Typechecking for XML Transformers*

Tova Milo
Tel Aviv University
milo@tau.math.ac.il

Dan Suciu
University of Washington
suciu@cs.washington.edu

Victor Vianu†
U.C. San Diego
vianu@cs.ucsd.edu

Abstract

We study the typechecking problem for XML transformers: given an XML transformation program and a DTD for the input XML documents, check whether every result of the program conforms to a specified output DTD. We model XML transformers using a novel device called a k-pebble transducer, that can express most queries without data-value joins in XML-QL, XSLT, and other XML query languages. Types are modeled by regular tree languages, a robust extension of DTDs. The main result of the paper is that typechecking for k-pebble transducers is decidable. Consequently, typechecking can be performed for a broad range of XML transformation languages, including XML-QL and a fragment of XSLT.

1 Introduction

Traditionally, database query languages have focused on data retrieval, with complex data transformations left to applications. The new XML data exchange standard for the Web, and emerging applications requiring data wrapping and integration, have shifted the focus towards data transformations. A typical scenario is provided by a mediator system, where the integrated view is constructed from the data sources [20]. Several languages have been proposed to express XML transformations [15, 12, 11, 33].

XML data can have complex structure and flexible typing. Elements may be nested to arbitrary depth and new tags (element names) can be created at will. The structure of an XML document may be constrained by a Document Type Definition (DTD). An XML document which specifies a DTD and which also conforms to that DTD is called *valid*. The benefits of the information provided

*Work supported in part by the U.S.-Israel Binational Science Foundation under grant number 97-00128

†This author supported in part by the National Science Foundation under grant number IIS-9802288.

by DTDs are numerous. Some are analogous to those derived from schema information in relational query processing. Perhaps most importantly to the context of the Web, DTD can be used to validate data exchange. In a typical scenario, a user community would agree on a common DTD and on producing only XML documents valid with respect to the specified DTD. This raises the issue of *typechecking*, which comes in two flavors: dynamic, and static. Dynamic typechecking validates at runtime each XML document produced, and can be performed by a validating parser (several are publicly available). Its obvious drawback is that errors are only detected at runtime. In contrast, this paper studies the *static typechecking* problem (simply called *typechecking* from here on). It consists of verifying at compile time that *every* XML document which is the result of a specified transformation satisfies the output DTD. DTDs restricting the input documents may also be available as input to the typechecker. As we shall see, typechecking of XML transformations requires novel query analysis techniques.

Studying formally the typechecking problem is complicated by the fact that XML transformation languages and the notion of DTD are constantly evolving. More specifically:

- There is no generally accepted XML transformation language. Several transformation languages exist for XML, including XML-QL and XSLT [15, 12, 11]. Other languages result from restricting query languages for semistructured data to trees (e.g. Lorel, StruQL, UnQL [27, 18, 9]). Yet more languages can be defined using algebras, motivated mainly by systems considerations (see [5]). These languages define incomparable classes of transformations, and it is not a priori clear how to define a single query language subsuming them, that could provide a relevant, broad framework for studying the typechecking problem.

- There is no generally accepted notion of schema for XML. DTDs are the most widely accepted because they are part of the existing XML standard, but several variations and extensions of DTDs are being considered by the XML community. A recent effort tries to unify these extensions into a single framework called XML-Schema [3, 7]

We address these difficulties as follows. *First* we define an abstract and rather general model of XML transformation languages called a *k-pebble tree transducer*, expressing transformations on ordered, labeled trees. Up to k pebbles can be placed on the input tree, moved up and down the tree, and removed, subject to a *stack discipline* where only the highest numbered pebble currently placed on the tree can be moved. Traditional top-down regular tree transducers [34] correspond to the particular case when $k = 1$ and the pebble is allowed to move only downwards. All transformations expressible without joins on data values in XML-QL, LOREL, UnQL, StruQL, and a fragment of XSLT, can also be expressed by k-pebble transducers. Therefore, the results on typechecking described in this paper also apply to these languages, and are likely to remain relevant as the languages evolve. In our view, the notion of k-pebble tree transducers is one of the main contributions of this work. Beyond their role in the study of typechecking, we believe they are of interest on their own as a simple, robust model for tree transformations. In particular, the transformations defined by k-pebble transducers have PTIME data complexity (in a sense made precise in the paper). From a practical perspective, k-pebble transducers correspond directly to simple recursive functions written in C++ or Java, accessing both the input and the output tree via a restricted API. The API for the input tree consists of k pointers to nodes in the tree. For each pointer we can read the tag, test whether the current node is a leaf, or the root, and advance the pointer to a child (given by the child number) or to the parent of the current node. The stack discipline requires us to reset (i.e. move to the root) pointers $i+1, i+2, \ldots, k$ whenever we move pointer i. The API for the output tree allows us to write one output node at a time, then call recursive functions in parallel to construct the node's children. All the functions inherit the current function's state, and they know the position of the child of the output node on which they act.

Second, as type formalism we adopt *regular tree languages*. A regular tree language is the set of trees accepted by a regular tree automaton (of which there are many equivalent variations, see [34] for the ranked case and [8] for the unranked case). It is easy to see that DTDs define a set of valid XML documents which (when viewed as labeled trees) form a regular tree language. Moreover, various proposed extensions of DTDs define regular tree languages as well; e.g., several such proposals decouple element names from tag names [4, 14, 32, 25, 31]. It is easily seen that DTDs with decoupling define *precisely* the regular tree languages [4, 32, 13]. Note that regular tree languages cannot capture semantic constraints, such as keys and foreign keys, or explicitly describe unordered data, hence our results do not carry over to such schemas.

