The XML Typechecking Problem*

Dan Suciu
University of Washington

1 Introduction

When an XML document conforms to a given type (e.g. a DTD or an XML Schema type) it is called a *valid* document. Checking if a given XML document is valid is called the validation problem, and is typically performed by a parser (hence, *validating parser*), more precisely it is performed right after parsing, by the same program module. In practice however XML documents are often generated dynamically, by some program: checking whether all XML documents generated by the program are valid w.r.t. a given type is called the typechecking problem. While a validation analyzes an XML document, a type checker analyzes a program, and the problem's difficulty is a function of the language in which that program is expressed. The XML typechecking problem has been investigated recently in [MSV00, HP00, HP01, AMN+01a, AMN+01b] and the XQuery Working Group adopted some of these techniques for typechecking XQuery [FFM+01]). All these techniques, however, have limitations which need to be understood and further explored and investigated. In this paper we define the XML typechecking problem, and present current approaches to typechecking, discussing their limitations.

2 Background

2.1 XML Data

For the purpose of this paper we view XML documents as ordered trees, with nodes labeled with tag or attribute names, or atomic values. This is a subset of the XQuery data model [FFM+01], which is sufficient to illustrate typechecking. For the purpose of typechecking we ignore the data values and only consider their atomic types instead. Thus, we fix an alphabet Σ of tag names, attribute names, and atomic type names, and denote $\mathcal{T}_\Sigma$ the set of ordered trees where each node is labeled with an element from Σ. For a simple illustration, the document in Fig. 1 (a) is represented as the tree in (b). Here:

$$\Sigma = \{ \texttt{ELEMENT catalog}, \texttt{ELEMENT product}, \texttt{ATTRIBUTE name}, \texttt{ELEMENT mfr-price}, \texttt{ELEMENT sale-price}, \texttt{ELEMENT color}, \texttt{INTEGER}, \texttt{STRING} \}$$

2.2 XML Types

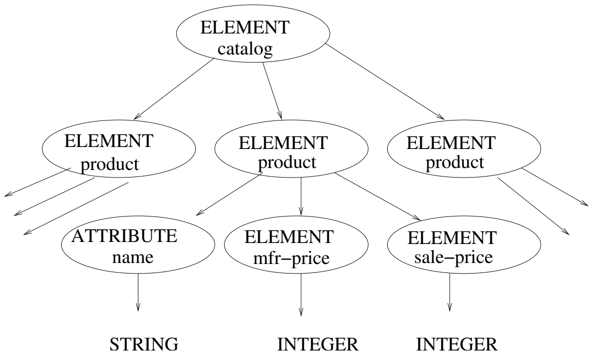
A type is a subset of $\mathcal{T}_\Sigma$ that is a regular tree language [RS97, Tho90]. We use XQuery's syntax to specify types.

Example 2.1 Consider the type definition:

```
TYPE Catalog  = ELEMENT catalog(Products)
TYPE Products = (ELEMENT product(Product))*
TYPE Product  = (ATTRIBUTE name(STRING)?,
                (ELEMENT mfr-price(INTEGER) | ELEMENT sale-price(INTEGER))*,
                (ELEMENT color(STRING))*)
```

*Database Principles Column. Column editor: Leonid Libkin, Department of Computer Science, University of Toronto, Toronto, Ontario M5S 3H5, Canada. E-mail: libkin@cs.toronto.edu.

```
<catalog>
  <product name="Widget">
    <mfr-price> 55 </mfr-price>
    <color> Red </color>
  </product>
  <product name="Gizmo">
    <mfr-price> 99 </mfr-price>
    <sale-price> 79 </sale-price>
  </product>
  <product>
    <color> Green </color>
    <color> Blue </color>
  </product>
</catalog>
```



(a) (b)

Figure 1: An XML Example and its Tree Representation

Here, `Catalog, Products, Product` are type identifiers. `Catalog` defines the set of trees with the following structure: the root is labeled `catalog`, all its children are labeled `product`, and their children form a sequence as defined by the `Product` type.

Formally, a type is defined by a set of type identifiers, T, and associates to each identifier a regular expression over $\Sigma \times T$.

