Type-based Inference of Size Relationships for XML Transformations

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. Type- and automata-theoretic techniques that exploit XML's tree structure have been proposed to address this problem. 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, which occurs when data is repeated. 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. Programs that use or produce parallel or repeated data will benefit from this analysis. 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.

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 run-time 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 $\forall x \in \tau_s . P(x) \in \tau_t$ [25]? 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 actual 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 [20]. It is unclear how

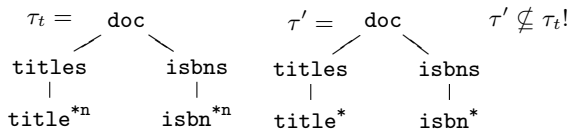


Figure 1: A target type with size relations. Conservatively inferred types using existing techniques cause correct programs to fail to type check.

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. Section 2 introduces our language.

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.

Figure 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 graphical view of the tree structure of XML types. A vertical or diagonal line means that the type at the lower end of the line is a child of the type at the upper end. In the type τ' in Figure 1, `titles, isbns` is a sequence type; the sequence constructor is implicit because `titles` and `isbns` are children of the same parent and are next to each other. Using existing type inference methods, the type τ' would be inferred. Because $\tau' \not\subseteq \tau_t$, this correct program fails to type check. Repeated data arises in many settings, and each time it does, this shortcoming of existing techniques limits the amount of automated checking available to programmers.

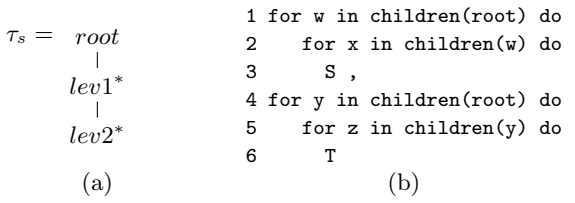


Figure 2: A source type with nested repetitions.

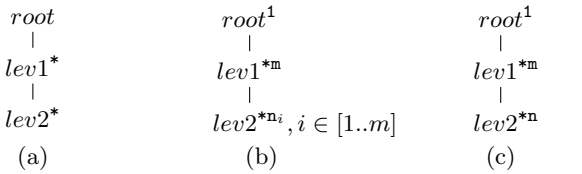


Figure 3: Two ways to annotate a source type τ_s.

1.1 Difficulties with Size Relation Inference

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 Figure 2. Lines `1`–`3` and `4`–`6` of the program in Figure 2 have the same semantics as $/root/\tilde{}/\tilde{}$ (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 Figure 10, lines `1`–`10` are equivalent to `/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 size relations between the input type and the type of each expression in order to relate the sizes of the expressions' types to each other. Suppose that in hoping to discover the size relationships precisely, we annotate τ_s as in Figure 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

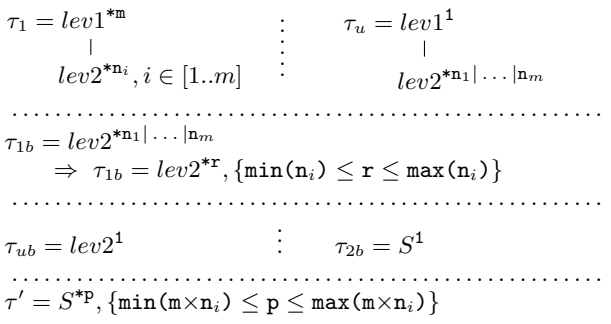


Figure 4: Some inferred types in our first attempt to infer all size relationships precisely.

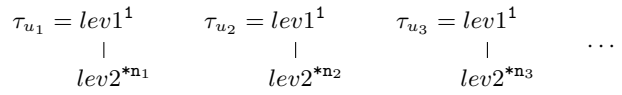


Figure 5: Some inferred types in our next attempt to infer all size relationships precisely.

occurrences in τ_1.