Formally, the typechecking problem studied in the paper is the following: given two types (regular tree languages) τ_1, τ_2, and a k-pebble transducer T, verify that $T(t) \subseteq \tau_2$ for every $t \in \tau_1$. [1] The main result of the paper is that typechecking is decidable. To understand our approach, consider first a closely related problem: type inference. In our framework, the type inference problem is the following: given a k-pebble transducer T and an input type τ_1, construct a type τ_1' that describes precisely the image of T on inputs satisfying τ_1. If type inference were possible, it would solve the typechecking problem: to check conformance of T to τ_2 on inputs satisfying τ_1, first infer τ_1', then check that $\tau_1' \subseteq \tau_2$ (inclusion of regular tree languages is decidable). Unfortunately, as discussed in the paper, type inference in the above sense is not possible because, even for simple XML queries, their image is not a regular tree language (and there is no smallest regular tree language containing the image). However, the failure of type inference for transformation languages has an interesting twist, which lies at the core of our approach: the *inverse* of any k-pebble transfor-

[1] The transducer T may be non-deterministic; $T(t)$ denotes the set of possible outputs of T on input t.

mation admits type inference. That is, given a k-pebble transducer T and an *output* type τ, the set of trees t such that $T(t) \subseteq \tau$ is a regular tree language τ^{-1} which can be inferred[2] from T and τ. We refer to the problem of finding τ^{-1} given T and τ as the *inverse type inference* problem. Its solution is the main result in our paper, and it implies immediately a typechecking algorithm. The technical development leading to the solution of the inverse type inference problem has three main stages:

(i) define an acceptor variant of the k-pebble transducer, called *k-pebble tree automaton*;

(ii) show that for each k-pebble transducer T and type τ, the set $\{t \mid T(t) \subseteq \tau\}$ is recognized by some k-pebble automaton; and,

(iii) prove that every k-pebble automaton recognizes a regular tree language.

Limitations There are two limitations of the typechecking method described here. The first is that it does not apply to transformations expressed with joins on data values. All XML query languages mentioned here can express such joins with conditions of the form `x` $=$ `y`, where `x, y` are variables bound to atomic values in the input XML data (`#PCDATA`, in XML terminology). The domain of such values is infinite (e.g. strings), while k-pebble transducers are defined only for trees with labels from a finite alphabet (corresponding to the tags in XML). To capture data values one can extend the data model to trees whose internal nodes are labeled from a finite alphabet (as before), and whose leaves are labeled from an infinite domain of data values. If we allow the k-pebble transducers to test only unary predicates on the data values (i.e. allow predicates such as `(x > 5)` or `(x like ''Smith''))`, then the same typechecking techniques still work. However, if we extend the transducers to allow them to perform joins between these data values (i.e. include the predicate `x` $=$ `y`), then typechecking becomes undecidable[3]. Thus, in the case of XML-QL, Lorel, StruQL, and UnQL, the type checking technique described here applies only to queries without joins on data values. We briefly describe in Section 5 a simple idea by which one can apply these techniques to an important class of queries with data joins, but we leave a full treatment for future work. We should clarify however that some languages (Lorel, StruQL) frequently use joins on node oids to search for patterns in the input data: such joins are also of the form `x` $=$ `y`, but with `x, y` bound now to nodes in the tree: these conditions *can* be expressed by k-pebble transducers, and, hence, typechecking works for these queries too. Finally, the case of XSLT is more interesting. In its early version (called XSL) it could not express data value joins and its expressive power for tree transformations was strictly less than that of k-pebble transducers. Recent additions to the language however include data value joins and constructs made

[2]For simplicity, we sometimes blur the distinction between a regular tree language and its specification as a regular tree automaton.

[3]This follows from a straightforward reduction from the finite satisfiability problem for first-order logic.

it at least as expressive as k-pebble transducers for tree transformations [6]. An analysis of the expressive power of the full language has to be postponed until the language settles.

The second limitation is the complexity. Our typechecking algorithm has non-elementary complexity, and we show that this is also a lower bound. The main source of complexity is the number of pebbles of the transducer. However, as discussed in the paper, there are many restricted cases of practical interest for which the complexity of typechecking is reasonable, because only one or two pebbles are needed in the corresponding transducer.

Related Work Most practical XML languages that perform type checking do this through type inference: XDuce [23] uses such an approach, and so does XQuery [17]. The idea is to infer the most general output type of a program, then checking that this type is contained in the specified output type. Checking containment of regular tree languages is decidable, and a practical algorithm specifically designed for XML typechecking is described in [24].

As we show in this paper such most general output type may not exists: in these cases the type inference will return some approximation, and, as a consequence, may fail to typecheck correct programs. Despite this limitation type inference is appealing in practice because of it is relatively easy to implement and to extend to a variety of languages.

Two kinds of tree transducers are described in [34]: top-down and bottom-up (called there *root-to-frontier* and *frontier-to-root*). Any top-down transducer can be simulated by a 1-pebble transducer with moves restricted to go downwards only. Interestingly, it is an open problem whether k-pebble transducers can simulate all bottom-up transducers (more on that below). Both top-down and bottom-up transducers are known to have the inverse type inference property: k-pebble transducers however can express transformations which are not expressible by either top-down or bottom-up transducers.

The k-pebble transducers generalize several formalisms considered in the literature. Aho and Ullman [2] introduce *tree-walk* automata. These devices have a single head which can move up and down the tree, starting from the root. The set of tree languages accepted by a tree-walk automata is included in the set of regular tree languages, but it is a long standing open problem whether the inclusion is strict [16]. The question whether k-pebble transducers can simulate all bottom-up transducers can be reduced to this open problem (in fact, as we explain in the sequel, the two problems become equivalent when $k = 1$). Note that for the case of strings, the analog of tree-walk automata are precisely the two-way automata, which are known to express all regular languages.

