture size relations. Figure 8 shows our constraint language. There are two kinds of constraints. The first kind consists of equality constraints between a size variable and an arithmetic expression. The function  $\Gamma_\pi$  maps paths to size variables/constants. The second kind of constraint is a path,  $\pi$ , which is not explicitly related to anything else in the constraints or types. Paths remain in the constraint set only temporarily during the typing of **for** expressions that match the semantics of paths. Paths, the function  $\Gamma_\pi$ , and the connection between them are explained in Section 4.3.

## 4. TYPE RULES

We use a constraint-based formulation of our type system. The type judgment  $\Gamma \vdash e : \tau, C$  is read: in environment  $\Gamma$ , expression  $e$  has type  $\tau$ , where the size variables in  $\Gamma$  and  $\tau$  are subject to the constraints  $C$ . Type environments are defined by the following grammar:

$$\Gamma ::= \emptyset \mid \Gamma \uplus \{x : \tau\} \mid \Gamma \uplus \{\text{for } x : \tau\}$$

where  $\{\text{for } x : \tau\}$  is used in typing the **for** expression. A type environment,  $\Gamma$ , maps variables and “**for**” variables to types according to the following rules:

$$\begin{aligned} \Gamma \uplus \{x : \tau\}(x') &= \tau && \text{if } x = x' \\ &= \Gamma(x') && \text{otherwise} \\ \Gamma \uplus \{\text{for } x : \tau\}(x') &= \tau && \text{if } x = x' \\ &= \Gamma(x') && \text{otherwise} \end{aligned}$$

Due to space constraints, we explain here the typing of the three most interesting expressions to size types in increasing order of difficulty. The complete list of type rules can be found in the companion technical report [24]. In the type rules that we discuss next,  $z$ ,  $u$ , and  $\tau$  are as in Fig. 7:  $z$  is a type without a size annotation,  $u$  is a unit type without an annotation, and  $\tau$  is a type with an annotation.

In Section 4.1, we explain the type rule for sequence expressions, in which two subexpressions are put in sequence. In

Section 4.2, we explain the type rule for conditionals. In Section 4.3, we explain the typing of the `for` expression, which is the most involved because it is the main language construct used to produce subtrees of unknown size.

## 4.1 Sequence Expressions

The type rule for sequence expressions is as follows:

$$\frac{\Gamma \vdash e_1 : \tau_1, C_1 \quad \tau_1 = z_1^{m_1} \quad \tau_2 = z_2^{m_2} \quad \Gamma \vdash e_2 : \tau_2, C_2 \quad \mathbf{n} \text{ is fresh}}{\Gamma \vdash e_1, e_2 : (\tau_1, \tau_2)^{\mathbf{n}}, C_1 \cup C_2 \cup \{\mathbf{n} = m_1 + m_2\}}$$

This rule is straightforward: the number of XML elements produced by the sequence expression as a whole is the sum of the numbers of XML elements produced by its subexpressions. The rule adds the constraint that  $\mathbf{n}$ , the size of the sequence expression, equals  $m_1 + m_2$ , the sum of the sizes of the subexpressions.

## 4.2 Conditional Expressions

Our type system allows the `true` and `false` branches of a conditional expression to have different types. The type of the conditional expression in most type systems is a choice type composed of the types of the branches:  $(\tau_1 | \tau_2)$ . The loss of precision from this approach poses a problem: we can determine that the value of the size variable for the conditional expression is within the range of the sizes of its branches, but we can no longer conclude that two sizes are equal. We address this by means of a *pair type*, introduced in Section 3, plus related constraints.

The main idea is this: if two different `if` expressions have the same boolean condition with equivalently bound variables and get executed the same number of times in one run of the program, then their `true` and `false` branches get executed the same number of times respectively. A unification analysis can conservatively determine which variables have the same binding. A straightforward syntactic analysis can conservatively determine which boolean expressions are equivalent and are executed the same number of times. All `if` expressions are given labels, and two `if` expressions will have the same label if and only if their boolean expressions were found to be equivalent.