In inferring a type for the program in Figure 2b, if τ_s is annotated as shown in Figure 3b, the type of expression `children(root)` on line 1 is τ_1, as shown in Figure 4. Figure 4 also shows the rest of the types we refer to in this paragraph. Suppose we find the type of `w`, τ_u, by taking the union of the unit types in τ_1, as done in XQuery's type system. The type of `children(w)` on line 2 is then τ_{1b}. However, it is not clear what the size annotation on τ_{1b} means. To add clarity, we rewrite the size annotation as shown in Figure 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).

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 Figure 3b, the type of `children(root)` is again τ_1, as shown in Figure 4. Because the number of children ($\mathtt{n}_i$) may be different for each of the `m` *lev*1's, the first unique unit type here is τ_{u_1}, as shown in Figure 5. The second is τ_{u_2}, the third is τ_{u_3}, etc. 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 Figure 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 Figure 4), or in addressing the distinctions directly, the type system faces increased complexity (as shown in Figure 5). In this paper, we present an approach to overcome these problems.

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 Figure 3c. The annotation `n` on *lev*2 in Figure 3c denotes the total number of *lev*2 elements in the input document that have as parent a *lev*1 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 $/root/lev1/lev2$,[1] 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 Figure 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 Section 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 preprocessing 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

[1]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	$::=$ `+` $\mid$ `-` $\mid$ `and` $\mid$ `or` $\mid$ `=` $\mid$ `<`
expression	e	$::=$ n $\mid x \mid a$`[`e`]` $\mid e$ `,` e $\mid$ `()` $\mid e\ op\ e \mid$ `let` x `=` e `do` e $\mid$ `for` x `in` e `do` $e \mid$ `children(`e`)` $\mid$ `case` e `of` x`:`p `=>` e `|` x `=>` e `end` $\mid$ `if` e `then` e `else` $e \mid f($`(`e`;...;`e`)`
pattern	p	$::= a \mid$ `~` $\mid s$
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	$::=$ `String` $\mid$ `Boolean` $\mid$ `Integer`
unit type	u	$::= a$`[`τ`]` $\mid$ `~[`τ`]` $\mid s$
type	z	$::= x \mid \tau$ `,` $\tau \mid \tau$`*` $\mid$ `()` $\mid$ `<`τ `,` τ`>` $\mid \tau$ `|` $\tau \mid \emptyset$
annotated type	τ	$::= u^1 \mid z^r$

Figure 7: Type 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 `<tag>...</tag>` to simplify notation. The dynamic semantics for this language is standard, but the companion technical report gives a complete presentation [24].

3. OUR TYPE LANGUAGE

Figure 7 gives our type language. The "size type" shows that either a constant or a variable can be used as a size annotation on a type. The size annotation denotes the number of unit values that may be matched to the annotated type. The grammar allows nonsense types to be written (*e.g.*, $()^2$), but our type system only infers meaningful types and programs only type check if all types are meaningful. 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 `<`$\tau_1$`,`$\tau_2$`>` $\neq$ `<`τ_2`,`τ_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.

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

Figure 8: Constraint language.

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 Γ_π maps paths to size variables/constants. The second kind of constraint is a path, π, 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 Γ_π, 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 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:

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

where $\{$`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:

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

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 the companion technical report [24]. In the type rules that we discuss next, z, u, and τ are as in Figure 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 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. We also discuss the typing of recursive functions in Section 4.3.2.

4.1 Sequence Expressions

The type rule for sequence expressions is as follows:

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

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 by means of a *pair type*, introduced in Section 3, plus related constraints.

The main ideas 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. We can use unification to conservatively determine which variables have the same binding. A straightforward 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:

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

The hypothesis "$\mathtt{if}.\mathit{label} = \mathtt{p}$" extracts the label, and uses it as a fresh size variable. The rule also uses the label to create a fresh size variable, `notp`, which represents the unknown number of times the boolean expression evaluates to `false`. The constraint uses the label to relate the sizes of conditional expressions with equivalent booleans.