Engelfriet et al. [16] extend the tree-walk automata with marbles and one pebble, and prove that the new machines capture precisely the regular tree languages. Unlike pebbles, marbles are in infinite supply, but the input head is restricted to stay below the last marble placed. Our k-pebble automata (the acceptor version of the k-pebble transducer) can also capture all regular tree languages, but use *branching*, a different extension of the tree-walk automata with a form of *and*-alternation.

String automata with *or*-alternations, *and*-alternations, and a rather restricted form of k-pebbles are considered by Goberman and Harel [21]. They prove certain lower bounds in the gap of succinctness of the expressibility of such automata. We do not address the corresponding lower bounds for k-pebble transducers or automata, but we note that the emptiness problem for k-pebble automata has a non-elementary lower bound.

Query automata are described by Neven and Schwentick in [29]. This work was the first to use Monadic Second Order logic (MSO) to study query languages for XML. Query automata however differ significantly from k-pebble transducers: they take an XML input tree and return a set of nodes in the tree. By contrast a k-pebble transducer returns a new output tree. Several transformation languages which are abstractions of XML are studied in [26], and connections to extended tree-walking transducers with look-ahead are established. Various static analysis problems are considered, such as termination, emptiness, and usefulness of rules. It is also shown that ranges of the transducers are closed under intersection with *generalized DTDs* (defined by tree regular grammars).

In previous work [28] we studied type inference for XML-QL selection queries: we are given a type of the input XML document and an XML pattern with one or several variables, and the problem is to find all possible types of the variable bindings. This was motivated by declarative query languages like XML-QL [15], which separate the variable binding step from the XML construction step: the type inference in [28] studies the binding step in isolation, and, thus, is a much more restricted problem than typechecking the entire XML transformation, which we consider here. In other work [32] the type inference for XML is solved for a restricted class of transformations. In the general case, type inference does not admit a solution (see discussion in Section 4).

The paper is organized as follows. Section 2 provides background material, including DTDs, regular tree languages, and regular tree automata. Section 3 introduces and illustrates the k-pebble transducer, and shows its PTIME data complexity. Our results on typechecking of k-pebble transducers are presented in Section 4, and some extensions are discussed in Section 5. Section 6 provides brief conclusions.

2 Background

2.1 Trees

We begin with some basic definitions.

Unranked trees Assume a fixed finite alphabet Σ. Unranked trees over Σ are trees with node labels from Σ, with no predefined bound on the number of children of each node. The trees we consider are ordered (the children of a given node form a list); a forest, F, is a list of trees, and a tree, T, is a node and a forest, representing the children:

$T ::= a(F), \quad a \in \Sigma; \qquad\qquad F ::= T, F$

$T ::= a(), \quad a \in \Sigma; \qquad F ::= T$

We denote U_Σ the set of unranked trees.

Given a tree t, a subexpression occurrence of type T is called a *node*. The set of all nodes in t is denoted $nodes(t)$; the root node is $root(t)$. Given a node $T = a(F)$, with F a forest or ε, its *symbol* is $symbol(T) \stackrel{\text{def}}{=} a$, and its *children* are $children(T) \stackrel{\text{def}}{=} F$. We abbreviated a leaf node $a()$ with a when no confusion arises.

A *path expression* is a string $w \in \Sigma^*$. The result of evaluating a path expression on a tree T is a set of nodes in T, defined as follows:

$$eval(\varepsilon, T) \stackrel{\text{def}}{=} \emptyset$$

$$eval(a, T) \stackrel{\text{def}}{=} \textsf{if } a = symbol(T) \textsf{ then } \{T\} \textsf{ else } \emptyset$$

$$eval(a.w, T) \stackrel{\text{def}}{=} \textsf{if } a = symbol(T)$$
$$\textsf{then } \bigcup_{T' \in children(T)} eval(w, T') \textsf{ else } \emptyset$$

A *regular path expression* is a regular expression r over Σ. The result of evaluating a regular path expression r on a tree T is $eval(r, T) \stackrel{\text{def}}{=} \bigcup\{eval(w, T) \mid w \in lang(r)\}$. Regular path expressions are used in all XML query languages to query the input tree.

Ranked trees We only consider ranked trees that are complete binary trees. The alphabet Σ is partitioned into $\Sigma = \Sigma_0 \cup \Sigma_2$. A ranked tree over Σ is a tree in which every node labeled with $a \in \Sigma_0$ is a leaf and every node labeled with $a \in \Sigma_2$ has two children. T_Σ denotes the set of ranked trees over Σ. Similarly, $T_\Sigma(X)$ denotes the set of ranked trees with "variables in X". Formally, $T_\Sigma(X) \stackrel{\text{def}}{=} T_\Omega$, where the alphabet Ω defined by $\Omega_0 \stackrel{\text{def}}{=} \Sigma_0 \cup X$, and $\Omega_2 \stackrel{\text{def}}{=} \Sigma_2$.

We will denote leaf symbols with $a_0, b_0, \ldots \in \Sigma_0$, while symbols for internal nodes with $a_2, b_2, \ldots \in \Sigma_2$. We omit the index when it is clear from the context, i.e. by writing $a, b, c \ldots$.

