Type-based Inference of Size Relationships for XML Transformations

Zhendong Su and Gary Wassermann

Department of Computer Science, University of California, Davis, CA 95616
{su,wassermg}@cs.ucdavis.edu

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) [20] 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 τ_s

and τ_t respectively, and transformation program P, is it true that $\forall x \in \tau_s . P(x) \in \tau_t$ [24]? One common approach to answer this question is based on type inference: an output type τ' 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 no size relations are present [19]. 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 [26], 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.

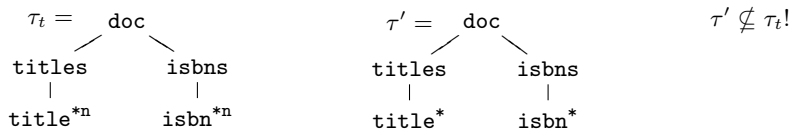

Fig. 1. A target type with size relations. Conservatively inferred types using existing techniques cause correct programs to fail to type check.

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 τ_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 τ' 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.

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 τ_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. 10a, 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 subex-

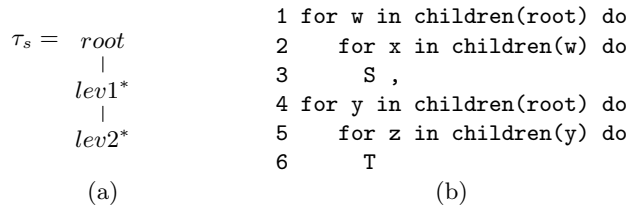

Fig. 2. A source type with nested repetitions.

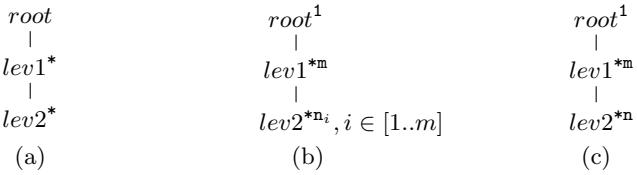

Fig. 3. Two ways to annotate a source type τ_s.

pressions 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 τ_s as in Fig. 3b, where `1`, `m`, and $\mathtt{n}_i$ 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` e_1 `do` e_2). 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, τ_1, and e_2 is typed once with x having that binding. Then type constructors are added to the inferred type based on their occurrences in τ_1.

In inferring a type for the program in Fig. 2b, if τ_s is annotated as shown in Fig. 3b, the type of expression `children(root)` on line `1` is τ_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 τ_1, as done in XQuery's type system. This yields τ_u. The type of `children(w)` on line `2` is then τ_{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 τ_{ub}, and so the body of the inner `for` expression on line `3` is typed as τ_{2b}. We then go back up and compose the type τ_{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 τ' (for clarity, `r` has been simplified out of the constraints).

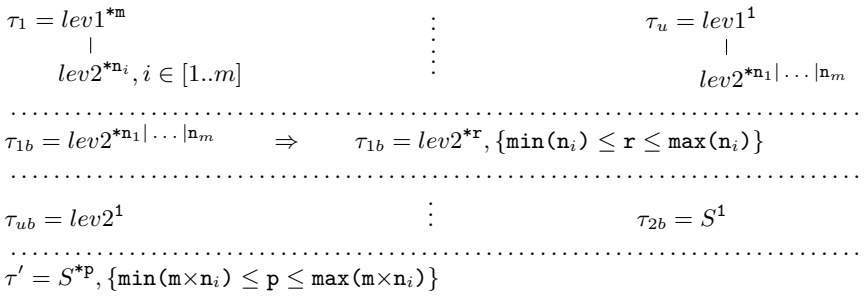

Fig. 4. Some inferred types in our first attempt to infer all size relationships precisely.

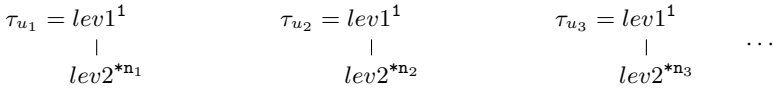

Fig. 5. Some inferred types in our next attempt to infer all size relationships precisely.

