Automata and Logic (CSC2428)
Typechecking for XML Transformations

Alvin Chin
achin@cs.toronto.edu

November 25, 2005

1 Introduction

XML pervades the web, and XML is the language for data exchange among disparate data sources like a relational database, an XML database, and a spatial database. A data provider or party creates a common XML schema and agrees to produce only XML data conforming to that schema. Since different parties have their own schemas, the challenge is how to communicate among different parties and exchange data. Transformations need to be done to convert from one XML document to another according to the associated schema. People perform queries on the web to get responses to questions. As illustrated by Figure 1 , the query response is an XML document which is a result from the many XML transformations and aggregation of data pulled together to satisfy that query.

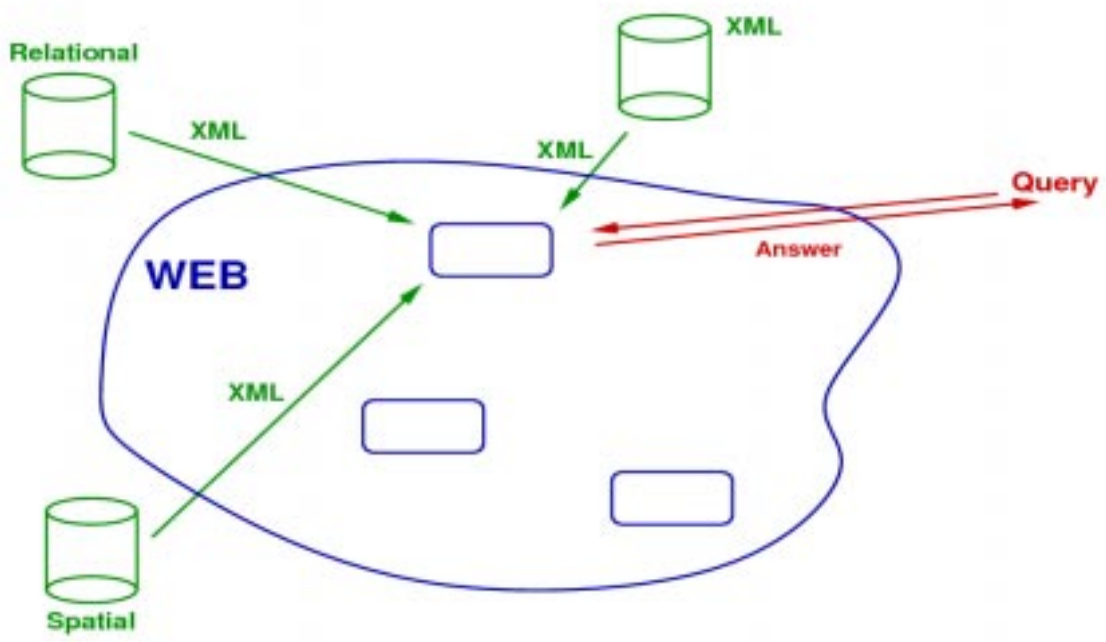

Figure 1: Performing a query on the web and how XML is used to return the query response [2]

2 The Typechecking Problem

The problem that arises from querying and performing transformations for XML data exchange on the web is called the *typechecking problem* and can be illustrated in Figure 2. The main reference from which this paper is based on is from Milo, Suciu and Vianu called Typechecking for XML Transformers from the PODS 2000 conference [3].

The question is this: Does every XML document that comes from an XML transformation T, and an XML schema S_{in} (input schema) satisfy the output schema S_{out}? For example, a mobile device wants to view information from Google Base but Google Base has its own XML schema (S_{out}) and the mobile device's browser is in WML (Wireless Markup Language) (Sin).