This definition of types hides much of the complexities of XML Schema, and it is actually more powerful than XML Schema types. To see this, consider the set of pairs $(\sigma, t) \in \Sigma \times T$ that occur in the regular expression for some type identifier. XML Schema requires that σ be a key in this collection. In other words if both (σ, t) and (σ, t') occur in this regular expression, then $t = t'$ (they are the same identifier). A DTD imposes an even stricter restriction, that σ be a key in the entire collection of pairs in all regular expressions.

Regular tree languages have been extensively studied for *ranked* trees, and are defined in terms of tree automata [RS97, Tho90]. XML trees, however, are unranked: there are several ways to carry over the definition of regular sets to unranked trees, by extending tree automata to unranked trees [BKMW98], as *specialized DTDs* [PV00], by mapping unranked trees into ranked binary trees [MSV00], or by defining types as in XDuce [HP00], XQuery [FFM$^+$01] (and as defined here). All these definitions are equivalent, see [Suc01] for a more detailed discussion.

For typechecking we need an important property of regular tree languages: given two types τ_1, τ_2, one can check whether $\tau_1 \subseteq \tau_2$. In general, this problem is EXPTIME complete [Sei90], but it is in PTIME when τ_2 corresponds to a deterministic tree automaton. A practical algorithm specifically designed for XML typechecking is described in [HP01].

2.3 The XML Typechecking Problem

Recall that in the *validation problem* we are given a tree $t \in \mathcal{T}_\Sigma$ and a type τ and have to decide whether $t \in \tau$. The tree in Fig. 1 is valid w.r.t. the type `Catalog` in Example 2.1.

In the *typechecking problem* we are given a program P, defining a function $P : \mathcal{D} \to \mathcal{T}_\Sigma$, and a type $\tau \subseteq \mathcal{T}_\Sigma$, and need to decide whether $\forall x \in \mathcal{D}$, $P(x) \in \tau$. Here, $\mathcal{D}$ denotes the program's input domain and will be discussed below.

One should think of the type-checker as a module which analyzes both the program and the XML output type, and decides whether all documents produced by the program are valid, and returns `yes` or `no`. If the answer is `no` we would also like to know where in the program typechecking failed. This however may hard, because typechecking is a global property and it may be impossible to say which subexpression caused the typechecker to fail. In XQuery the user can annotate expressions with their expected type[1] reducing the granularity of the type-checker from the entire program to smaller expressions. A more serious issue is that typechecking may not be possible, as we show below, and we may need to settle for an *incomplete type-checker*, which may return false negatives, i.e reject a program even if it typechecks.

A related problem is the *type inference problem*. Here, we are asked to compute, for the given program P, the type $P(\mathcal{D}) = \{P(x) \mid x \in \mathcal{D}\}$. When type inference is possible, we have a simple solution to the typechecking problem: given P, τ, first compute $P(\mathcal{D})$, then check if $\tau_P \subseteq \tau$. However, type inference is not always possible, and in practice we may need to accept some *incomplete type inference*, which computes some superset of $P(\mathcal{D})$. Incomplete type inference can be used to do incomplete type checking.

2.4 Applications

We restrict our discussion to two applications: XML publishing and XML transformations. Each imposes certain kinds of restrictions on the program P, which can be exploited in typechecking.

XML Publishing In this application the XML document is a view over a relational database [FST00, SSB+00]. Here, the program's domain is $\mathcal{D} = \mathit{Inst}(S)$, the set of all database instances of some schema, S. We assume S to include key and foreign key constraints. We will restrict P to perform only simple select-project-join queries on the database, nest the results and add appropriate XML tags, which is sufficient for most XML publishing needs. SilkRoute [FST00] and TreeQL [AMN+01a] are examples of such languages. In this paper we shall use XQuery syntax to express such programs, as in the following example:

Example 2.2 Consider the following relational database schema:

$$S = \begin{cases} \texttt{product(}\underline{\texttt{pid}}\texttt{:STRING, name:STRING, mfrprice:INTEGER),} \\ \texttt{colors(}\underline{\texttt{cid}}\texttt{:STRING, pid:STRING, color:STRING),} \\ \texttt{sale(}\underline{\texttt{sid}}\texttt{:STRING, pid:STRING, price:INTEGER)} \end{cases}$$