Our type rule for conditional expressions is as follows:

$$\frac{\Gamma \vdash e_b : \text{Boolean}^1, C_b \quad \Gamma \vdash e_1 : \tau_1, C_1 \quad \Gamma \vdash e_2 : \tau_2, C_2 \quad \tau_1 = z_1^{n_1} \quad \tau_2 = z_2^{n_2} \quad \text{if.label} = \mathbf{b} \quad \mathbf{n} \text{ is fresh}}{\Gamma \vdash \text{if } e_b \text{ then } e_1 \text{ else } e_2 : \langle \tau_1, \tau_2 \rangle^{\mathbf{n}}, C_b \cup C_1 \cup C_2 \cup \{\mathbf{n} = \mathbf{b} \times n_1 + \text{notb} \times n_2\}}$$

The hypothesis “`if.label = b`” extracts the label. The constraint uses the label to relate the sizes of conditional expressions with equivalent booleans. A precise description of how conditional expressions are labeled is given in the companion technical report (see Section 3.2.2 [24]).

## 4.3 Repetition Expressions

The `for` expression is the principle mechanism for producing lists of unspecified length, and is the most involved to handle in our type system.

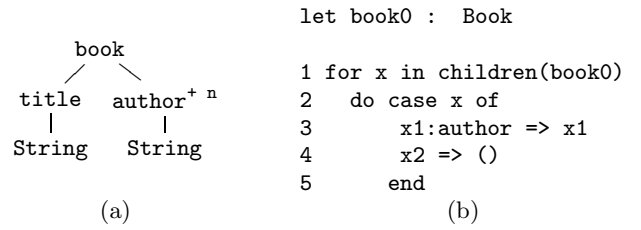


Figure 9: Simple type tree and expression.

Consider the input type and the program fragment shown in Figure 9. By inspection we can see that because of the input type, the expression “`children(book0)`” has type

$$\text{title[ String ] , author[ String ]}^{+ \mathbf{n}}$$

In other words, all data that that expression produces are included in the set defined by that type. Although the “ $\mathbf{n}$ ” does not tell us how many elements each member of the type has, it does say that the number of elements produced is the same as the number of the input elements. The body of the `for` expression operates on each element individually from the value of the “`children`” expression. Each value the `for` expression as a whole produces is included in the type

$$\text{author[ String ]}^{+ \mathbf{n}}$$

Two main ideas allow us to infer such types. The first idea is the use of auxiliary rules for typing the `for` expression. Auxiliary rules allow us to infer a type for the `for` expression which has the same regular structure as the type it iterates over. The second idea is to make explicit the implicit size-related information that we know how to reason about, and equip the `for` rules to use this information. We determine what kind of information we can reason about based on the argument in Section 1.1.

### 4.3.1 Auxiliary rules.

Auxiliary rules for the `for` expression were introduced by Fernandez, Simeon, and Wadler [10], and we review them here. The `for` expression has the form

$$\text{for } \mathbf{x} \text{ in } e_1 \text{ do } e_2 .$$

Suppose that for the program fragment in Figure 9b, we have determined that “`children(book0)`” has type

$$\tau_1 = \begin{array}{cc} \text{title} & , \text{author}^+ \\ | & | \\ \text{String} & \text{String} \end{array}$$

Their type rule (which does not reason about sizes) for the `for` expression looks roughly like:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \uplus \{\text{for } \mathbf{x} : \tau_1\} \vdash e_2 : \tau_2}{\Gamma \vdash \text{for } \mathbf{x} \text{ in } e_1 \text{ do } e_2 : \tau_2}$$