2.1 Types of typechecking

In order to solve the typechecking problem, we need to look at the different types of typechecking that exist. There are two types of typechecking: dynamic typechecking and static typechecking. Dynamic

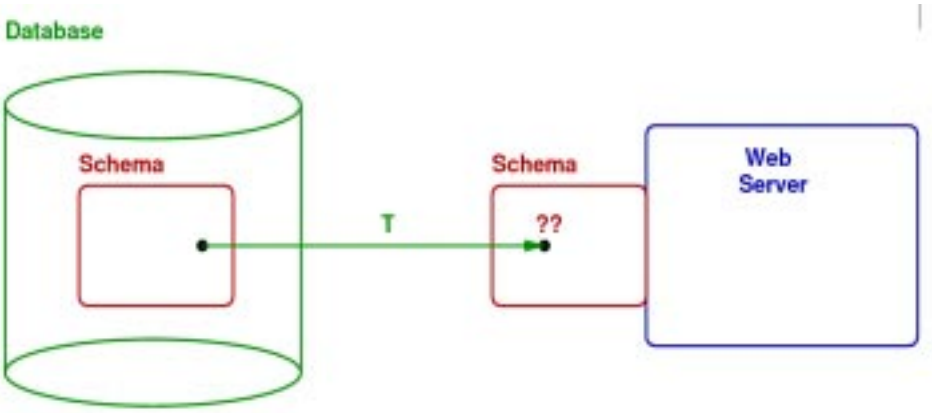

Figure 2: The typechecking problem [2]

typechecking involves validating each XML document at run time. The drawbacks of dynamic typechecking are that it is resource-intensive and the XML document has to be parsed in real-time, which makes it slow. Static typechecking, on the other hand, validates each XML document at compile time. The focus of this paper is on static typechecking.

2.2 Addressing the typechecking problem

2.2.1 Issues

There are basically two issues in addressing the typechecking problem. First, there is no generally accepted and universally agreed upon XML transformation language. The authors [3] address this problem by defining an abstract model called a *k-pebble tree transducer* which allows all transformations in XML query languages to be expressed without joins on data values. Second, regular tree languages can be used to formalize schemas without having to consider the type of schema (like DTD or XML Schema). In fact, it is known that a DTD forms a regular tree language.

2.2.2 Previous work

Several researchers have addressed the typechecking problem, notably Alon et al. who conclude that typechecking becomes undecidable when there are data or attribute values [1]. From the authors, they conclude that typechecking is decidable for a large fragment of tree transformations using structural properties [3].

2.3 Formal definition of the typechecking problem

We can provide a formal definition of the typechecking problem. Given an input tree language τ_{in}, an output tree language τ_{out}, and XML transformation T, verify that:

$$\forall t \ \in \ \tau_{in} \Longrightarrow T(t) \ \in \ \tau_{out}$$

3 Background to Solving the Typechecking Problem

The solution for solving the typechecking problem involves modelling the XML schemas as regular trees and XML transformations [5] as *k-pebble tree transducers*. Before showing the solution, since we know regular trees, we need to know what a k-pebble tree transducer is.

3.1 *k*-Pebble Tree Transducer

A *k-pebble tree transducer* uses k pebbles to mark certain nodes in the tree. The pebbles are ordered and numbered from *1,2,,k*. Pebbles are placed in order, and removed in reverse order like pushing and popping from a stack. Only the highest-numbered pebble on the tree can be moved.

3.1.1 How a *k*-pebble tree transducer works

Figure 3 illustrates the k-pebble tree transducer in action on a tree with k=5.