Underlined attributes represent keys, and there are foreign key constraints suggested by attribute names. Assuming a standard XML representation of the relational schema [ABS99] the following XQuery program generates an XML view of the relational database, similar to that in Fig. 1:

```
<catalog>
 { FOR $p IN $db/product/tuple
   RETURN <product name = { data($p/name) } >
            <mfr-price> { data($p/price)} </mfr-price>
            { FOR $s in $db/sale/tuple
              WHERE $p/pid = $s/pid
              REUTRN <sale-price> { data($s/sprice) } </sale-price>,
            { FOR $c in $db/color/tuple
              WHERE $p/pid = $c/pid
              REUTRN <color> { data($c/color) } </color>
            }
 }
</catalog>
```

XML Transformations Such applications include mapping between two different XML Schemas, tag renamings, projections, simple field computations, and translation of XML data into HTML (for displaying). In these applications the input is an XML document itself, i.e. the program's domain $\mathcal{D}$ is either $\mathcal{T}_\Sigma$ or some XML type τ, and the program traverses recursively and modifies the input tree. Here, we restrict the language to a subset of XSLT that includes the following features:

- recursive templates

[1] The "upcast" construct, *Expr* : *Type* [FFM+01].

- modes
- `apply-template` can be called along any XPath axis [Cla99]: `child`, `descendant`, `parent`, `ancestor`, `following-sibling`, etc.
- variables can be bound to nodes in the input tree, then passed as parameters
- an equality test can be performed between node ID's, but not between node values.

We omit a formal description of this language. We refer the reader to [BMN00] for a description of a similar fragment (although more powerful than this) and an illustration of the features above. The important property this language fragment has is that it can be expressed in terms of k-pebble tree transducers [MSV00]; readers interested in a proof of this statement may consult [Suc01], where a related language is translated into k-pebble tree transducers.

3 Type Inference and Its Limitations

Consider the program in Example 2.2. Inspecting it carefully, we can actually infer its output type as:

```
TYPE T1 = ELEMENT catalog(T2)
TYPE T2 = (ELEMENT product(T3))*
TYPE T3 = ATTRIBUTE name(STRING), ELEMENT mfr-price(INTEGER)
          (ELEMENT sale-price(INTEGER))*, (ELEMENT color(STRING))*
```

Clearly the program returns a `catalog` tag at the root (hence, `T1`), with several `product` children (hence, `T2`). By analyzing the `RETURN` clause we further infer that the `product` has exactly one `name` attribute, one `mfr-price` child, and several `sale-price` and `color` children. The atomic types are further inferred from the relational database.

This approach is very simple, and very powerful. The general idea is that one infers the type of a `RETURN` expression from the types of its components. The XQuery formal Semantics [FFM+01], building on earlier work on XDuce [HP00], applies this to the entire XQuery language by providing type inference rules for each language construct, and [PV00] describe a complete type inference procedure for a simpler language.

Type inference is used to do typechecking. For example assume that the output to our program needs to conform to the `Catalog` type in Example 2.1. For that it suffices to infer the program's output type `T1`, then check that `T1` $\subseteq$ `Catalog`.

An enhancement to XQuery's type inference rules needed in XML publishing is to use the constraints in the input database schema S to further refine the inferred type. This is because often the correct output type can only be inferred from keys and foreign keys constraints. For example, assume that in the table `sale(`<u>`sid`</u>`, pid, price)` `pid` is also a key, meaning that each product has at most one sale prices. Then we can refine the inferred type by replacing `(ELEMENT sale-price(INTEGER))*` in `T3` with `(ELEMENT sale-price(INTEGER))?`.

Limitations Unfortunately type inference is not complete, as illustrated by the following example.

Example 3.1 The relational schema has a single table, `R(x,y)` and the XQuery is:

```
<result>
   { FOR $x IN $db/R/tuple  RETURN <a/>,
     FOR $x IN $db/R/tuple  RETURN <b/>
   }
</result>
```

This creates an `<a/>` for each tuple in the database, then creates a `<b/>` for each tuple. XQuery infers the following output type:

```
TYPE T = ELEMENT result((ELEMENT a)*, (ELEMENT b)*)
```