4.3 Repetition Expressions

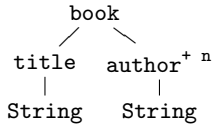
The `for` expression is the principle mechanism for producing lists of unspecified length, and is the most involved to handle in our type system.

Consider the input type and the program fragment shown in Figure 9. By inspection of the input type, we conclude that the expression "`children(book0)`" has type:

$$\tau \ = \ \mathtt{title[\ String\ ]\ ,\ author[\ String\ ]}^{+\ \mathtt{n}}$$

In other words, all data that `children(book0)` produces are included in the set defined by τ. Although the "`n`" in τ does not tell us how many elements each member of τ 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 execution of the `for` expression produces data included in the type:

$$\tau' \ = \ \mathtt{author[\ String\ ]}^{+\ \mathtt{n}}$$



(a)

```
let book0 :  Book

1 for x in children(book0)
2   do case x of
3        x1:author => x1
4        x2 => ()
5      end
```

(b)

Figure 9: Simple type tree and expression.

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 that the input type and the program contain implicit size-related information, which we make explicit. We determine which information to make explicit based on the argument in Section 1.1. We then equip the `for` rules to use this information.

4.3.1 Auxiliary Rules

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

$$\mathtt{for\ x\ in}\ e_1\ \mathtt{do}\ e_2\ .$$

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

$$\tau_1 = \ \mathtt{title}\ \mathtt{,}\ \mathtt{author}^{+} \\ \mathtt{String}\quad\ \mathtt{String}$$

We show the sequence constructor here to make it explicit that this is a sequence type. Fernandez *et al.*'s type rule (which does not reason about sizes) for the `for` expression looks roughly like:

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

We have already found τ_1, and at the top level, τ_1 is a sequence type. To find τ_2 for the second hypothesis, we first use the following auxiliary rule:

$$\frac{\Gamma \uplus \{\mathtt{for\ x:}\tau_1\} \vdash e_2 : \tau_1' \\ \Gamma \uplus \{\mathtt{for\ x:}\tau_2\} \vdash e_2 : \tau_2'}{\Gamma \uplus \{\mathtt{for\ x:}\tau_1\mathtt{,}\tau_2\} \vdash e_2 : \tau_1'\mathtt{,}\tau_2'}$$

This rules types the original program fragment by

- finding the `for` expression's type as though e_1 had type `title[..]`,
- finding the `for` expression's type as though e_1 had type $\mathtt{author[..]}^{+}$,
- and putting those two types in sequence.

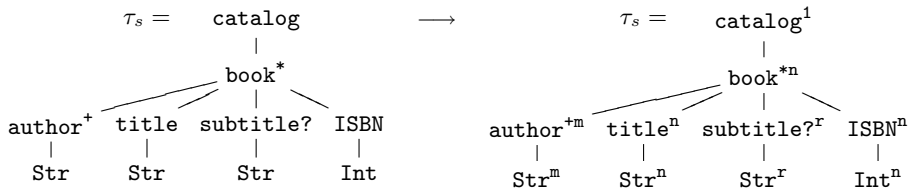
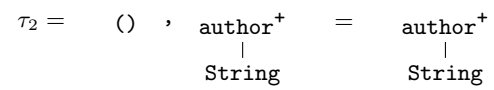


Figure 11: The input type tree annotated.

```
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 ]
```

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

Other auxiliary rules handle other type constructors, such as repetitions ("+"). If e_1's type is a unit type (*e.g.*, `title[..]`), then the `for` expression can be typed like a `let` expression. The purpose of these auxiliary rules is to decompose the input type, infer types for the `for` expression over unit types, and compose the inferred types according to the structure of the input type. The resulting type of the `for` expression in Figure 9 is:



4.3.2 Making Size-related Information Explicit

In Sections 1.1 and 1.2, we argued that a natural, type-based analysis can reason about size information at, but not below, the top level of repetition. We define *top-level repetition* for a given type to be types which are repeated (`*`'d) and which have no repeated ancestors. We get the information that we need for reasoning about sizes by answering:

1. Which `for` expressions produce top-level repetition in the output type?

2. Which parts of the input type do those `for` expressions iterate over?

3. How many elements are in those parts of the input type?

Aside from conditional expressions, which Section 4.2 addressed, the `for` expression is the only expression which can both produce output of unspecified length and have other expressions nested in it. To answer the first question conservatively, we use the syntactic structure of the program. A `for` expression may not produce top-level repetition if it is nested in an expression e, where e is not a `for` or `case` expression and e is nested in the body of a `for` expression. Otherwise, the `for` expression will produce top-level repetition. We answer the second question similarly. Consider the nested `for` and `case` expressions on lines 1–10 of the program in Figure 10, for which τ_s in Figure 11 is the input type. Collectively, these expressions iterate over the `title` element, or more specifically the path `/catalog/book/title`, in τ_s. We infer this path by checking for certain syntactic structures (*e.g.*, e_1 is a `children` expression in all `for` expression, and each expression uses a variable bound by the expression above it), and piecing together the patterns from the `case` expressions. If the relevant syntactic structures do not appear, we do not infer a path. We make paths explicit for each group of `for` and `case` expressions that produce top-level repetition by labeling the first expression with *start* and the last expression with the path it iterates over.

To answer the third question, we first note that the number of elements represented by a path in the input type is the same each time an expression iterates over that path. We therefore give names to the numbers of elements along each path of the input type. Figure 11 shows an example of this naming. Starting from the root, sizes are given names. If the number of child elements per parent element is constant (*e.g.*, 1 `title` per `book`), the child's size is proportional to the parent's. These names get used as size variables when an expression outputs the corresponding portion of the input tree. The precise details on how size names are put on the input type tree and how the transformation program is annotated can be found in the companion technical report (see Section 3.2.3 [24]).

Recursion, both in the input type and the program, is another source of statically unknown output. In the case of recursive types, we give a name to the total number elements represented by each tag at the first "back-edge." In a recursive type, there is at least one pair of type elements or type constructors where each can be reached from the other. We say that a back-edge goes to the type element/constructor that can be reached from the root of the type tree first. A type element/constructor with an incoming first back-edge has no anscestors with incoming back-edges. Figure 12 shows a recursive type first textually [10] then graphically. The annotation at the choice type-constructor means, for

```
type Part = Basic | Composite
type Basic = basic[ cost[ Integer ] ]
type Composite = composite[
                        assembly_cost[ Integer ],
                        subparts[ Part+ ] ]
```

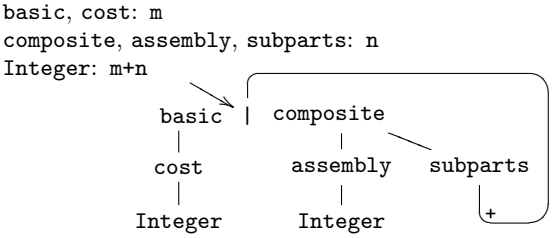


Figure 12: Annotation of a recursive type.

```
type Part2 = part[ total_cost[ Integer ],
                   subparts[ Part2* ] ]
convert(p : Part<basic: m, composite: n>)
          : Part2<part: m+n>
```

Figure 13: Annotation of a recursive function.

example, that the number of `basic` tags represented by this recursive type is `m`. Recursive functions typically operate on recursive types. We handle recursive functions by requiring the user to annotate each recursive function with a type signature and type checking the functions based on those annotations. Figure 13 shows a second recursive type, `Part2`. For each `part` the `total_cost` includes any costs for that part plus the sum of all costs of subparts. The function `convert` transforms a `Part` into a `Part2`. Due to space limitations, we omit the function body. Since both basic parts and composite parts are parts, the total number of `part`'s should equal the sum of the numbers of `basic`'s and `composite`'s. The user need not annotate the function with the size of every type element—in Figure 13 only 3 have annotations.