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 τ_1 and combine them based on the type structure of τ_1. Given the annotation for τ_s in Fig. 3b, the type of `children(root)` is again τ_1, as shown in Fig. 4. Because the number of children ($\mathtt{n}_i$) may be different for each of the `m` $lev1$'s, the first unique unit type here is τ_{u_1}, as shown in Fig. 5. The second is τ_{u_2}, the third is τ_{u_3}, and so on. Trying to infer a type for the `for` expression by inferring a type for e_2 based on τ_{u_1}, τ_{u_2}, τ_{u_3}, ..., is cumbersome and requires complicated symbolic reasoning about summations such as $\sum_{i=1}^{m} \mathtt{n}_i$.

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.*, $lev2^{*\mathtt{n}_i}$) 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 $/root/lev1/lev2$, 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 τ_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 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

Fig. 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, functions, and modules. We do not expect much difficulty in extending our technique to cover these language constructs.

tag	a		
variable	x		
constant	n	::=	c_{int} | c_{str} | c_{bool}
operator	op	::=	`+` | `-` | `and` | `or` | `=` | `<`
expression	e	::=	n | x | a`[`e`]` | e `,` e | `()` | e op e
			| `let` x `=` e `do` e | `for` x `in` e `do` e | `children(`e`)`
			| `case` e `of` x`:`p `=>` e `|` x `=>` e `end` | `if` e `then` e `else` e
pattern	p	::=	a | `~` | s
data	d	::=	n | a`[`d`]` | d `,` d | `()`

Fig. 6. XML transformation language.

tag	a		
size type	r	::=	`c` | `n`
scalar type	s	::=	`String` | `Boolean` | `Integer`
unit type	u	::=	a`[`τ`]` | `~[`τ`]` | s
type	z	::=	τ `,` τ | τ `|` τ | `<`τ `,` τ`>` | τ`*` | `()` | $\emptyset$
annotated type	τ	::=	u^1 | z^r

Fig. 7. Type language.

Note that the `case` expression matches a value against a p, defined by the "pattern" derivation (which parallels the "unit type" derivation in Fig. 7). Also, as shown by the "data" derivation, we denote XML elements as `tag[...]` rather than `<tag>...</tag>` to simplify notation. The dynamic semantics for this language is standard, but a complete presentation is given in the companion technical report [23].

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

Among the types z, "τ `,` τ" is the type of two values in sequence. The union type is "τ `|` τ"; a value whose type is either of the choices matches it. We introduce the pair type, "`<`τ `,` τ`>`," 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 $\texttt{<}\tau_1\texttt{,}\tau_2\texttt{>} \neq \texttt{<}\tau_2\texttt{,}\tau_1\texttt{>}$, 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.

We also have a constraint language to capture size relations. Fig. 8 shows our constraint language. There are two kinds of constraints. The first kind consists

$$
\begin{array}{lll}
\text{expression} & e & ::= \mathtt{c} \mid \mathtt{n} \mid (\Gamma_\pi(\pi)) \mid (e) \mid e + e \mid e \times e \mid e \,/\, e \\
\text{path} & \pi & ::= \pi/a \mid \varepsilon \\
\text{constraint} & \mathtt{C} & ::= \mathtt{C} \cup \{\mathtt{n} = e\} \mid \mathtt{C} \cup \{\pi\} \mid \emptyset
\end{array}
$$

Fig. 8. Constraint language.

of equality constraints between a size variable and an arithmetic expression. Unique to these expressions is the mapping using the path environment Γ_π. A *valid path* is a path where the first tag matches some root-level tag(s) in τ_s by being identical to it (them) or is `~`, in which case it matches all root-level tags, and each successive tag matches in the same way some element(s) in τ_s which are children of elements matched by the tag's predecessor. For example, $/root/lev1/lev2$ and $/root/\mathtt{\sim}/lev2$ are both valid paths for τ_s as given in Fig. 3. However, if $lev1$ in τ_s were replaced with `~`, then $/root/lev1/lev2$ would not be a valid path even though in a valid input tree such a sequence may occur. The path is invalid because our annotations of τ_s tell us nothing about how many $lev2$'s in a $/root/lev1/lev2$ path there are. If the path π being mapped by Γ_π is a valid path in the source type τ_s, $\Gamma_\pi(\pi)$ returns a sum of constants and variables from the annotated τ_s representing the number of unit values this path includes. If π is not a valid path in τ_s, $\Gamma_\pi(\pi)$ returns a fresh variable, which will not appear in any other constraints. This is equivalent to returning "unknown."

Constraints may also include a path, π, which is not explicitly related to anything else in the constraints or types. Such a path remains in the constraint set while the nested `for` and `case` expressions that match the semantics of the path as an expression are being typed. In the last phase of typing that sequence of expressions, the path will be extracted from the constraint set and mapped with the Γ_π environment. The way that the path and Γ_π get connected in our type inference algorithm is explained through an example in Sect. 4.3.