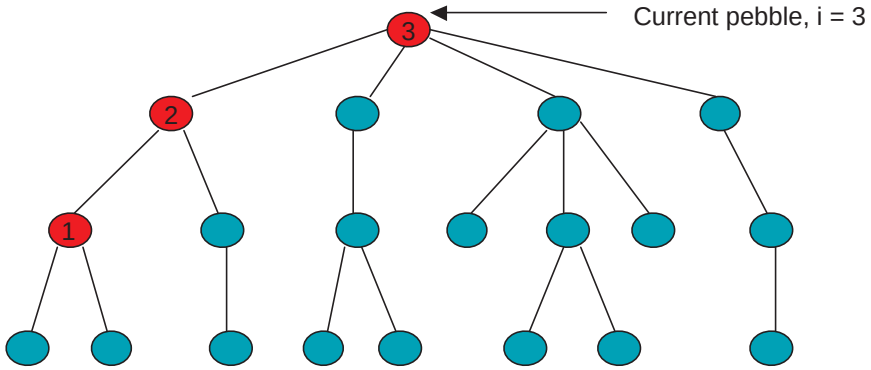

Figure 3: Example of k-pebble tree transducer with k=5

First, pebble 1 is placed on the root. The next pebble is placed onto the root which pushes the other pebbles down the tree. As can be seen from the above figure, 3 pebbles have been placed on the tree. We can only move the highest-numbered pebble (in this case, it is pebble 3) by examining the current state, the symbol under the current pebble, and the presence or absence of the other *i-1* pebbles where the current pebble being moved is *i*. Transitions for pebble i are either move and output. Valid moves are down-left, down-right, up-left, up-right, stay, place-new-pebble and pick-current-pebble. The output node is returned with a symbol from the output alphabet.

The output from the k-pebble tree transducer is as follows. First, we start with a single computation branch and no output nodes. Then, the computations result in some top fragment of the output tree. The remaining output subtrees are then computed. The outputs $a' \in \Sigma'$ can be binary ($a' \in \Sigma'_2$) in which there are two computation branches that compute the left and right child, or they can be nulary ($a' \in \Sigma'_0$) in which only the leaf is output and the branch of computation halts. a' is the output symbol which is from the output alphabet Σ, Σ'_2 indicates the output alphabet for the left and right child, and Σ'_0 indicates the output alphabet for the leaf.

3.1.2 Formal definition of a *k*-pebble tree transducer

A k-pebble tree transducer is

$$T = (\Sigma, \Sigma', Q, q_0, P) \text{ where :}$$

1. Σ, Σ' are the ranked input and output alphabets
2. Q is a finite set of states and is partitioned into: $Q = Q_1 \bigcup ... \bigcup Q_k$
3. $q_0 \in Q_1$ is the initial state

4. P is a finite set of transitions:
   $(a, \overline{b}, q_1^{(i)}) \longrightarrow (q_2^{(i)}, stay)$
   $(a, \overline{b}, q_1^{(i)}) \longrightarrow (q_2^{(i)}, down - left)$
   $(a, \overline{b}, q_1^{(i)}) \longrightarrow (q_2^{(i)}, down - right)$
   $(a, \overline{b}, q_1^{(i)}) \longrightarrow (q_2^{(i)}, up - left)$
   $(a, \overline{b}, q_1^{(i)}) \longrightarrow (q_2^{(i)}, up - right)$
   $(a, \overline{b}, q_1^{(i)}) \longrightarrow (q_2^{(i+1)}, place - new - pebble)$
   $(a, \overline{b}, q_1^{(i)}) \longrightarrow (q_2^{(i-1)}, pick - current - pebble)$
   $(a, \overline{b}, q_1^{(i)}) \longrightarrow (a_0^{'}, output0)$
   $(a, \overline{b}, q_1^{(i)}) \longrightarrow (a_2^{'}(q_1^{(i)}, q_2^{(i)}), output2)$

   $\overline{b} \in \{0,1\}^{i-1}, for\ i = 1, \overline{b} = \epsilon$

What this means is that P has a set of transitions in which the move operations for current pebble i are stay, down-left, down-right, up-left, and up-right given an input symbol a, the presence/absence of the previous *i-1* pebbles denoted by $\overline{b} \in \{0,1\}^{i-1}$ (obviously if $i = 1$, then there is no $\overline{b}$), then the state transitions from q_1 to q_2. If the move is to place a new pebble, then pebble *i+1* is placed and the state moves from q_1 to q_2. If the current pebble is picked, then pebble i is in state q_1, and the previous pebble *i-1* enters the q_2 state. For output transitions, if the output is nulary, then the leaf is output $a_0^{'}$, otherwise if it is binary output, then left child enters q_1, the right child enters q_2 and the output is the left and right child along with the output symbol $a_2^{'}$.