We encode unranked trees into complete binary trees over alphabet $\Sigma' \stackrel{\text{def}}{=} \Sigma \cup \{-, |\}$, with $-, |$ two new symbols. Define $\Sigma_0 \stackrel{\text{def}}{=} \{|\}$, and $\Sigma_2 \stackrel{\text{def}}{=} \Sigma \cup \{-\}$. The encoding is given by $encode : U_\Sigma \to T_{\Sigma'}$ (defined recursively with $encode_f$, the encoding of a forest):

$$encode(a(F)) \stackrel{\text{def}}{=} a(encode_f(F), |)$$

$$encode(a()) \stackrel{\text{def}}{=} a(|, |)$$

$$encode_f(T, F) \stackrel{\text{def}}{=} -(encode(T), encode_f(F))$$

$$encode_f(T) \stackrel{\text{def}}{=} encode(T)$$

For example $encode(a(b, b, c(d), e)) = a(-(b, -(b, -(c(-(d, |), |), -(e, |)))), |)$, see Figure 1.

There is a one-to-one label-preserving mapping between the nodes in T and nodes in $encode(T)$. It is easy to translate path expressions, and regular path expressions, while preserving semantics under this mapping (we have chosen

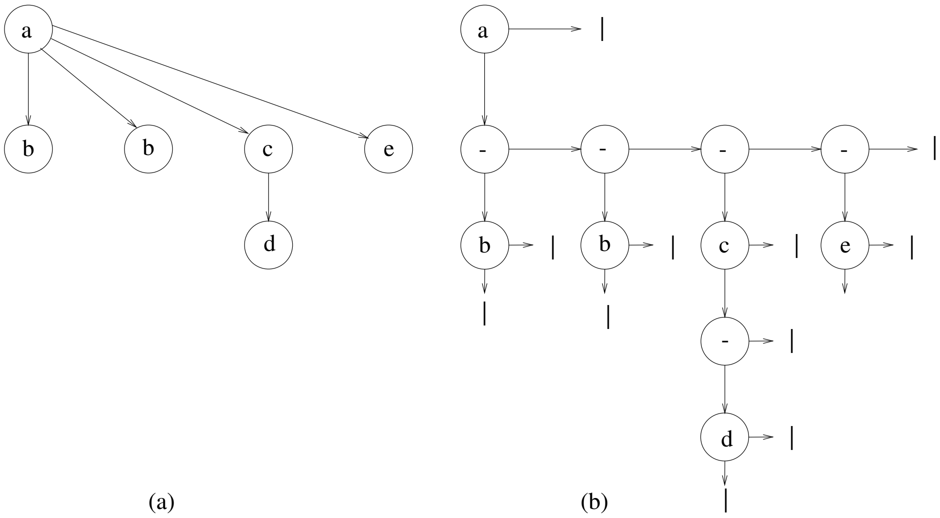

Figure 1: Encoding of an unranked tree (a) into a binary tree (b).

the binary encoding such as to make the translation possible). A (regular) path expression r over Σ is translated into a (regular) path expression $translate(r)$ over $\Sigma \cup \{-\}$ (the symbol $|$ is never used): we omit its (rather obvious) formal definition, but give two examples.

$translate(a.c.d) = a.(-)^*.c.(-)^*.d$

$translate(a.(b|(c.d))^*.e) =$

$a.(-)^*.(b.(-)^*|(c.(-)^*.d.(-)^*))^*.e$

When the translated regular expression is evaluated on the encoded binary tree we get an encoded version of the original result; more precisely:

$$eval(translate(r), encode(t)) = \{encode(t') \mid t' \in eval(r,t)\}.$$

2.2 XML (eXtensible Markup Language)

In this paper we take the simplifying assumption that XML is a syntax for unranked trees. Symbols in the alphabet Σ correspond to XML tags. For example, the unranked tree in Fig. 1 is written in XML as follows:

```
<a> <b></b> <b></b> <c><d></d></c> <e></e> </a>
```

In XML terminology, `a, b, c, ...` are *tags*. Each tag has a start- and end-tag (e.g. `<c>` and `</c>`): the text between them (tags including) is called an *element*.

Under this assumption, query languages for XML are formalisms for expressing transformations on U_Σ. Some care is needed here, however, since some of these languages (XML-QL, Lorel, StruQL) were initially defined on a graph data model (OEM [30]). Restricting these languages to trees gives us a formalism for transforming *unordered* trees, while our ranked trees are ordered. It has been shown both for XML-QL and Lorel how to cope with order, and the same can be done for UnQL and StruQL. XSLT was designed for an ordered model.

We will not describe these query languages here, but only mention a feature present in all of them: pattern matching. A *pattern* is a tree in $U_{Reg(\Sigma)}$, where $Reg(\Sigma)$ is the set of regular expressions over Σ. To illustrate, consider a pattern with three nodes, $p = [a.b]([c.(a \mid b)], [c^*.a])$, containing three regular expressions. Given an input tree $t \in U_\Sigma$, a *matching* of the pattern in the tree is given by three nodes $x_1, x_2, x_3 \in nodes(t)$, satisfying the following three conditions:

$$\begin{aligned} x_1 &\in eval(a.b, t) \\ x_2 &\in eval(c.(a \mid b), x_1) \\ x_3 &\in eval(c^*.a, x_1) \end{aligned}$$

As for the case of regular expressions, patterns over unranked trees can be translated into equivalent patterns over ranked trees. This observation is important, and justifies our choice to consider ranked trees only.

Of course, identifying XML documents with unranked trees is a simplifying assumption that ignores several aspects of XML, such as data values (`#PCDATA`), attributes and references. This limits the applicability of the results in this paper to the full XML.

We comment on these limitations next.

- **Fixed set of tags** In XML new tags can be defined at will, while our unranked trees are over a fixed, finite alphabet Σ. This is not a serious limitation. Our focus is on type inference, where "type" is defined as a DTD (see below): in this setting the set of all XML tags is also fixed in advance, hence fits our framework.