4.3.3 Auxiliary Rules for Size Types

We write auxiliary rules to make use of the size-related information discussed in Section 4.3.2. Basically, two groups of auxiliary rules are needed: one group for those `for` expressions that are not known to produce top-level repetitions, and one group for those that are. The first group of auxiliary rules is similar to the rules discussed in Section 4.3.1—they decompose the input type, 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 structure of the input type. The auxiliary rules are enhanced only to carry size information with them. Figure 14 shows the complete list of type rules for the `for` expression; the auxiliary rules named [For#] are the first group of rules. The "$\neg\exists\pi.\ \pi \in \mathtt{C}$" 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 a potentially non-empty list (*i.e.*, produce output). In the program in Figure 10, 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 the `for` expression that iterates over that path will produce output. In the program in Figure 10, 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 [ForΠ#]. In rules [ForΠ1]–[ForΠ5], τ can be instantiated to `(`τ_1`,`τ_2`)`, `(`τ_1`|`τ_2`)`, or `<`τ_1`,`τ_2`>`. The rule [ForΠU] extracts the path annotation and adds the path the constraint set. It also adds a star to the inferred type. Hypotheses of the form $(\neg)\exists\pi.\ \pi \in \mathtt{C}$ require that there must (not) be a path in the constraint set. The rule `[For`Π`]` removes the path from the constraint set, uses the "Γ_π" 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 a type preservation theorem for our system.

Theorem 1 (Subject Reduction). *If* $\Gamma \vdash e : \tau$`, C` *and* $E \vdash e \Downarrow d$ *then* $\Gamma \vdash d : \tau$`, 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.

5. SUBTYPING

XML type checking first requires inferring a type for the transformation program, which Section 4 addressed. This section addresses the second part of XML type checking—checking whether the inferred type is a subtype of the target type. A type denotes a set of documents. Our notion of subtyping is simply inclusion between the sets denoted by two types. A restriction on the types that may be inferred makes deciding subtyping possible: Based on observations in Section 1.1, our type system does not infer size relationships for nested repetitions. In other words, in a type tree a repeated element $\mathtt{b}^*$ will not have a size annotation that relates it to other parts of the type tree if one of its ancestors is also repeated (*e.g.*, $\mathtt{a[...]}^{*\mathtt{n}}$). This also implies that none of the annotations of `b`'s descendants will relate their sizes to other parts of the type tree.

Size annotations must be matched to establish a subtype relationship. We use N to record how size annotations are matched, and we define it as:

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

$$\frac{\Gamma \vdash e_1 : \tau_1,\ \mathtt{C}_1 \quad \Gamma \uplus \{\mathtt{for}\ x : \tau_1\} \vdash e_2 : \tau_2,\ \mathtt{C}_2 \quad \mathtt{for}.label = \emptyset}{\Gamma \vdash \mathtt{for}\ x\ \mathtt{in}\ e_1\ \mathtt{do}\ e_2 : \tau_2,\ \mathtt{C}_1 \cup \mathtt{C}_2}\ [\textsc{For}]$$

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

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

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

$$\frac{}{\Gamma \uplus \{\mathtt{for}\ x : ()^0\} \vdash e_2 : ()^0,\ \mathtt{C}}\ [\textsc{For}\mathrm{N}]$$

$$\frac{}{\Gamma \uplus \{\mathtt{for}\ x : \emptyset\} \vdash e_2 : \emptyset,\ \mathtt{C}}\ [\textsc{For}\mathrm{E}]$$

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

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

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

$$\frac{\Gamma \uplus \{\mathtt{for}\ x : \tau_1\} \vdash e_2 : \tau',\ \mathtt{C} \quad \tau = z^{\mathtt{m}} \quad \tau' = z_1^{\mathtt{m}'} \quad \neg\exists \pi.\ \pi \in \mathtt{C}}{\Gamma \uplus \{\mathtt{for}\ x : \tau *^{\mathtt{n}}\} \vdash e_2 : \tau' *^{\mathtt{n}'},\ \mathtt{C}}\ [\textsc{For}\mathrm{R}]$$