But in fact, the real output type is:

$$\texttt{P}(\mathcal{D}) = \{\texttt{ELEMENT result((ELEMENT a)}^{\texttt{n}}\texttt{, (ELEMENT b)}^{\texttt{n}}\texttt{)} \mid \texttt{n} \geq \texttt{0}\}$$

since we have the same number of a's and b's. Obviously, this is not a regular tree language, hence, we cannot "infer" it, and we settle for T instead. But T is an ad-hoc choice, and, as a consequence, the type-checker may fail sometimes to perform typechecking correctly. For example, assume that the user needs to generate XML documents of the following output type:

$$\begin{aligned}\texttt{T1} \quad = \quad & \texttt{ELEMENT result()} \mid \\ & \texttt{ELEMENT result(ELEMENT a, (ELEMENT a)*, ELEMENT b, (ELEMENTb)*)}\end{aligned}$$

It says that there are either no a's and no b's, or there is at least one a and one b (it rules out the cases $(0, 1+)$ and $(1+, 0)$). The program typechecks w.r.t to T1, since it will return no a's and no b's when the database is empty, and at least one a and one b otherwise. But the type-checker rejects the program, because $\texttt{T} \not\subseteq \texttt{T1}$. This is a serious problem, since the user cannot run a program which is correct. The current solution is for the user to rewrite the program in a cumbersome way to help the system typecheck it w.r.t. T1. Moreover, the user won't be able to understand why her program was rejected and how to rewrite it unless she knows all the type inference rules of the language: there is no mathematical property that singles out T as a good candidate for the inferred output type other than the specific choice of inference rules. These, however, may change over time or differ from vendor to vendor.

Can we re-design the type inference rules to choose a better output type ? Unfortunately no. There is no "best" regular tree language, say T', that approximates $P(\mathcal{D})$, because any such approximation can be further improved to $\texttt{T}' - \{\texttt{t}\}$, for some tree $\texttt{t} \in \texttt{T}' - \texttt{P}(\mathcal{D})$.

4 Type Checking and Its Limitations

Another approach is to do typechecking without relying on type inference. If we impose certain restrictions on the programming language, and sometimes on the output type, then one can find algorithms that always do correct typechecking. We illustrate here two such techniques, for XML publishing and XML transformation.

4.1 Typechecking for XML Publishing

Recall that the input to the program P is a relational database. Given P and an output type τ, we can typecheck P by enumerating all "small" input databases (up to a size which depends only on P and τ), running P on each of them, and checking that the output conforms to τ. If for some database the output fails to validate against τ then we return no; otherwise we return yes. Since there are only finitely many databases up to a given size, the algorithm will eventually halt. This procedure, described in [AMN+01a], is rather inefficient, but it always returns the correct answer, and at least provides an existence proof that the typechecking problem is decidable: more efficient typechecking algorithms may be discovered in the future.

However, we need to impose the following two restrictions on the output type τ for this procedure to be correct.

- τ is a DTD type.
- all regular expressions in τ are "star-free".

Star-free regular expressions [RS97, Per90] are much more expressive than their name suggests. We are not allowed to use the Kleene closure, however we can use the complement, denoted *compl*, and the empty set, $\emptyset$. For example, if $\Sigma = \{a, b, c\}$, then $compl(\emptyset)$ denotes Σ^*, $compl(\Sigma^*.b.\Sigma^* \mid \Sigma^*.c.\Sigma^*)$ denotes a^*. Thus, regular expressions like $a^*.b^*.(c \mid a.b^*)$ and even $(a.b)^*$ are star-free. In fact, *all* examples of regular expressions used in this paper are star-free, and one could well argue that star-free regular expressions are sufficient for virtually all practical applications. An example which is not star-free is $(a.a)^*$.

Theorem 4.1 *[AMN+01a] Typechecking for XML publishing is decidable when the query language is restricted as in Sec. 2.4, and the output type is a star-free DTD.*

It is possible to lift the restriction on the output type to be star-free, but then we need to require the queries to have no projections [AMN+01a].

Limitations Unfortunately the restrictions in Theorem 4.1 are critical: if we allow output types that are not DTDs, or try to increase the expressive power of the language, then typechecking becomes undecidable [AMN+01a]. This has two consequences in practice. First, we need to accept some incomplete typechecking algorithm after all. Second, type inference remains the best known approximation algorithm in this case: if we apply the procedure described in this section (enumerating all "small" databases) in cases that do not statisfy the conditions in Theorem 4.1 then one may get false positives, i.e. it may fail to detect that a program does not typecheck, which is inacceptable for most applications.