- **No data values** XML elements have *content*, which can be an arbitrary text not restricted to a fixed, finite alphabet. For example:

```
<a> <b> this is some text </b>
    <c> <d> some other text </d>
        <e> yet more text <e/>
    </c>
</a>
```

  A more accurate model of XML would allow leaves to be labeled with elements from an infinite data universe D. In the new model, queries can express joins on data values. However, as we explained in the introduction, the typechecking technique described in this paper fails for the extended model, hence this limits its applicability to queries without joins on data values.

- **XML attributes** XML has *attributes*: while these can be represented as special kinds of nodes in the unranked tree, their interest for typechecking is that DTDs can enforce, to a certain degree, the datatype of an attribute (for example can impose that an attribute has a value from a fixed, finite set): by contrast the datatype of an element is only text (`#PCDATA`). The typechecking techniques in this paper can be easily extended to deal with data types using the same technique as in [4].

- **References** XML has a limited form of reference, through attributes of types ID, IDREF, and IDREFS. We do not deal here with references or attribute constraints.

2.3 Types and DTDs

We next present DTDs and tree automata as type systems for unranked and ranked trees, and explain how the latter can be used to model the former.

DTDs In XML one can define a *Document Type Definition* (DTD) to be used for validation purposes. Formally, a DTD is an extended context-free grammar[4] with non-terminals Σ. An XML document (or, the unranked tree representing it) is valid with respect to the DTD if it is a derivation tree for the grammar. For a DTD D, we denote by $inst(D)$ the set of unranked trees representing valid XML instances of D. For example, the XML document represented by the tree of Fig. 1 is valid w.r.t to the DTD: $a := b^*.c.e;\;\; b := \varepsilon;\;\; c := d^*;\;\; d := \varepsilon;\;\; e := \varepsilon$

Tree automata A tree automaton is a device that accepts or rejects ranked trees. We only consider here binary trees, hence the alphabet Σ is $\Sigma_0 \cup \Sigma_2$. The definition below is adapted from [37].

Definition 2.1 *A* non-deterministic top-down tree automaton *is*

$$A = (\Sigma, Q, q_0, Q_F, P)$$

where Q is a finite set of states, $q_0 \in Q$ is the initial state, $Q_F \subseteq \Sigma_0 \times Q$ are the final symbol-state pairs, and $P \subseteq \Sigma_2 \times Q \times Q \times Q$.

We write

$$(a, q) \quad \rightarrow \quad (q_1, q_2) \tag{1}$$

whenever $(a, q, q_1, q_2) \in P$. Informally, the tree automaton starts in state q_0, and with a head on the root node of the input tree, labeled with symbol a. The automaton chooses two states q_1, q_2 such that $(a, q_0) \rightarrow (q_1, q_2)$, then continues the computation both on the left child (starting in state q_1) and on the right child (starting in state q_2). This can be thought of as spawning two computation branches, the first starting at the left child and the second at the right child. More branches are spawned for the subtrees and they stop when reaching a leaf. The input tree is accepted if all branches stop in an accepting state. $inst(A)$ denotes the set of trees accepted by A. A *type*, τ, is a set of the form $\tau = inst(A)$, for some A.

Note that the automaton exhibits two kinds of nondeterminism:

[4]In an extended CFG, the right-hand sides of productions are regular expressions over the terminals and non-terminals.

- *Or*-nondeterminism: for a given pair (a, q) we may have multiple transitions $(a, q) \to (q_1, q_2)$;
- *And*-nondeterminism: at every node, both the left computation and the right computation must accept. This leads to a form of *alternation* [10].

An automaton such as the above is referred to in the literature as a top-down or root-to-frontier tree automaton. Bottom-up tree automata (deterministic or non-deterministic), that start from the leaves and work their way up to the root, are known to be equivalent to the non-deterministic top-down automata. Deterministic top-down automata are strictly weaker. For a comprehensive discussion see [34].

DTDs and tree automata What is the relationship between tree automata and DTDs? DTDs describe sets of valid XML documents, hence unranked trees. Recall however that each such unranked tree t has a unique representation $encode(t)$ as a ranked binary tree. Given a DTD D it is straightforward to construct a tree automaton A such that $inst(A) = \{encode(t) \mid t \in inst(D)\}$. The converse is false: tree automata are strictly stronger than DTDs. For a simple example consider the tree $t = a(b(c), b(d))$ over the unranked alphabet $\Sigma = \{a, b, c, d\}$. The set $T = \{t\} \subset U_\Sigma$ cannot be described by a DTD, because the "type" of the first b differs from that of the second b. However the set $\{encode(t)\} \subseteq T_{\Sigma'}$ is regular, since every singleton set is regular. A very natural extension of DTDs with decoupled tags (also called specialized DTDs) turn out to be precisely equivalent to tree automata [8, 32]. From now on, whenever we use the term *type* we mean regular tree language (or its specification as a tree automaton).

Ranked vs. unranked trees Since both queries and types on unranked trees can be encoded into complete, binary trees, we shall consider only the latter in the remainder of the paper. Hence, from now on, the alphabet Σ is assumed to be partitioned into $\Sigma_0 \cup \Sigma_2$. All results carry over to unranked trees via the encoding.