We have already resolved the first hypothesis, and at the top level,  $\tau_1$  is a sequence type. To resolve the second hypothesis, we first use the following auxiliary rule:

$$\frac{\Gamma \uplus \{\text{for } \mathbf{x} : \tau_1\} \vdash e_2 : \tau'_1 \quad \Gamma \uplus \{\text{for } \mathbf{x} : \tau_2\} \vdash e_2 : \tau'_2}{\Gamma \uplus \{\text{for } \mathbf{x} : \tau_1, \tau_2\} \vdash e_2 : \tau'_1, \tau'_2}$$



$$\begin{array}{c}
\frac{\Gamma \vdash e_1 : \tau_1, C_1 \quad \Gamma \uplus \{\text{for } x : \tau_1\} \vdash e_2 : \tau_2, C_2 \quad \text{for.label} = \emptyset}{\Gamma \vdash \text{for } x \text{ in } e_1 \text{ do } e_2 : \tau_2, C_1 \cup C_2} [\text{FOR}] \\
\frac{\Gamma \uplus \{x : u^1\} \vdash e_2 : \tau, C \quad x.\text{label} = \emptyset}{\Gamma \uplus \{\text{for } x : u^1\} \vdash e_2 : \tau, C} [\text{FORU}] \\
\frac{}{\Gamma \uplus \{\text{for } x : ()^0\} \vdash e_2 : ()^0, C} [\text{FORN}] \\
\frac{}{\Gamma \uplus \{\text{for } x : \emptyset\} \vdash e_2 : \emptyset, C} [\text{FORE}] \\
\frac{\Gamma \uplus \{\text{for } x : \tau_1\} \vdash e_2 : \tau'_1, C_1 \quad \Gamma \uplus \{\text{for } x : \tau_2\} \vdash e_2 : \tau'_2, C_2 \quad \tau'_1 = z_1^{m_1} \quad \tau'_2 = z_2^{m_2} \quad \neg \exists \pi. \pi \in C_{1,2}}{\Gamma \uplus \{\text{for } x : (\tau_1, \tau_2)^n\} \vdash e_2 : (\tau'_1, \tau'_2)^{n'}, C_1 \cup C_2 \cup \{n' = m_1 + m_2\}} [\text{FORS}] \\
\frac{\Gamma \uplus \{\text{for } x : \tau_1\} \vdash e_2 : \tau'_1, C_1 \quad \Gamma \uplus \{\text{for } x : \tau_2\} \vdash e_2 : \tau'_2, C_2 \quad \neg \exists \pi. \pi \in C_{1,2}}{\Gamma \uplus \{\text{for } x : (\tau_1 | \tau_2)^n\} \vdash e_2 : (\tau'_1 | \tau'_2)^{n'}, C_1 \cup C_2} [\text{FORC}] \\
\frac{\Gamma \uplus \{\text{for } x : \tau_1\} \vdash e_2 : \tau'_1, C_1 \quad \Gamma \uplus \{\text{for } x : \tau_2\} \vdash e_2 : \tau'_2, C_2 \quad \tau'_1 = z_1^{m_1} \quad \tau'_2 = z_2^{m_2} \quad \neg \exists \pi. \pi \in C_{1,2} \quad \{n = b \times r_1 + \text{notb} \times r_2\}}{\Gamma \uplus \{\text{for } x : \langle \tau_1, \tau_2 \rangle^n\} \vdash e_2 : \langle \tau'_1, \tau'_2 \rangle^{n'}, C_1 \cup C_2 \cup \{n = b \times r_1 + \text{notb} \times r_2\} \cup \{n' = b \times m_1 + \text{notb} \times m_2\}} [\text{FORP}] \\
\frac{\Gamma \uplus \{\text{for } x : \tau\} \vdash e_2 : \tau', C \quad \tau = z^m \quad \tau' = z_1^{m'} \quad \neg \exists \pi. \pi \in C}{\Gamma \uplus \{\text{for } x : \tau^{*n}\} \vdash e_2 : \tau' *^{n'}, C} [\text{FORR}] \\
\frac{\Gamma \vdash e_1 : \tau_1, C_1 \quad \Gamma \uplus \{\text{for } x : \tau_1\} \vdash e_2 : (z^m)^{*n}, C_2 \quad \text{for.label} = \text{start} \quad \exists \pi. \pi \in C_2}{\Gamma \vdash \text{for } x \text{ in } e_1 \text{ do } e_2 : (z^m)^{*n}, C_1 \cup (C_2 \setminus \{\pi\}) \cup \{n = m \times \Gamma_\pi(\pi)\}} [\text{FORII}] \\
\frac{\Gamma \uplus \{x : u^1\} \vdash e_2 : z^m, C \quad x.\text{label} = \pi}{\Gamma \uplus \{\text{for } x : u^1\} \vdash e_2 : (z^m)^{*n'}, C \cup \{\pi\}} [\text{FORIIU}] \\
\frac{\Gamma \uplus \{\text{for } x : \tau_1\} \vdash e_2 : \tau', C \quad \Gamma \uplus \{\text{for } x : \tau_2\} \vdash e_2 : \tau', C \quad \exists \pi. \pi \in C}{\Gamma \uplus \{\text{for } x : \tau\} \vdash e_2 : \tau', C} [\text{FORIII}] \\
\frac{\Gamma \uplus \{\text{for } x : \tau_1\} \vdash e_2 : \tau', C \quad \Gamma \uplus \{\text{for } x : \tau_2\} \vdash e_2 : ()^0, C' \quad \exists \pi. \pi \in C \quad \tau' \neq ()^0}{\Gamma \uplus \{\text{for } x : \tau\} \vdash e_2 : \tau', C} [\text{FORII2}] \\
\frac{\Gamma \uplus \{\text{for } x : \tau_1\} \vdash e_2 : ()^0, C' \quad \Gamma \uplus \{\text{for } x : \tau_2\} \vdash e_2 : \tau', C \quad \exists \pi. \pi \in C \quad \tau' \neq ()^0}{\Gamma \uplus \{\text{for } x : \tau\} \vdash e_2 : \tau', C} [\text{FORIII3}] \\
\frac{\Gamma \uplus \{\text{for } x : \tau_1\} \vdash e_2 : \tau', C \quad \Gamma \uplus \{\text{for } x : \tau_2\} \vdash e_2 : \emptyset, C' \quad \exists \pi. \pi \in C \quad \tau' \neq \emptyset}{\Gamma \uplus \{\text{for } x : \tau\} \vdash e_2 : \tau', C} [\text{FORII4}] \\
\frac{\Gamma \uplus \{\text{for } x : \tau_1\} \vdash e_2 : \emptyset, C' \quad \Gamma \uplus \{\text{for } x : \tau_2\} \vdash e_2 : \tau', C \quad \exists \pi. \pi \in C \quad \tau' \neq \emptyset}{\Gamma \uplus \{\text{for } x : \tau\} \vdash e_2 : \tau', C} [\text{FORII5}] \\
\frac{\Gamma \uplus \{\text{for } x : \tau\} \vdash e_2 : \tau', C \quad \exists \pi. \pi \in C}{\Gamma \uplus \{\text{for } x : (\tau^*)^n\} \vdash e_2 : \tau', C} [\text{FORIIR}]
\end{array}$$