$$\frac{\Gamma \uplus \{\mathtt{for}\ x : \tau_1\} \vdash e_2 : \tau',\ \mathtt{C} \quad \Gamma \uplus \{\mathtt{for}\ x : \tau_2\} \vdash e_2 : \tau',\ \mathtt{C} \quad \exists \pi.\ \pi \in \mathtt{C}}{\Gamma \uplus \{\mathtt{for}\ x : \tau\} \vdash e_2 : \tau',\ \mathtt{C}}\ [\textsc{For}\Pi 1]$$

$$\frac{\Gamma \uplus \{\mathtt{for}\ x : \tau_1\} \vdash e_2 : \tau',\ \mathtt{C} \quad \Gamma \uplus \{\mathtt{for}\ x : \tau_2\} \vdash e_2 : ()^0,\ \mathtt{C}' \quad \exists \pi.\ \pi \in \mathtt{C} \quad \tau' \neq ()^0}{\Gamma \uplus \{\mathtt{for}\ x : \tau\} \vdash e_2 : \tau',\ \mathtt{C}}\ [\textsc{For}\Pi 2]$$

$$\frac{\Gamma \uplus \{\mathtt{for}\ x : \tau_1\} \vdash e_2 : ()^0,\ \mathtt{C}' \quad \Gamma \uplus \{\mathtt{for}\ x : \tau_2\} \vdash e_2 : \tau',\ \mathtt{C} \quad \exists \pi.\ \pi \in \mathtt{C} \quad \tau' \neq ()^0}{\Gamma \uplus \{\mathtt{for}\ x : \tau\} \vdash e_2 : \tau',\ \mathtt{C}}\ [\textsc{For}\Pi 3]$$

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

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

$$\frac{\Gamma \uplus \{\mathtt{for}\ x : \tau\} \vdash e_2 : \tau', \mathtt{C} \quad \exists \pi.\ \pi \in \mathtt{C}}{\Gamma \uplus \{\mathtt{for}\ x : (\tau*)^{\mathtt{n}}\} \vdash e_2 : \tau',\ \mathtt{C}}\ [\textsc{For}\Pi\mathrm{R}]$$

Figure 14: Type rules for the `for` expression: $\mathtt{n}'$ is fresh in all rules it appears in.

The matching $\{n \leftarrow m\}$ means that m, a size annotation in the subtype, is assigned to n, an annotation in the supertype. The other matching records that the sum of two annotations equals a third annotation, and it is used by an auxiliary function (introduced later in this section). We ensure that subgoals in a subtyping goal match size annotations consistently by means of a "$\diamond$" operator: $N_1 \diamond N_2$ means 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 list as a subscript (*e.g.*, $N_{1,2} \diamond N_3$) abbreviates several consistency checks (*e.g.*, $N_1 \diamond N_3$, $N_2 \diamond N_3$). A subtyping relationship, $N : \tau_1 <: \tau_2$, is read τ_1 is a subtype of τ_2 with size annotations matched according to N.

Our type inference algorithm generates constraints on size annotations, and those constraints must be resolved before subtyping. We only need to do a subtype check at function calls and at the end of type inference. The type inference algorithm guarantees that the constraint set will not include a path (π) at either of these times. After constraint resolution, size annotations which are bound to equivalent expressions are renamed so that they are identical. Pair types can then be changed into choice types. 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 (`*`'ed) types with unbound size annotations—these mark the top of the lower part. For example, in the un-annotated type tree in Figure 11, the upper part ends at `book*`, and the lower part is everything below `book*`.