4 Type Rules

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

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

where $\{\mathtt{for}\ x : \tau\}$ is used in typing the `for` expression. A type environment, Γ, maps variables and "`for`" variables to types according to the following rules:

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

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 [23]. In the type rules that we discuss next, z, u, and τ are as in Fig. 7: z is a type without a size annotation, u is a unit type without an annotation, and τ is a type with an annotation.

In Sect. 4.1, we explain the type rule for sequence expressions, in which two subexpressions are put in sequence. In Sect. 4.2, we explain the type rule plus pre- and post-processing for conditionals. In Sect. 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:

$$\frac{\tau_1 = z_1^{\mathtt{m}_1} \qquad \tau_2 = z_2^{\mathtt{m}_2} \\ \Gamma \vdash e_1 : \tau_1\mathtt{,\ C}_1 \qquad \Gamma \vdash e_2 : \tau_2\mathtt{,\ C}_2 \qquad \mathtt{n} \text{ is fresh}}{\Gamma \vdash e_1\mathtt{,}e_2 : (\tau_1,\tau_2)^{\mathtt{n}}\mathtt{,\ C}_1 \cup \mathtt{C}_2 \cup \{\mathtt{n} = \mathtt{m}_1 + \mathtt{m}_2\}}$$

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 `n`, the size of the sequence expression, equals $\mathtt{m}_1 + \mathtt{m}_2$, 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 problem by means of a *pair type*, introduced in Sect. 3, plus some post-processing to relate pair types.

Our type rule for conditional expressions is as follows:

$$\frac{\Gamma \vdash e_b : \mathtt{Boolean}^{\mathtt{1}}\mathtt{,\ C}_b \qquad \Gamma \vdash e_1 : \tau_1\mathtt{,\ C}_1 \qquad \Gamma \vdash e_2 : \tau_2\mathtt{,\ C}_2 \\ \mathtt{n} \text{ is fresh} \qquad \mathtt{if}.\mathit{label} = \mathtt{b} \qquad \tau_1\mathtt{=}\ {z_1}^{\mathtt{n}_1} \qquad \tau_2\mathtt{=}\ {z_2}^{\mathtt{n}_2}}{\begin{array}{c}\Gamma \vdash \mathtt{if}\ e_b\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \mathtt{<}\tau_1,\tau_2\mathtt{>}^{\mathtt{n}}\mathtt{,} \\ \mathtt{C}_b \cup \mathtt{C}_1 \cup \mathtt{C}_2 \cup \{\mathtt{n} = \mathtt{b} \times \mathtt{n}_1 + \mathtt{notb} \times \mathtt{n}_2\}\end{array}}$$

A pre-processing phase labels each conditional expression with a unique label. The hypothesis "$\mathtt{if}.\mathit{label} = \mathtt{b}$" extracts that label for use in the constraints. The variables `b` and `notb` in the conclusion cannot simply be fresh variables because in the post-processing it is necessary to relate the variables to the conditional expression that produces them.

```
τs = Book = book[ author[ Str ]+,          let book0 : Book
                  title[ Str ],
                  subtitle[ Str ]?,       1 for x in children(book0) do
                  isbn[ Int ] ]           2         case x of
                                          3           x1:title => x1
τ' =()+, title[ Str ], ()?, ()            4           x2 => ()
  = title[ Str ]                          5         end
```

Fig. 9. An example for the `for` expression.

This rule adds a constraint which equates `n`, the size of the pair type, to an expression which includes `b` and `notb`. Intuitively these can be thought of as representing the number of times e_b is `true` and the number of times it is `false` across all executions of the conditional expression in one run of the program. During type inference, these are considered unbound variables.

The variables `b` and `notb` become useful when constraints are added during the post-processing phase that declares them to be equal to another pair of variables, `b'` and `notb'`, respectively. The post-processing adds these constraints as it puts the boolean expressions (e_b), which these variables name, into equivalence classes. The companion technical report ([23], Sect. 3.2.2) explains the post-processing more completely and precisely, but intuitively, if two boolean expressions are executed the same number of times in one run of the program and are structurally equivalent (and thus are `true`/`false` the same number of times), they are equivalent.

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.