Top-down tree automata with silent transitions For technical reasons we need to consider later a certain generalization of top-down automata. A top-down tree automaton with *silent* transitions has, in addition to transitions (), transitions of the following form:

$$(a, q) \quad \to \quad q' \tag{2}$$

Such a move causes a state change from q to q' while keeping the head on the current node. We define their semantics formally. Given tree t, a *configuration* of A on t is $\delta = [q, x]$, where $q \in Q$, $x \in nodes(t)$; let Δ denote the set of all configurations. The relation $\to$ on $\Delta \times \Delta^*$ is defined as follows. Let $\delta = [q, x]$. For every non-silent transition (2.3) applicable to δ we define $\delta \to \delta_1\delta_2$ where $\delta_1 = [q_1, x_1], \delta_2 = [q_2, x_2]$, and x_1, x_2 are the two children of x. For every silent transition (2) applicable to δ we define $\delta \to \delta'$, where $\delta' = [q', x]$. Finally, whenever $(a, q) \in Q_F$, where a is x's label, then we also define $\delta \to \varepsilon$. Next, define $\Rightarrow$ on $\Delta^* \times \Delta^*$ to be the congruential closure of $\to$: $\delta \to w'$ implies $\delta \Rightarrow w'$, and $w_1 \Rightarrow w_2$ implies $w_1 w \Rightarrow w_2 w$ and $w w_1 \Rightarrow w w_2$. Let $\overset{*}{\Rightarrow}$ be its transitive and reflexive closure. A tree t is accepted by A if $\delta_0 \overset{*}{\Rightarrow} \varepsilon$, where $\delta_0 = [q_0, \mathit{root}(t)]$ is the initial configuration. For automata without silent transitions, this definition of acceptance coincides with the informal one above.

For every top-down automaton $A = (\Sigma, Q, q_0, Q_F, P)$ with silent transitions there exists an equivalent top-down automaton A_0 without silent transitions. To see that, denote $q \to_a q'$ whenever a transition $(a, q) \to q'$ is in P, and let $q \overset{*}{\to}_a q'$ be the transitive and reflexive closure. Define P' to consists of all transitions of the form $(a, q) \to (q_1, q_2)$ whenever $q \overset{*}{\to}_a q'$ and $(a, q') \to (q_1, q_2)$ is in P, and $Q'_F = \{(a, q) \mid q \overset{*}{\to}_a q', (a, q') \in Q_F\}$. Then $A_0 \overset{\text{def}}{=} (\Sigma, Q, q_0, Q_F, P_0)$ is equivalent to A (proof omitted) and has no silent transitions.

3 Tree Transformations

We use k-pebble transducers to model tree transformations. They generalize classical top-down tree transducers, which are in turn extensions of the tree automata described above.

3.1 The k-pebble tree transducer

The k-pebble transducer uses up to k pebbles to mark certain nodes in the tree. Transitions are determined by the current node symbol, the current state, and by the existence/absence of the various pebbles on the node. The pebbles are ordered and numbered $1, 2, \ldots, k$. The machine can place pebbles on the root, move them around, and remove them. In order to limit the power of the transducer (so that typechecking can be performed) the use of pebbles is restricted by a stack discipline: pebbles are placed on the tree in order and removed in reverse order, and only the highest-numbered pebble present on the tree can be moved. More precisely, pebble i, $(i > 1)$ can only be placed on the tree if pebbles $1, \ldots, i-1$ are present, and pebble $i-1$ cannot be moved as long as pebble i is still on the tree. The stack discipline is essential for the results in the paper.

The transducer works as follows. The computation starts by placing pebble 1 on the root. At each point, pebbles $1, 2, \ldots, i$ are on the tree, for some $i \in \{1, \ldots, k\}$; pebble i is called the *current pebble*, and the node on which it sits is the *current node*. The current pebble serves as the head of the machine. The machine decides which *transition* to make, based on the following information: the symbol under the current pebble, a, the presence/absence of the other $i-1$ pebbles on the current node, denoted by a vector $\bar{b} \in \{0,1\}^{i-1}$, and the current state, q. There are two kinds of transitions: *move* and *output* transitions. *Move* transitions are of four kinds:

- *down-left, down-right, up-left, up-right:* the current pebble is moved from the current node in the given direction (one edge only). If a move in the specified direction is not possible, the transition does not apply.

- *stay:* the current pebble does not move, only the state changes.[5]
- *place-new-pebble:* pebble $i+1$ is placed on the root (becoming the current pebble).
- *pick-current-pebble:* the current pebble $i > 1$ is removed from the tree and pebble $i-1$ becomes the current pebble.

After each move transition the machine enters a new state, as specified by the transition. The moves *up-left* and *up-right* allow us to move up and test whether the current node is the lef child or right child: a pair of two transitions, one that moves *up-left* and enters state q_1 and another one that moves *up-right* and enters state q_2, have the same effect as moving the pebble to the parent and checking wheter we are coming from the right (in which case we enter state q_1) or from the left (in which case we enter state q_2).

An *output* transition emits some output node with a symbol in $a' \in \Sigma'$ and does not move the input head. There are two kinds of output transitions. In a *binary* output $a' \in \Sigma'_2$ and the machine spawns two computation branches computing the left and right child respectively. Both branches inherit the positions of all pebbles on the input, and do not communicate; each moves the k pebbles independently of the other. In a *nullary* output $a' \in \Sigma'_0$, the node being output is a leaf and that branch of computation halts.

Looking at the global picture, the machine starts with a single computation branch and no output nodes. After a while it has constructed some top fragment of the output tree, and several computation branches continue to compute the remaining output subtrees. The entire computation terminates when all computation branches terminate.

We now define k-pebble transducers formally:

Definition 3.1 *A k-pebble tree transducer is $T = (\Sigma, \Sigma', Q, q_0, P)$, where:*

- *Σ, Σ' are the ranked input and output alphabets.*
- *Q is a finite set of states and is partitioned into:*

$$Q = Q_1 \cup \ldots \cup Q_k$$

  Each state in Q_i "controls" pebble i: we denote elements in Q_i with a superscript, as in $q^{(i)}$

- *$q_0 \in Q_1$ is the initial state*
- *P is a finite set of transitions of the forms:*

[5] Observe that this can be simulated by moves of the above form, and is used here only for convenience.