Figure 12: Type rules for the for expression:  $n'$  is fresh in all rules it appears in.

rules is similar to the rules discussed in Section 4.3.1: they decompose the input type structure, find types for the body of the `for` expression as though it were a `let` expression iterating over a single unit type, and re-compose the inferred types according to the input type structure. They are enhanced only to carry size information with them. Figure 12 shows the complete list of type rules for the `for` expression; the auxiliary rules named [FOR#] are this first group of rules. The “ $\neg \exists \pi. \pi \in C$ ” hypothesis will be explained in Section 4.3.4.

The second group of rules have a slightly different purpose and function. They also decompose the structure of the input type, and they find types for the `for` expression as though it were a `let` expression iterating over a single unit type. However, they do not re-compose the types according to the structure of the input type.

The path annotation from Section 4.3.2 tells exactly which of the unit types will cause the `for` expression to produce output. In the program in Figure 11, for example, the `for`

expression starting on line 4 will only produce output when `x` is bound to a `title` element; all other types will result in `()`, the empty sequence. Furthermore, the path annotation can be mapped back to a size name, which could be a constant, in the input type, so it tells exactly how many times output will be produced. In the program in Figure 11, the first half of the program will produce output “`n`” times, because “`n`” is the size name given to “`/catalog/book/title`”.