3.1.3 Example of a *k*-pebble transducer (1-pebble)

This example copies the input tree to the output tree and is obtained from Example 3.2 of [3].

$$T = (\Sigma, \Sigma', q, q_1, q_2, q_0, q, P) \text{ where :}$$

1. Σ, Σ' are the ranked input and output alphabets

2. P is the following:
   $(a_2, q) \longrightarrow (a_2(q_1, q_2), output2)$ for all $a_2 \in \Sigma_2$
   $(a_2, q_1) \longrightarrow (q, down - left)$
   $(a_2, q_2) \longrightarrow (q, down - right)$
   $(a_0, q_1) \longrightarrow (a_0, output0)$ for all $a_0 \in \Sigma_0$

As can be seen, every input symbol is then output. *output0* is a leaf, while *output2* is the left and right child. Note that there is no $\overline{b}$ because k = 1, and $\overline{b} = \epsilon$ so it is omitted.

There are other examples of k-pebble transducers like in Example 3.3 and 3.4 of [3].

3.1.4 Complexity of *k*-pebble transducer

Let T be a fixed k-pebble transducer. Then for each input tree t,

1. the set $T(t)$ is a regular tree language

2. one can construct in PTIME (in the size of t) a regular tree automaton $\mathcal{A}_t$ that accepts the language $T(t)$

3.1.5 Restating the typechecking problem - formal definition

Given an input tree type τ_{in}, an output tree type τ_{out}, and a k-pebble tree transducer T, verify:

$$\forall t \ \in \ \tau_{in} \Longrightarrow T(t) \ \in \ \tau_{out}$$

4 Solution to the Typechecking Problem

Now that we have the background information on k-pebble tree transducers, we can solve the typechecking problem. In solving the typechecking problem, the authors examine a closely related problem which is type inference. The idea is that if type inference can be solved, then that immediately leads to solving the typechecking problem.

4.1 The type inference problem

Given an input tree type τ_{in}, an output tree type τ_{out}, and a k-pebble tree transducer T, construct $\tau_{out}^{'}$ such that

$$\tau_{out}^{'} = T(\tau_{in})$$

How can type inference solve typechecking? If we infer that $\tau_{out}^{'} = T(\tau_{in})$, then we can check that $\tau_{out}^{'} \subseteq \tau_{out}$. This implies that $T(\tau_{in}) \subseteq \tau_{out}$ and therefore we have solved the typechecking problem. However, type inference is not possible for all types and transformations as can be seen from Example 4.2 from [3]. Since type inference is not possible, then what can we do? The authors discovered that inverse type inference is possible.

4.2 The inverse type inference problem

If we can solve the inverse type inference problem, then this admits type inference which immediately yields a solution to typechecking. The inverse type inference problem is this. Given an output type τ and a k-pebble transducer T, construct τ^{-1} (inverse inferred type) such that

$$\tau^{-1} = \{t \mid T(t) \subseteq \tau\}$$

To typecheck T with respect to input τ_{in} and output τ_{out}, find the inverse type τ_{out}^{-1} for T and τ_{out}. Then check that $\tau_{in} \subseteq \tau_{out}^{-1}$. This means that $\tau_{out}^{-1} = T(\tau_{in})$ and since $\tau_{out}^{-1} \subseteq \tau_{out}^{-1}$, then $T(\tau_{in}) = \tau_{out}$. So, a solution to the inverse type inference problem immediately yields a solution to the typechecking problem.