$(a, \bar{b}, q_1^{(i)}) \to (q_2^{(i)}, \textit{stay})$
$(a, \bar{b}, q_1^{(i)}) \to (q_2^{(i)}, \textit{down-left})$
$(a, \bar{b}, q_1^{(i)}) \to (q_2^{(i)}, \textit{down-right})$
$(a, \bar{b}, q_1^{(i)}) \to (q_2^{(i)}, \textit{up-left})$
$(a, \bar{b}, q_1^{(i)}) \to (q_2^{(i)}, \textit{up-right})$
$(a, \bar{b}, q_1^{(i)}) \to (q_2^{(i+1)}, \textit{place-new-pebble})$
$(a, \bar{b}, q_1^{(i)}) \to (q_2^{(i-1)}, \textit{pick-current-pebble})$
$(a, \bar{b}, q^{(i)}) \to (a'_0, \textit{output0})$
$(a, \bar{b}, q^{(i)}) \to (a'_2(q_1^{(i)}, q_2^{(i)}), \textit{output2})$
where $a \in \Sigma, a'_0, a'_2 \in \Sigma', b \in \{0,1\}^i, q^{(i)}, q_1^{(i)}, q_2^{(i)} \in Q_i,\ i = 1, \ldots, k.$

In general T can be nondeterministic and the transformation it defines is a binary relation in $T_\Sigma \times T_{\Sigma'}$. We give the formal definition of the transformation next. For a tree $t \in T_\Sigma$, a *configuration* of T on t is a triple $\gamma = (i, q^{(i)}, \bar{x})$, where $1 \leq i \leq k$, $q^{(i)} \in Q_i$, and $\bar{x} \in (\textit{nodes}(t))^i$. Let Γ be the set of all configurations for transducer T and input tree t. The *initial* configuration is: $\gamma_0 \stackrel{\text{def}}{=} (1, q_0, \textit{root}(t))$, where q_0 is the initial state in T.

A partial computation of the transducer T on input t is captured by a tree in $T_{\Sigma'}(\Gamma)$. Nodes labeled with $a \in \Sigma'$ represent the output generated so far, while leaves labeled with $\gamma \in \Gamma$ represent the on-going branches of the computation. The transducer T defines a rewriting $\Rightarrow$ on $T_{\Sigma'}(\Gamma)$, called the T-rewriting, describing one step of the computation: intuitively $t'_1 \Rightarrow t'_2$ if t'_2 can be obtained from t'_1 by applying one transition step to one of the on-going branches of computations in t'_1. The T-rewriting is defined in terms of a simpler rewriting $\to$, from Γ to $T_{\Sigma'}(\Gamma)$, with the meaning that $\gamma \to t'$ if t' can be obtained from γ in one transition step, and we define $\to$ first. Recall that P is the set of transitions in the transducer T. Given $\gamma = (i, q^{(i)}, \bar{x})$ and a transition $(a, \bar{b}, q_1^{(i')}) \to \ldots$ in P, we say that the transition *applies* to γ if the following four conditions hold: (1) $i = i'$, (2) $\textit{symbol}(x_i) = a$, (3) $\beta_1\beta_2\ldots\beta_{i-1} = \bar{b}$ where $\beta_j = 1$ if $x_j = x_i$, and $\beta_j = 0$ if $x_j \neq x_i$, for $j = 1, \ldots, i-1$, and (4) $q^{(i)} = q_1^{(i)}$. Now we can finally define $\to$. This is done separately for move transitions and for output transitions. For each move transition $(a, \bar{b}, q_1^{(i)}) \to (q_2^{(j)}, d)$ in P that applies to γ, we define $\gamma \to \gamma'$, where $\gamma' = (j, q_2^{(j)}, \bar{y})$, such that the following conditions hold: if $d = \textit{stay}$, then $i = j$ and $\bar{x} = \bar{y}$; if $d = \textit{down-left}$, then $i = j$ and $\bar{x} = \bar{z}x_i$, $\bar{y} = \bar{z}x'_i$, and x'_i is the left child of x_i; if $d = \textit{up-left}$, then $i = j$, $\bar{x} = \bar{z}x_i$, $\bar{y} = \bar{z}x'_i$, and x_i is the left child of x'_i; similarly for $d = \textit{down-right}$ and $d = \textit{up-right}$; if $d = \textit{place-new-pebble}$ then $j = i + 1$ and $\bar{y} = \bar{x}\textit{root}(t)$; if $d = \textit{pick-current-pebble}$ then $j = i - 1$ and $\bar{x} = \bar{y}x_i$, for some node x_i. For each output transition $(a, \bar{b}, q^{(i)}) \to (a'_0, \textit{output0})$ in P, that applies to γ, we define $\gamma \to a'_0$. For each output transition $(a, \bar{b}, q^{(i)}) \to (a'_2(q_1^{(i)}, q_2^{(i)}), \textit{output2})$ that applies to γ, we define $\gamma \to a'_2(\gamma_1, \gamma_2)$, where $\gamma_1 = (i, q_1^{(i)}, \bar{x})$, $\gamma_2 = (i, q_2^{(i)}, \bar{x})$.

Now we define $\Rightarrow$ to be the congruence closure of $\to$: (1) if $\gamma \to t'$ then $\gamma \Rightarrow t'$, and (2) if $t'_1 \Rightarrow t'_2$, then:

$$\begin{array}{rcl} a'_2(t'_1, t') & \Rightarrow & a'_2(t'_2, t') \\ a'_2(t', t'_1) & \Rightarrow & a'_2(t', t'_2) \end{array}$$

for all $a'_2 \in \Sigma'_2$ and $t' \in T_{\Sigma'}(\Gamma)$. Let $\stackrel{*}{\Rightarrow}$ be the transitive closure of $\Rightarrow$.