Instead of re-composing the structure of the input type, the second group of auxiliary rules first infers a type for the unit type that causes the `for` expression to produce output. The auxiliary rules then add a star (`*`) and the size name (e.g., `n`) to the inferred type. In this way, the auxiliary rules flatten the type structure and declare that the type is repeated “`n`” times.

#### 4.3.4 The rules themselves

The second group of auxiliary rules have names of the form [FORII#]. In rules [FORII1]–[FORII5],  $\tau$  can be instantiated to  $(\tau_1, \tau_2)$ ,  $(\tau_1 | \tau_2)$ , or  $\langle \tau_1, \tau_2 \rangle$ . The rule [FORIIU] extracts

the path annotation and adds the path the constraint set. It also adds a star to the inferred type. Hypotheses of the form  $(\neg)\exists\pi. \pi \in \mathbf{C}$  require that there must (not) be a path in the constraint set. The rule **[ForII]** removes the path from the constraint set, uses the “ $\Gamma_\pi$ ” function to map the path back to a size name, and adds a constraint that multiplies the number of elements from one iteration by the number of iterations that produce output.

## 4.4 Type Soundness

We now state some formal properties about our type system.

**THEOREM 1 (SUBJECT REDUCTION).** *Given the predicates provided by the pre- and post-processing, we have: If  $\Gamma \vdash e : \tau, \mathbf{C}$  and  $E \vdash e \Downarrow d$  then  $\Gamma \vdash d : \tau, \mathbf{C}$ .*

The full proof of this theorem is given in the companion technical report [24]. It follows the style of Wright and Felleisen [28]. The “ $d$ ” refers to data, as defined in Figure 6. This theorem relies on a few lemmas. The first is a substitution lemma, which is needed for the expressions that bind variables, namely **for**, **case**, and **let**:

**LEMMA 1.1 (SUBSTITUTION).** *If  $\Gamma \vdash v : \tau, \mathbf{C}$  and  $\Gamma \uplus \{x : \tau\} \vdash e : \tau', \mathbf{C}'$  then  $\Gamma \vdash e [d/x] : \tau', \mathbf{C} \cup \mathbf{C}'$ .*

The **case** expression relies on a *split* function to mimic the case analysis on types (see [24] for the definition of *split*). The second lemma therefore proves that the *split* function conservatively matches the semantics of the **case** expression.

**LEMMA 1.2 (CASE).** *Assume  $\Gamma \vdash d : \tau, \mathbf{C}$  and  $\text{split}^p(\tau) = u'^1|\tau'$ . If  $d \in \text{Dom}(p)$  then  $\Gamma \vdash d : u'^1, \mathbf{C}$ , and if  $d \notin \text{Dom}(p)$  then  $\Gamma \vdash d : \tau', \mathbf{C}$ .*

## 5. SUBTYPING

A thorough treatment of subtyping could fill a separate paper, so we only give an overview here. A type denotes a set of documents. Our notion of subtyping is simply inclusion between the sets denoted by two types. Two restrictions on the types that may be inferred make practical subtyping possible:

- Based on observations in Section 1.1, our type system does not infer size relationships for nested repetition. In other words, in a type tree a repeated element  $\mathbf{b}^*$  will not have a bound size annotation that relates it to other parts of the type tree if one of its ancestors is also repeated (e.g.,  $\mathbf{a}[\dots]^{\mathbf{n}}$ ). This also implies that none of the annotations of  $\mathbf{b}$ 's descendants will be bound except for those bound to constants.
- The size relations we infer are based on the finite structure of the transformation program. The program may have recursive functions, but we infer sizes based on type signatures, which the user provides and our algorithm verifies, and the annotated input type. We do not require the user to provide type signatures for non-recursive functions, because we can inline them an infer types for them.