4.3 Solving the inverse type inference problem

The authors solve the inverse type inference problem using the following three steps: 1) define an acceptor variant of the transducer, called k-pebble automaton, 2) for each k-pebble transducer T and type τ, the complement of $\{t \mid T(t) \subseteq \tau\}$ (denoted by $\overline{\{t \mid T(t) \subseteq \tau\}}$) is recognized by some k-pebble automaton, and 3) prove that every k-pebble automaton recognizes a regular tree language.

4.3.1 Define a k-pebble automaton

A k-pebble automaton is a 4-tuple (Σ, Q, q_0, P) where:

1. Σ, Q, q_0 are as in a k-pebble transducer
2. P is a set of transitions. Move transitions are as in a k-pebble transducer. Output transitions are replaced by:
   $(a_0, \overline{b}, q^{(i)}) \longrightarrow (branch0)$
   $(a_2, \overline{b}, q^{(i)}) \longrightarrow ((q_1^{(i)}, q_2^{(i)}), branch2)$

A k-pebble automaton is like a k-pebble transducer except that it produces no output, and just accepts or rejects input. *branch0* stops the current computation branch and accepts, whereas *branch2* spawns two independent computations in states $q_1^{(i)}$, and $q_2^{(i)}$.

4.3.2 Inverse type inference with *k*-pebble automata

The next step is to realize inverse type inference with k-pebble automata. For each k-pebble transducer T and type τ, there exists a k-pebble automaton $\mathcal{A}$ such that

$$inst(\mathcal{A}) = \overline{\{t \mid T(t) \subseteq \tau\}}$$

Note that this is the inverse of type inference. The proof of this can be found in Appendix B, Proof of Proposition 4.5 from [3].

4.3.3 *k*-pebble tree automata accepts regular tree languages

The third step is to prove that every k-pebble tree automaton recognizes a regular tree language. The proof of this can be found in Appendix B, Proof of Theorem 4.6 from [3].

By solving the inverse type inference problem, we have immediately solved the typechecking problem.

4.4 Limitations of the typechecking approach

Even though k-pebble tree automata is concise and convenient, the problem is that it has high complexity because it is hyperexponential in k. A second problem is that it lacks data values. Data in XML ($\#PCDATA$) is modeled in a tree with an infinite alphabet D. To address this, the k-pebble transducer can be extended with 3 kinds of transitions. The first transition is a comparison predicate that checks whether $x = y$, state q_1 is entered when $x = y$ and state q_2 is entered when $x <> y$. The second transition is unary comparison on data values, while the third transition is an output transition that copies the data value from the input tree to the output tree.

5 XML Transformation Languages with Typechecking

We have described the solution and the formal model and definition for the typechecking problem. This section explores the XML transformation languages that have typechecking in practice [4]. XDuce is considered the first programming language with typechecking of XML operations using schemas. Other languages that Moller compare are XACT, XJ, XOBE, JDOM (Java implementation of DOM), JAXB, HaXml, Cω, XQuery (SQL for XML), XSLT, and tree transducers.

In practice, XML transformation languages do not perform rigorous typechecking because of the complexity and for most implementations, an approximation is adequate.

References

[1] Wim Martens and Frank Neven. Typechecking top-down xml transformations. ICDT 2003 talk, 2003.

[2] Wim Martens and Frank Neven. Frontiers of tractability for typechecking simple xml transformations. PODS 2004 talk, 2004.

[3] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for xml transformers. In *PODS '00: Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 11–22, New York, NY, USA, 2000. ACM Press.

[4] Anders Mller and Michael I. Schwartzbach. The design space of type checkers for xml transformation languages. In *Proceedings of 10th International Conference on Database Theory: Lecture Notes in Computer Science*, volume 3363, pages 17–36. Springer-Verlag, 2005.

[5] Frank Neven. Automata, logic and xml. In *Proceedings of 16th International Workshop of the EACSL on Computer Science Logic: Lecture Notes in Computer Science*, volume 2471, pages 2–26. Springer-Verlag, 2003.