Fernandez *et al.* describe a technique for typing the `for` expression, in which a type for e_2 is found once for each unique unit type in τ_1 (e_1's type), and those types are composed according to the type structure of τ_1 [10]. We build on that technique, so we review its main idea here. Consider the source type and program in Fig. 9. The type `Book` has a root with tag `book`, which has some children. Those children are one or more `author`s, a `title`, an optional `subtitle`, and an `isbn` number. Each of those has a scalar child. The variable `book0` is an instance of `Book`. The program iterates over the children of `book0`, and, in the case when the current child is a `title`, it outputs the child; otherwise it outputs an empty sequence.

The expression `children(book0)` has four unique unit types, so we type the `case` expression (e_2) with each of the unit types bound to the variable `x`:

- when `x` has type `author[Str]`, e_2 has type `()`;
- when `x` has type `title[Str]`, e_2 has type `title[Str]`;
- when `x` has type `subtitle[Str]`, e_2 has type `()`;
- when `x` has type `isbn[Int]`, e_2 has type `()`.

```
τs = Catalog = catalog[ Book* ]
let cat0 : Catalog
 1 titles[ for w in children(cat0) do
 2   case w of
 3     w1:book =>
 4       for x in children(w1) do
 5         case x of
 6           x1:title => x1
 7           x2 => ()
 8         end
 9     w2 => ()
10   end ],
11 isbns[ for y in children(cat0) do
12   case y of
13     y1:book =>
14       for z in children(y1) do
15         case z of
16           z1:isbn => z1
17           z2 => ()
18         end
19     y2 => ()
20   end ]
```

(a)

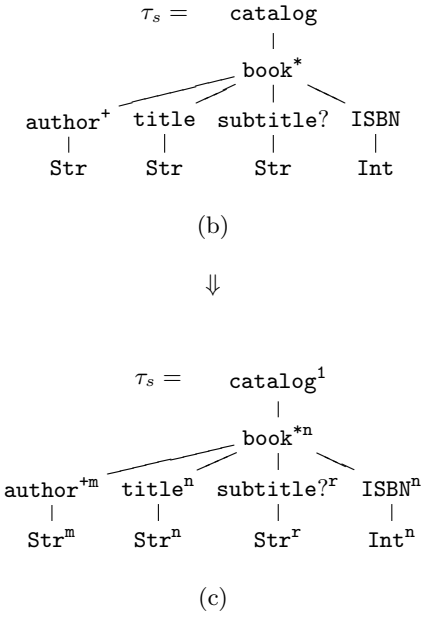

Fig. 10. A program that makes a list of all titles followed by all ISBNs from a catalog.

When we replace each of these unit types in τ_1 with the corresponding inferred types for e_2, we get τ', as shown in Fig. 9. After doing some simplification to get rid of the empty sequence types, we get $\tau' =$ `title[Str]`. This inference is accomplished by means of some auxiliary rules, similar to the auxiliary rules in our type system (see Fig. 11).

Fig. 10a shows a program fragment that we will use as an example for explaining our treatment of the `for` expression. The type `Book` is as defined in Fig. 9. The sequences of nested `for` and `case` expressions have the same semantics as the paths in the example program in Sect. 1, so with the exception of the missing outermost `doc` element, Fig. 10a has the same semantics as the example program. Because each `book` has exactly one `title` and one `isbn`, we expect `titles` and `isbns` to have the same number of children.

Source Type Annotation If two expressions must produce lists of the same length, unless the output is constant and static, it is because the expressions are related in the same way to the input type. More specifically, they both iterate over input-tree-paths of equal size, and they both output the same number of unit values per iteration. To capture this intuition, we annotate the source type in a pre-processing phase. The annotations added denote the total number of elements in a concrete input document represented by a path in the input type tree. Two annotations will be algebraically related if the number of elements

they denote must be algebraically related. For example, in Fig. 10c, `book` has exactly one child `title`, so `title` has the same annotation as `book`. The number of `author`'s each `book` has is not specified, so `author` is annotated with `m`, a fresh size variable, to denote unknown size.

These annotations are interpreted differently from the annotations appearing on the output type and optionally appearing on the input type. These annotations denote the total number of elements on a path; the annotations on input or output types denote the number of elements under a single parent.