Size annotations must be matched to establish a subtype relationship. For this, we use  $N$ , defined as:

$$N ::= \emptyset \mid N \cup \{n \leftarrow m\} \mid N \cup \{n \leftarrow n_1 + n_2\}$$

We ensure that subgoals in a subtyping goal match size annotations consistently by means of a “ $\diamond$ ” operator:  $N_1 \diamond N_2$  mean that if  $n$  is matched in both  $N_1$  and  $N_2$ , then  $N_1(n) = N_2(n)$  (i.e., the matchings are consistent). A subtyping relationship,

$$N : \tau_1 <: \tau_2 ,$$

is read  $\tau_1$  is a subtype of  $\tau_2$  with size annotations matched according to  $N$ .

Our algorithm for deciding subtyping relationships first resolves all constraints and gives like-bound size variables in a type like-names. Pair types can then be changed into choice types. We need not worry about having a path  $\pi$  in the constraint set because it will only be necessary to check subtyping at function calls, which can be inlined if there would be a path in the constraint set, and at the end of type inference. The algorithm then divides type trees into two parts: the *upper* part (near the root) which may have relational size annotations, and the *lower* part which must not have relational size annotations. The algorithm discovers the border between the two when it encounters type names or repeated ( $\star$ 'ed) types with unbound size annotations — these mark the top of the lower part.

In the absence of relational size annotations, our type language describes regular expression types for XML, which is the type language for XDuce. XDuce also uses a set-based notion of subtyping, and it has a sound and complete algorithm for deciding subtype relationships [15]. We use XDuce's subtyping for the lower part:

$$\frac{\text{XDuce subtyping}}{\emptyset : \tau_1^* <: \tau} [\text{REP}]$$

In the case of recursive types, we match annotations at the top and use XDuce's subtyping for the types in the absence of annotations.

Because the upper part is free of recursion, we can split choice types and check each of the choices separately:

$$\frac{N_1 : \tau_1 <: z \quad N_2 : \tau_2 <: z}{N_1 \diamond N_2 \quad N_{1,2} \diamond \{n \leftarrow m\}} [N\text{CHOICE}]$$

$$N_1 \cup N_2 \cup \{n \leftarrow m\} : (\tau_1|\tau_2)^{\mathbf{m}} <: z^{\mathbf{n}}$$

The rule for sequences must check subtype relations such as:

$$(a|b), c <: (a, c)|(b, c)$$

Two functions, an *rmc* (remove choices) function and a *tls* (top-level split) function, assist in this. Both functions operate at the top level in the sense that they treat elements as atomic and ignore the type structure of their children. This is required to avoid complications with recursive types, but it means that the rule will reject some true subtype relationships. The *rmc* function takes a type tree  $\tau$  as an argument and returns the set of all type trees derivable by selecting unique combinations of the choices in the top level of  $\tau$ . For example:

$$(a|b), c \Rightarrow \{a, c, b, c\}$$

The  $tls$  function first calls  $rmc$  on its argument. For each resulting type tree  $\tau$ , the set of pairs of type trees that, in sequence, describe  $\tau$  is returned. For example:

$$a, b^{*n}, c \Rightarrow \{(\emptyset, a, b^{*n}, c), [a, b^{*n_1}, b^{*n_2}, c], [a, b^{*n}, c, \emptyset]\}$$

In the example, the size annotated  $b^{*n}$  was “split” in the second pair, so its size annotation was split into  $n_1$  and  $n_2$ . The split is recorded in  $N$  with  $\{n = n_1 + n_2\}$ .

$$\frac{\begin{array}{l} \forall \tau'_1 \in rmc(\tau_1) \quad \forall \tau'_2 \in rmc(\tau_2) \\ \exists N_p : \tau_a, \tau_b \in tls(z) \\ N_a : \tau'_1 <: \tau_a \quad N_b : \tau'_2 <: \tau_b \\ \forall i, j. N_i \diamond N_j \quad N_i \diamond \{n \leftarrow m\} \end{array}}{\bigcup N_i \cup \{n \leftarrow m\} : (\tau_1, \tau_2)^m <: z^n} [NSEQ]$$

Our subtyping algorithm include nine other rules not shown here. This subtyping algorithm is sound and it terminates.