4.2 Typechecking for XML Transformation

Recall that here we consider a certain XSLT fragment, discussed in Sec. 2.4. The unexpected property that this fragment has is that one can do *inverse* type inference. Formally, if $P : \mathcal{T}_\Sigma \to \mathcal{T}_\Sigma$ is a transformation expressed in this language and $\tau \subseteq \mathcal{T}_\Sigma$ is a type (i.e. a regular tree language), then $P^{-1}(\tau) = \{x \mid P(x) \in \tau\}$ is also a type (i.e. regular tree language). We illustrate this with an example.

Example 4.2 The following program, `P`, is an XSLT variant of the query in Example 3.1:

```
<xsl:template match="p">
  <r> <xsl:apply-templates mode="mode-a"/>
      <xsl:apply-templates mode="mode-b"/>
  </r>
</xsl:template>
<xsl:template mode="mode-a" match="q">
   <a/>
</xsl:template>
<xsl:template mode="mode-b" match="q">
   <b/>
</xsl:template>
```

For example on some input document of the form:

```
<p> <q/> <q/> . . . <q/> </p>
```

`P` returns results of the form:

```
<r> <a/> . . . <a/> <b/> . . . <b/> </r>
```

(same number of `a`'s and `b`'s.) The same discussion as in Example 3.1 applies here: the naively inferred type, `T`, is incomplete, and cannot be used to typecheck if the output type is `T1`. However, assume we fix the output type `T1` and decide to do inverse type inference, i.e. compute $\texttt{P}^{-1}\texttt{(T1)}$. Assuming $\Sigma = \{\texttt{p, q}\}$, the result of inverse type inference is simply `Any`, defined as:

$$\begin{aligned}\texttt{P}^{-1}\texttt{(T1)} &= \texttt{Any}\\ \texttt{Any} &= \texttt{ELEMENT p(Any*) | ELEMENT q(Any*)}\end{aligned}$$

(any XML tree over `p` and `q`). For a more interesting example, consider the output type:

```
T2 = ELEMENT r(ELEMENT a, ELEMENT a, (ELEMENT a)*, (ELEMENT b)*)
```

(it requires at least two `a`'s). Then the inferred input type is:

$$\texttt{P}^{-1}\texttt{(T2)} \;=\; \texttt{ELEMENT p(Any*, ELEMENT q(Any*), Any*, ELEMENT q(Any*), Any*)}$$

(the root must be labeled `p` and it must have at least two `q` children).

While this example illustrates inverse type inference only for a simple, non-recursive XSLT program, inverse type inference is possible for much more complex programs, that are recursive, use modes, navigate through the XML trees using all XPath axis, and make use of XSLT variables and parameters in the restricted way discussed in Sec. 2.4. This enables us to do typechecking, as follows. Given $P : \mathcal{D} \rightarrow \mathcal{T}_\Sigma$, and an output type τ, compute $P^{-1}(\tau)$ then check $\mathcal{D} \subseteq P^{-1}(\tau)$: recall that $\mathcal{D}$ is either $\mathcal{T}_\Sigma$, or some regular tree language $\tau \subseteq \mathcal{T}_\Sigma$. Formally:

Theorem 4.3 *[MSV00, Suc01] For any program P in the XSLT fragment considered in Sec. 2.4 and any regular tree language τ, $P^{-1}(\tau)$ is also a regular tree language. In consequence typechecking for this language is decidable.*

The algorithm resulting from the proof in [MSV00] is very inefficient (hyperexponential), but it applies to a more powerful formalism than the XSLT fragment described here. A more efficient algorithm (exponential) for a different fragment of XSLT is described in [Toz01].

Limitations Like before, this result only holds for the XSLT fragment described. If we add, for example, joins to the language (by allowing comparisons between parameters' values) then typechecking becomes undecidable [MSV00], hence we need to accept some approximation algorithm here too. However, unlike in XML publishing, a typechecking algorithm may be better than one based on type inference[2]: [MSV00] describes an extension of the typechecking algorithm to queries with joins, by replacing each join comparison with a non-deterministic choice, then typechecking the resulting non-deterministic transformation. This may return false negatives, but it can typecheck more programs than type inference (e.g. the program in Example 4.2).