Program Annotation Next, a pre-processing phase on the program is executed. The purpose of this pre-processing is to find and annotate iteration expressions that can be related directly back to τ_s. Our formulation of the pre-processing finds structures of nested `for` and `case` expressions which have equivalent semantics to path expressions. The companion technical report includes the algorithm for this pre-processing (see [23] Sect. 3.2.3, Fig. 16). In Fig. 10a, such structures appear twice: first in lines `1` through `10` and then again in lines `11` through `20`. The first half corresponds to the path `/books/book/title` and the second half corresponds to `/books/book/isbn`. For each path found, the `for` that corresponds to the beginning of a path is labeled *start*, and the iteration variable at the end is labeled with the path π. For the program in Fig. 10a, the `for`'s on lines `1` and `11` are labeled *start*, `x` on line `4` is labeled `/books/book/title`, and `z` on line `14` is labeled `/books/book/isbn`.

Typing Repetition Expressions Fig. 11 gives the full list of rules and auxiliary rules for typing the `for` expression. The rules [For] and [ForΠ] type `for` expressions, and the rest are auxiliary rules. When one of the hypotheses includes a type assignment of the form $\{\texttt{for}\ x{:}\tau\}$, the auxiliary rules resolve the type of the expression in that hypothesis. The hypothesis "$\texttt{for}.label = \ldots$" in the hypotheses of [For] and [ForΠ] refers to the result of program annotation from last paragraph: if the `for` expression is the first expression in a sequence that represents a path, it is labeled *start*; otherwise it has no label. Equivalently with "$x.label = \ldots$," if the `for` expression is the last `for` expression in a sequence that represents a path, the iteration variable is labeled with the path it represents; otherwise it has no label. In the rules [ForΠ1] through [ForΠ5], τ can be instantiated to (τ_1,τ_2), $(\tau_1|\tau_2)$, or $<\tau_1,\tau_2>$. The hypothesis $\neg\exists\pi.\ \pi \in \texttt{C}_{1,2}$ in [ForS] through [ForP] means that there exists no path constraints in the constraint sets of the preceding hypotheses. Conversely, in [ForΠ1] through [ForΠ5], $\exists\pi.\ \pi \in \texttt{C}$ means there is a path constraint in the constraint set `C`.

We explain the intuition behind these rules using the example program in Fig. 10a. The companion technical report illustrates more completely the use of the auxiliary rules for typing the `for` expression through a full type derivation tree for the `for` expression on lines `1`–`10` of Fig. 10a (see [23], Fig. 17).

The `x` on line `5` of Fig. 10a iterates over the values represented by the type of the children of `book`. The part of the input type tree that represents these values is shown as the third level on Fig. 10c. The type rule invoked on line `5` is [For]. In its second hypothesis, the type environment includes $\{\texttt{for}\ x\ :\ \tau_1\}$.

$$\frac{\Gamma \vdash e_1 : \tau_1,\ \mathtt{C}_1 \quad \Gamma \uplus \{\mathtt{for}\ x : \tau_1\} \vdash e_2 : \tau_2,\ \mathtt{C}_2 \quad \mathtt{for}.label = \emptyset}{\Gamma \vdash \mathtt{for}\ x\ \mathtt{in}\ e_1\ \mathtt{do}\ e_2 : \tau_2,\ \mathtt{C}_1 \cup \mathtt{C}_2}[\textsc{For}] \qquad \frac{\Gamma \vdash e_1 : \tau_1,\ \mathtt{C}_1 \quad \Gamma \uplus \{\mathtt{for}\ x : \tau_1\} \vdash e_2 : (z^{\mathtt{m}})^{*\mathtt{n}},\ \mathtt{C}_2 \quad \mathtt{for}.label = start \quad \exists \pi.\ \pi \in \mathtt{C}_2}{\Gamma \vdash \mathtt{for}\ x\ \mathtt{in}\ e_1\ \mathtt{do}\ e_2 : (z^{\mathtt{m}})^{*\mathtt{n}},\ \mathtt{C}_1 \cup (\mathtt{C}_2 \setminus \{\pi\}) \cup \{\mathtt{n} = \mathtt{m} \times \Gamma_\pi(\pi)\}}[\textsc{For}\Pi]$$

$$\frac{\Gamma \uplus \{x : u^1\} \vdash e_2 : \tau,\ \mathtt{C} \quad x.label = \emptyset}{\Gamma \uplus \{\mathtt{for}\ x : u^1\} \vdash e_2 : \tau,\ \mathtt{C}}[\textsc{For}\mathrm{U}]$$