## 5.1 Syntactic/Semantic Balance

This type system, as presented here, has certain syntactic aspects. Most notably, alternating sequences of **for** and optional **case** expressions are detected syntactically and annotated with labels that are used in type inference. Although it would be nice to work more in the semantic domain like Xi and Pfenning do with dependant types [30], a few significant hurdles prevent it. First, type systems for ML-like functional languages need only worry about the program, which is entirely available at the time of type inference. In the context of XML type checking, we must worry about both the program and the input type, which gives only limited information about execution-relevant data. Second, we must deal with the tree structure of XML types, which is a richer structure than that of flat arrays, which are common in ML-like languages. This was explained in detail in Section 1.1. Third, we require an algorithm for subtyping. The functional programming paradigm does not rely on subtyping (although the object-oriented paradigm does), so type systems can be designed for ML-like languages without subtyping in mind.

## 6. RELATED WORK

### 6.1 Automata-based Techniques

In [21], six ways of representing XML types (including XML Schema) are classified by expressiveness. The types we work with here are regular expression tree types with size annotations, which are at least as expressive as the six surveyed.

One significant automata-based work on XML type checking uses a generalization of traditional top-down regular tree transducers called  $k$ -pebble tree transducers to demonstrate the decidability of type checking for the broad range of queries that can be expressed by these automata [20]. This technique was applied to a subset of XSLT for backward type inference [26]. However, it is unclear how to support size information in these formalisms. Adding sizes naively can produce non-context-free languages and make type checking undecidable.

### 6.2 Type System-based Techniques

Instead of using automata-based approaches, many XML transformation languages use type systems to accomplish XML type checking. The aspects of XQuery’s type system

relevant to this work were explained in Sect. 1.1. Other languages, such as XDuce, have similar type systems [8, 14]. Fernandez *et al.*’s more precise type system was discussed in Sect 4.3 [10]. However, our type system appears to be the first one to support size inference.

Other work on XML type checking is aimed at integrating XML into general-purpose programming languages. One integrates XML into Java [18], and the work relies on Jwig [4], an extension of Java. Xobe [17] is also an extension of Java with a similar goal, but it differs in that XML trees in Xobe can only be constructed bottom-up, as opposed to allowing named gaps that can be filled in any order. Castor [13] and JAXB [19] use Java to generate an object model of XML documents from XML Schema in order to gain a higher level of abstraction.

## 6.3 Size Analysis

We view size analysis as seeking to make claims about the sizes of data structure and other closely-related aspects of programs. The study of size analysis started with the inference of linear constraints for imperative languages [5]. This abstract interpretation-based approach inferred linear relationships among variables automatically. This topic has significance to logic programming in the sense that inferring bounds on argument sizes can ensure termination [23].

Type-based size analyses relate more closely to our work. One such analysis uses dependent types [29, 30]. They use parameterized types to infer lengths of lists. The parameters can be constrained with linear equalities and inequalities to determine size relationships. Unlike our type system, theirs requires user annotations. Hughes, Pareto, and Sabry type check recursive data structures with size information in the context of a lazy functional language [16]. Chin and Khoo build on this approach by inferring sizes for recursive functions in the context of strict functional languages [3]. They define the size of a function as both a relation between input and output parameters, and invariants of input parameters across recursive calls. They infer sizes in terms of array lengths, tree heights, and integer values. All of these previous approaches only infer flat sizes; even when sizes for trees are inferred, they are in infer sizes for the richer tree structure and take into account the levels of the subtrees.

## 7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a type system for XML transformations to infer size relationships within the output type. Our approach also allows size annotations to be added to the input type that would then be propagated through to the inferred output type. In addition to helping programmers confirm properties of XML transformation programs, size relations may provide efficiency improvements. Knowledge of concrete sizes can benefit query optimization and database storage, so we expect size relations to yield similar benefits [12, 22].