In the absence of relational size annotations, our type language describes regular expression types for XML, the same type language as XDuce uses. XDuce also uses a set-based notion of subtyping, and it has a sound and complete algorithm for deciding subtyping [15]. To check subtyping algorithm for the lower part of the type tree (*e.g.*, $\emptyset : \tau_1^* <: \tau$), we use XDuce's subtyping algorithm. 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:

$$\frac{\begin{array}{cc} N_1 : \tau_1 <: z & N_2 : \tau_2 <: z \\ N_1 \diamond N_2 & N_{1,2} \diamond \{n \leftarrow m\} \end{array}}{N_1 \cup N_2 \cup \{n \leftarrow m\} : (\tau_1 | \tau_2)^{\mathtt{m}} <: z^{\mathtt{n}}} [\text{NCHOICE}]$$

If an expected annotation is missing, an annotation can be added. The annotation will be a constant if the size is constant (*e.g.*, 1). If the annotation is for a sequence type, and the sizes of the sequence's members equal the sizes of another sequence, the annotation will be identical to the annotation on the other sequence. Otherwise, the annotation will be fresh, which is equivalent to "unknown." The rule for sequences must check subtype relations such as:

$$(a|b)\mathtt{,}c <: (a\mathtt{,}c)|(b\mathtt{,}c)$$

We use two functions: *rmc* (remove choices) and *tls* (top-level split). Both functions operate at the top level of types by treating elements as atomic and ignoring their children. This allows us to keep the upper and lower parts of the type tree separate, but it means that the rule will reject some true subtype relationships. The *rmc* function takes a type tree τ as an argument and returns the set of all type trees derived by selecting one or the other of the types from the choice types in τ. For example:

$$rmc((a|b)\mathtt{,}c) \quad = \quad \{a\mathtt{,}c\ ,\ b\mathtt{,}c\}$$

The *tls* function first calls *rmc* on its argument. For each resulting type tree τ, *tls* returns the set of pairs of type trees that, in sequence, describe τ. For example:

$$tls(a\mathtt{,}b^{*\mathtt{n}}\mathtt{,}c) = \\ \{[()\ ,\ \ a\mathtt{,}b^{*\mathtt{n}}\mathtt{,}c],\ \ [a\mathtt{,}b^{*\mathtt{n}_1}\ ,\ \ b^{*\mathtt{n}_2}\mathtt{,}c],\ \ [a\mathtt{,}b^{*\mathtt{n}}\mathtt{,}c\ ,\ \ ()]\}$$

In the example, the size annotated $b^{*\mathtt{n}}$ was "split" in the second pair, so its size annotation was split into $\mathtt{n}_1$ and $\mathtt{n}_2$. The split is recorded in N with $\{n = n_1 + n_2\}$.

$$\frac{\begin{array}{c} \forall \tau_1' \in rmc(\tau_1) \quad \forall \tau_2' \in rmc(\tau_2) \\ \exists N_p : \tau_a, \tau_b \in tls(z) \\ N_a : \tau_1' <: \tau_a \qquad N_b : \tau_2' <: \tau_b \\ N_{a,b,p} \diamond N_{a,b,p} \qquad N_{a,b,p} \diamond \{n \leftarrow m\} \end{array}}{\bigcup N_{a,b,p} \cup \{n \leftarrow m\} : (\tau_1\mathtt{,}\tau_2)^{\mathtt{m}} <: z^{\mathtt{n}}} [\text{NSEQ}]$$

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

6. RELATED WORK

6.1 Automata-based Techniques

Murata *et al.* classify six ways of representing XML types (including XML Schema) in terms of expressiveness [21]. 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 Section 1.1. Other languages, such as XDuce, have similar type systems [8, 14]. Fernandez *et al.*'s more precise type system was discussed in Section 4.3 [10]. However, our type system to the best of our knowledge is the first one to support size inference.

Other work on XML type checking aims 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, it is in terms of their one-dimensional height. We 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. Our type inference algorithm only infers in the constraints simple and usually small equations, which can be solved efficiently using

simple symbolic algebra. 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.

Acknowledgments

We would like to thank Mary Fernandez for helpful answers to our questions on XQuery and on "An Algebra for XQuery" [10]. We also thank the anonymous reviewers of an earlier version of our paper for their helpful comments.

8. REFERENCES

[1] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL*, 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, 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, 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 XOBE. 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, CA, 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*, 1998.

[30] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.