$$\frac{\Gamma \uplus \{x : u^1\} \vdash e_2 : z^{\mathtt{m}},\ \mathtt{C} \quad x.label = \pi}{\Gamma \uplus \{\mathtt{for}\ x : u^1\} \vdash e_2 : (z^{\mathtt{m}})\ *^{\mathtt{n}'},\ \mathtt{C} \cup \{\pi\}}[\textsc{For}\Pi\mathrm{U}]$$

$$\frac{}{\Gamma \uplus \{\mathtt{for}\ x : ()^0\} \vdash e_2 : ()^0}[\textsc{For}\mathrm{N}] \qquad \frac{}{\Gamma \uplus \{\mathtt{for}\ x : \emptyset\} \vdash e_2 : \emptyset}[\textsc{For}\mathrm{E}] \qquad \frac{\Gamma \uplus \{\mathtt{for}\ x : \tau_1\} \vdash e_2 : \tau',\ \mathtt{C} \quad \Gamma \uplus \{\mathtt{for}\ x : \tau_2\} \vdash e_2 : \tau',\ \mathtt{C} \quad \exists \pi.\ \pi \in \mathtt{C}}{\Gamma \uplus \{\mathtt{for}\ x : \tau\} \vdash e_2 : \tau',\ \mathtt{C}}[\textsc{For}\Pi 1]$$

$$\frac{\Gamma \uplus \{\mathtt{for}\ x : \tau_1\} \vdash e_2 : \tau_1',\ \mathtt{C}_1 \quad \Gamma \uplus \{\mathtt{for}\ x : \tau_2\} \vdash e_2 : \tau_2',\ \mathtt{C}_2 \quad \tau_1' = z_1^{\mathtt{m}_1} \quad \tau_2' = z_2^{\mathtt{m}_2} \quad \neg\exists \pi.\ \pi \in \mathtt{C}_{1,2}}{\Gamma \uplus \{\mathtt{for}\ x : (\tau_1, \tau_2)^{\mathtt{n}}\} \vdash e_2 : (\tau_1', \tau_2')^{\mathtt{n}'},\ \mathtt{C}_1 \cup \mathtt{C}_2 \cup \{\mathtt{n}' = \mathtt{m}_1 + \mathtt{m}_2\}}[\textsc{For}\mathrm{S}] \qquad \frac{\Gamma \uplus \{\mathtt{for}\ x : \tau_1\} \vdash e_2 : \tau',\ \mathtt{C} \quad \Gamma \uplus \{\mathtt{for}\ x : \tau_2\} \vdash e_2 : ()^0,\ \mathtt{C}' \quad \exists \pi.\ \pi \in \mathtt{C} \quad \tau' \neq ()^0}{\Gamma \uplus \{\mathtt{for}\ x : \tau\} \vdash e_2 : \tau',\ \mathtt{C}}[\textsc{For}\Pi 2]$$

$$\frac{\Gamma \uplus \{\mathtt{for}\ x : \tau_1\} \vdash e_2 : \tau_1',\ \mathtt{C}_1 \quad \Gamma \uplus \{\mathtt{for}\ x : \tau_2\} \vdash e_2 : \tau_2',\ \mathtt{C}_2 \quad \neg\exists \pi.\ \pi \in \mathtt{C}_{1,2}}{\Gamma \uplus \{\mathtt{for}\ x : (\tau_1 | \tau_2)^{\mathtt{n}}\} \vdash e_2 : (\tau_1' | \tau_2')^{\mathtt{n}'},\ \mathrm{C}_1 \cup \mathrm{C}_2}[\textsc{For}\mathrm{C}] \qquad \frac{\Gamma \uplus \{\mathtt{for}\ x : \tau_1\} \vdash e_2 : ()^0,\ \mathtt{C}' \quad \Gamma \uplus \{\mathtt{for}\ x : \tau_2\} \vdash e_2 : \tau',\ \mathtt{C} \quad \exists \pi.\ \pi \in \mathtt{C} \quad \tau' \neq ()^0}{\Gamma \uplus \{\mathtt{for}\ x : \tau\} \vdash e_2 : \tau',\ \mathtt{C}}[\textsc{For}\Pi 3]$$