Finally, our type system does not add significant complexity to either type inference or document validation. Only simple and usually small equations, which can be solved efficiently using simple symbolic algebra, will be inferred in the constraints. XML Schema is designed so that validation can be implemented by a top-down parser with limited look

ahead [21]. Adding size annotations requires only the addition of counters to keep track the number of elements, which does not increase the algorithmic complexity of performing document validation.

There are a few possible directions for future work. In this work we dealt with recursive functions by requiring the user to provide a type signature and type checking to verify the correctness of that signature. We may be able to infer a type without being provided a type signature by using XDuce's type system to infer a type signature that does not include size annotations and then adding size annotations on a second pass. We may be able to avoid doing two passes by using ideas from Chin and Khoo [3]. Finally, it would be interesting to implement our inference procedure to gain some practical experiences.

## 8. REFERENCES

- [1] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL*, pages 163–173, 1994.
- [2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *ICFP-03*, volume 38, 9, pages 51–63, August 25–29 2003.
- [3] W. Chin and S. Khoo. Calculating sized types. In *PEPM*, pages 62–72, 1999.
- [4] A. S. Christensen and A. Møller. JWIG user manual, June 2002. URL: <http://www.brics.dk/JWIG/manual/>.
- [5] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.
- [6] J. Clark (eds.). XML transformations (XSLT) version 1.0. *W3C*, Nov. 1999. URL: <http://www.w3.org/TR/xslt>.
- [7] Roberto Chinnici (eds.). Web services description language (WSDL) version 2.0. *W3C*, March 2004. URL: <http://www.w3.org/TR/wsd120/>.
- [8] Scott Boag (eds.). XQuery: the W3C query language for XML – W3C working draft. *W3C*, November 2003. URL: <http://www.w3.org/TR/xquery>.
- [9] Tim Bray (eds.). Extensible markup language (XML) version 1.0. *W3C*, February 2004. URL: <http://www.w3.org/TR/PR-xml-971208.ps>.
- [10] M. Fernandez, J. Siméon, and P. Wadler. An algebra for XML Query. In *FST TCS*, pages 11–45, December 2000.
- [11] M. Fitzgerald. Relaxer tutorial, 2003. URL: <http://www.relaxer.org/doc/tutorial/tutorial.html>.
- [12] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX: making XML count. In *SIGMOD*, pages 181–191, New York, NY 10036, USA, June 2002.
- [13] Exolab Group. Castor, 2002. URL: <http://castor.exolab.org>.
- [14] H. Hosoya and B. C. Pierce. “XDuce: A Typed XML Processing Language”. In *WebDB*, Dallas, TX, 2000.
- [15] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for xml. In *ICFP*, pages 11–22, 2000.
- [16] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL*, 1996.
- [17] M. Kempa and V. Linnemann. Type checking in XOBEL. In *BTW '03*, pages 227–246, February 2003.
- [18] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of xml transformations in java. URL: <http://citeseer.nj.nec.com/593778.html>.
- [19] Sun Microsystems. JAXB, 2002. URL: <http://java.sun.com/xml/jaxb>.
- [20] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *PODS*, pages 11–22, 2000.
- [21] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.
- [22] Carlo Sartiani. A framework for estimating XML query cardinality. In *WebDB*, San Diego, California, 2003.
- [23] D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, pages 199–260, 1994.
- [24] Z. Su and G. Wassermann. A type-based dimensional analysis for XQuery. Technical Report CSE-2004-8, University of California, Davis, April 2004. URL: <http://wwwcsif.cs.ucdavis.edu/~wassermg/research/SizeTechRpt.ps>.
- [25] D. Suciu. The XML typechecking problem. *SIGMOD Record*, March 2002.
- [26] A. Tozawa. Towards static type checking for XSLT. In *Document Eng*, pages 18–27, 2001.
- [27] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *ICFP*, pages 148–159, 1999.
- [28] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.
- [29] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, pages 249–257, 1998.
- [30] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.