5 Discussion

XML typechecking is clearly an important problem, because we need to validate dynamically generated XML documents. We have seen two approaches to typechecking: an incomplete method, based on heuristic type inference, and two complete methods, using specialized algorithms. None is completely satisfactory in practice. The first results in systems that reject correct programs without warnings. The second may work in some cases, but the decidable cases seem to be more the exception than the rule. Moreover, even where typechecking is decidable, the complexity is high.

Today, the front-runner in practice is the type inference method, because it is easy to extend, even if incompletely, to a variety of language constructs[3]. What is intriguing about type inference is that, despite its theoretical incompleteness, it seems to be "complete" for practical applications. In our example in Sec. 3, the type `T1` that caused typechecking to fail is artificial, and one can argue that such types do not occur in practice. One could make the argument that with the inferred type `T` the system can correctly typecheck the program for all "practical" output types (since the output types on which it fails, like `T1`, are "artificial"). But so far no formal statements of this kind have been proven. This raises several questions:

- Is there a notion of "practical" types for which type inference is complete ? This notion should exclude the type `T1` in Sec. 3, while allow interesting types with Kleene closures (perhaps with the limitations of star-free languages), alternations, and optional elements.
- Is it possible to issue a warning when type inference fails ? We would like the typechecking algorithm to return `yes`, `no`, or `unknown` rather than return false negatives.
- As alternative techniques to type inference one might consider approximation, or randomized typechecking algorithm. Of course, type inference offers an approximation algorithm already, but it is not clear yet that is is the best in practice.

Acknowledgments The author thanks Frank Neven for his comments. The author is supported by the NSF CAREER Award 0092955, a Sloan Fellowship and a gift from Microsoft Research.

[2]Except for the complexity.

[3]*Inverse* type inference (Sec. 4.2) seems a promising approach too, but is less investigated.

References

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

[AMN+01a] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking XML views of relational databases. In *LICS*, pages 421–430, 2001.

[AMN+01b] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. In *PODS*, pages 138–149, 2001.

[BKMW98] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree languages over non-ranked alphabets, 1998. Available at `ftp://ftp11.informatik.tu-muenchen.de/pub/misc/caterpillars/`.

[BMN00] G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. In Lloyd et al., editor, *Computational Logic – CL 2000*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1137–1151. Springer, 2000.

[Cla99] James Clark. XML path language (XPath), 1999. available from the W3C, `http://www.w3.org/TR/xpath`.

[FFM+01] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 formal semantics, 2001. available from the W3C, `http://www.w3.org/TR/query-semantics`.

[FST00] M. Fernandez, D. Suciu, and W. Tan. SilkRoute: trading between relations and XML. In *Proceedings of the WWW9*, pages 723–746, Amsterdam, 2000.

[HP00] Haruo Hosoya and Benjamin C. Pierce. XDuce: An XML processing language (preliminary report). In *WebDB'2000*, pages 226–244, 2000. http://www.research.att.com/conf/webdb2000/.

[HP01] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for xml. In *ACM SIGPLAN, SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 67–80, January 2001.

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

[Per90] D. Perrin. Finite automata. In *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 1, pages 1–57. Elsevier, Amsterdam, 1990.

[PV00] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proceedings of PODS*, pages 35–46, Dallas, TX, 2000.

[RS97] G. Rozenberg and A. Salomaa. *Handbook of Formal Languages*. Springer Verlag, 1997.

[Sei90] Helmut Seidl. Deciding equivalence of finite tree automata. *SIAM J. Comput*, 19(3):424–437, 1990.

[SSB+00] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as xml documents. In *Proceedings of VLDB*, pages 65–76, Cairo, Egipt, September 2000.

[Suc01] D. Suciu. Typechecking for semistructured data. In *Proceedings of the International Workshop on Database Programming Languages*, Italy, September 2001. Springer Verlag. (to appear).

[Tho90] W. Thomas. Automata on infinite objects. In *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 4, pages 133–192. Elsevier, Amsterdam, 1990.

[Toz01] Akihiko Tozawa. Towards static type inference for xslt. In *Proc. of ACM Symposium on Document Engineering*, 2001.