$$\frac{\Gamma \uplus \{\mathtt{for}\ x : \tau_1\} \vdash e_2 : \tau_1',\ \mathtt{C}_1 \quad \Gamma \uplus \{\mathtt{for}\ x : \tau_2\} \vdash e_2 : \tau_2',\ \mathtt{C}_2 \quad \tau_1' = z_1^{\mathtt{m}_1} \quad \tau_2' = z_2^{\mathtt{m}_2} \quad \neg\exists \pi.\ \pi \in \mathtt{C}_{1,2} \quad \{\mathtt{n} = \mathtt{b} \times \mathtt{r}_1 + \mathtt{notb} \times \mathtt{r}_2\}}{\Gamma \uplus \{\mathtt{for}\ x : \langle \tau_1, \tau_2 \rangle^{\mathtt{n}}\} \vdash e_2 : \langle \tau_1', \tau_2' \rangle^{\mathtt{n}'},\ \mathtt{C}_1 \cup \mathtt{C}_2 \cup \{\mathtt{n} = \mathtt{b} \times \mathtt{r}_1 + \mathtt{notb} \times \mathtt{r}_2\} \cup \{\mathtt{n}' = \mathtt{b} \times \mathtt{m}_1 + \mathtt{notb} \times \mathtt{m}_2\}}[\textsc{For}\mathrm{P}]$$

$$\frac{\Gamma \uplus \{\mathtt{for}\ x : \tau_1\} \vdash e_2 : \tau',\ \mathtt{C} \quad \Gamma \uplus \{\mathtt{for}\ x : \tau_2\} \vdash e_2 : \emptyset \quad \exists \pi.\ \pi \in \mathtt{C} \quad \tau' \neq \emptyset}{\Gamma \uplus \{\mathtt{for}\ x : \tau\} \vdash e_2 : \tau',\ \mathtt{C}}[\textsc{For}\Pi 4] \qquad \frac{\Gamma \uplus \{\mathtt{for}\ x : \tau_1\} \vdash e_2 : \emptyset \quad \Gamma \uplus \{\mathtt{for}\ x : \tau_2\} \vdash e_2 : \tau',\ \mathtt{C} \quad \exists \pi.\ \pi \in \mathtt{C} \quad \tau' \neq \emptyset}{\Gamma \uplus \{\mathtt{for}\ x : \tau\} \vdash e_2 : \tau',\ \mathtt{C}}[\textsc{For}\Pi 5]$$

$$\frac{\Gamma \uplus \{\mathtt{for}\ x : \tau\} \vdash e_2 : \tau',\ \mathtt{C} \quad \tau = z^{\mathtt{m}} \quad \tau' = z_1^{\mathtt{m}'} \quad \neg\exists \pi.\ \pi \in \mathtt{C}}{\Gamma \uplus \{\mathtt{for}\ x : \tau *^{\mathtt{n}}\} \vdash e_2 : \tau'\ *^{\mathtt{n}'},\ \{\mathtt{n}' = (\mathtt{n}/\mathtt{m}) \times \mathtt{m}'\} \cup \mathtt{C}}[\textsc{For}\mathrm{R}] \qquad \frac{\Gamma \uplus \{\mathtt{for}\ x : \tau\} \vdash e_2 : \tau', \mathtt{C} \quad \exists \pi.\ \pi \in \mathtt{C}}{\Gamma \uplus \{\mathtt{for}\ x : (\tau*)^{\mathtt{n}}\} \vdash e_2 : \tau',\ \mathtt{C}}[\textsc{For}\Pi\mathrm{R}]$$

Fig. 11. Type rules for the `for` expression: $\mathtt{n}'$ is fresh in all rules it appears in.

This must be resolved with the auxiliary rules. Because `x` is at the end of a path, it was labeled in the preprocessing phase with the path `/catalog/book/title`.

The variable `x` was labeled with a path so that the number of values it iterates over could be related directly back to the input type. On each iteration, the body of the `for` expression (lines 5–8) produces some output of type τ. For all types τ, τ is a subtype of $\tau^{*\mathtt{n}}$ (where `n` is unbound). As the final step of type inference for the `for` expression, `n` can be bound to the product of two factors. The first is the variable assigned to the path that labels `x` in the input type. The second is the outermost size variable on τ. Note, then, that even though $\tau^{*\mathtt{n}} \equiv (\tau^{*\mathtt{m}})^{*\mathtt{n}}$ when `n` and `m` are unbound, if the `n`'s on both sides are bound to the product of some `c` and of the outermost size variable underneath them, the two types will not be equivalent. Because `m` is unbound, we have no information about the number of elements represented by the second type. Therefore, the auxiliary rules are designed to propagate back to the [For] rule a type of the form $\tau^{*\mathtt{n}}$, where `n` is unbound and τ is the type of the body of the `for` expression.

When `x` has type $\mathtt{title[Str^1]^1}$, the body of the `for` expression has type $\mathtt{title[Str^1]^1}$. The rule `[For`Π`U]` infers the type $(\mathtt{title[Str^1]^1})^{*\mathtt{r}}$, where `r` is a fresh variable, for the reason explained above. When `x` has any other type, the body of the `for` expression is $()^0$. The starred type gets propagated back to the [For] rule on line 5, and then it gets propagated further to the [ForΠ] rule on line 1. The [ForΠ] then binds the size variable `r` to the product described in the previous paragraph, namely `r = 1 * n = n`. By similar reasoning for the expression on lines 11–20, we can infer that the expression produces `n` ISBNs, the information that we desire.

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* $\Gamma \vdash e : \tau, \mathtt{C}$ *and* $E \vdash e \Downarrow v$ *then* $\Gamma \vdash v : \tau, \mathtt{C}$.

The full proof of this theorem is given in the companion technical report [23]. It follows the style of Wright and Felleisen [27]. 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 (Substitution). *If* $\Gamma \vdash v : \tau, \mathtt{C}$ *and* $\Gamma \uplus \{x : \tau\} \vdash e : \tau', \mathtt{C'}$ *then* $\Gamma \vdash e\ [v/x] : \tau', \mathtt{C} \cup \mathtt{C'}$.

The `case` expression relies on a *split* function to mimic the case analysis on types (see [23] for the definition of *split*). The second lemma therefore proves that the *split* function conservatively matches the semantics of the `case` expression.

Lemma 2 (Case). *Assume* $\Gamma \vdash v : \tau, \mathtt{C}$ *and* $split^p(\tau) = u'^{\mathtt{1}} | \tau'$.
If $v \in Dom(p)$ *then* $\Gamma \vdash v : u'^{\mathtt{1}}, \mathtt{C}$, *and if* $v \notin Dom(p)$ *then* $\Gamma \vdash v : \tau', \mathtt{C}$.

5 Related Work

5.1 Automata-based Techniques

In [20], 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 [19]. This technique was applied to a subset of XSLT for backward type inference [25]. 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.

5.2 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 [17], and the work relies on JWIG [4], an extension of Java. XOBE [16] 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 [18] use Java to generate an object model of XML documents from XML Schema in order to gain a higher level of abstraction.

5.3 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 [22].

Type-based size analyses relate more closely to our work. One such analysis uses dependent types [28, 29]. 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 [15]. 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 one-dimensional heights. We infer sizes for the richer tree structure and take into account the levels of the subtrees.

6 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, 21].

Finally, our type system does not add significant complexity to either type inference or document validation. Only simple, and usually small, linear arithmetic equations, which can be solved efficiently, 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 [20]. 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 have not considered functions or recursive types. However, we expect that our technique could be extended to include these language constructs. In the case of functions, for better precision, we may need to pursue a context-sensitive analysis. Precise handling of recursive functions seems challenging, but ideas based on context-free language reachability may be applied. The challenge with recursive types lies in proper source type annotation. We believe this could be accomplished by annotating the states of the term automata representation of recursive types. In this paper, we have shown how to infer precise types with size information. However, in order to perform XML type checking, we also need to investigate techniques for subtyping. In general, it is undecidable to check inclusion of tree automata enhanced with size information. However, it is interesting to study conservative, but practical notions of subtyping. Finally, it would be interesting to implement our inference procedure to gain some practical experiences.

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, New York, 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/wsdl20/.
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. J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL*, 1996.
16. M. Kempa and V. Linnemann. Type checking in XOBE. In *BTW '03*, pages 227–246, February 2003.
17. C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of xml transformations in java. URL: http://citeseer.nj.nec.com/593778.html.
18. Sun Microsystems. JAXB, 2002. URL: http://java.sun.com/xml/jaxb.
19. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *PODS*, pages 11–22, 2000.
20. M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.
21. Carlo Sartiani. A framework for estimating XML query cardinality. In *WebDB*, San Diego, California, 2003.
22. D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, pages 199–260, 1994.
23. 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.
24. D. Suciu. The XML typechecking problem. *SIGMOD Record*, March 2002.
25. A. Tozawa. Towards static type checking for XSLT. In *Document Eng*, pages 18–27, 2001.
26. M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *ICFP*, pages 148–159, 1999.
27. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.
28. H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, pages 249–257, 1998.
29. H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.