Finally, the *transformation* defined by the k-pebble tree transducer T is defined as follows. For every $t \in T_\Sigma$, $T(t) \stackrel{\text{def}}{=} \{t' \mid t' \in T_{\Sigma'}, \gamma_0 \stackrel{*}{\Rightarrow} t'\}$, where γ_0 is the initial configuration of T on t. Thus, the transducer defines the binary relation $\{(t, t') \mid t' \in T(t)\}$ on $T_\Sigma \times T_{\Sigma'}$.

Relationship to top-down and bottom-up transducers Top-down and bottom-up tree transducers have been considered previously [34] (the terms frontier-to-root and root-to-frontier are used there). We recall here the definition of a top-down transducer.

Definition 3.2 *A top-down tree transducer is $T = (\Sigma, \Sigma', Q, q_0, P)$ where Q is the set of states, $q_0 \in Q$ is the initial state, and P are the transitions of the following form:*

- *$(a, q) \to t'$, where $a \in \Sigma_2, q \in Q, t' \in T_{\Sigma'}(\{\xi_1, \xi_2\} \times Q)$.*
- *$(a, q) \to t'$, where $a \in \Sigma_0, q \in Q, t' \in T_{\Sigma'}$.*

Here ξ_1, ξ_2 are two new symbols.

Transitions are as in tree automata, except that now the machine also constructs at each transition part of the output tree. We describe T's computation informally next. Given an input tree the transducer starts at the root in state q_0. In general it is on some node x labeled a, and in some state q. T chooses some transition of the form $(a, q) \to t'$. If $a \in \Sigma_2$ (i.e. T is at an internal node) then it outputs the tree $t' \in T_{\Sigma'}(\{\xi_1, \xi_2\} \times Q)$. This tree represents a fragment of the transducer's output. The fragment t' may have leaves labeled with elements of the form (ξ_i, q'), $i = 1, 2$, $q' \in Q$. The leaves will be replaced by other output trees. For that, the transducer spawns a new computation branch for each leaf, which will compute the tree to replace that leaf. If the leaf was labeled (ξ_1, q'), then the corresponding branch starts on the left child of x and in state q'; if the leaf was labeled (ξ_2, q'), then the branch starts on the right child of x and in state q'. When a leaf is reached in the input tree, then a transition of the form $(a, q) \to t'$ with $t' \in T_{\Sigma'}$ is selected: the transducer outputs t' and halts that computation branch. The computation halts when all branches halt, and the current output tree is defined to be T's output. Since T is nondeterministic, it may have several (but finitely many) outputs, for a given input tree.

It is easy to see that every top-down transducer can be expressed as a 1-pebble transducer. The relationship to bottom-up transducers (which work in reverse to top-down ones, see [34] for a definition) is open and related to an open problem on tree-walk automata. A *tree-walk* automaton [16] is a 1-pebble transducer without output transitions and with an accepting state. A tree-walk automaton starts with the pebble on the root, and walks up and down the tree. If it ever enters the accepting state, then it accepts the tree. (Tree-walk automata are equivalent to 1-pebble automata, Section 4, without branching instructions.) The tree language accepted by a tree-walk automaton is regular (this also follows from Theorem 4.7 below), but it is an open problem whether tree-walk automata can express all regular tree languages.

Given a bottom-up transducer T, we may attempt to simulated it with a 1-pebble transducer, by running T in reverse. Starting from the root, at each node x we guess in which state T would end up at x, output the corresponding symbol a', then branch to compute the two children of a'. The problem is that, during its bottom-up computation, T may choose to ignore the entire output it produced for a subtree of x: hower, it still uses the terminating state of the computation on that subtree. In the top-down simulation, we need to guess that state, then check on the subtree that the guess was correct. This test is like checking that the subtree is in a certain regular language: when doing so, we cannot branch (because whenever we branch, we must output something), i.e. we are restricted to the behavior of a tree-walk automaton. Thus, 1-pebble transducers can simulate all bottom-up transducers iff tree-walk automata can express all regular tree languages. Similarly, k-pebble transducers can simulate all bottom-up transducers iff k-pebble automata without branching (Section 4) can express all regular languages: this is open too.

3.2 XML transformation languages and k-pebble transducers

We state, without proof, that all transformations over unranked trees over a given finite alphabet expressed in existing XML query languages (XML-QL, Lorel, StruQL, UnQL, and a fragment of XSLT) can be expressed as k-pebble transducers. Note that, in particular, this does not extend to queries with joins on data values, since these require an infinite alphabet. In lieu of a proof, we illustrate the power of k-pebble transducers by means of several examples.

Example 3.3 A 1-pebble transducer that simply copies the input tree to the output is $T = (\Sigma, \Sigma, \{q, q_1, q_2\}, q, P)$, where P is:

$(a_2, q) \rightarrow (a_2(q_1, q_2), \mathit{output2})$ $\quad$ $for\ all\ a_2 \in \Sigma_2$

$(a_2, q_1) \rightarrow (q, \mathit{down\text{-}left})$

$(a_2, q_2) \rightarrow (q, \mathit{down\text{-}right})$

$(a_0, q) \rightarrow (a_0, \mathit{output0})$ $\quad$ $for\ all\ a_0 \in \Sigma_0$

Example 3.4 We show here a useful "subroutine" that moves pebble 1 from a given node to the next node in pre-order. Start in state q_1, stop in state q_2; if the tree is exhausted, then enter state q_f. Assume that $r \in \Sigma_2$ is the root symbol:

$(a_2, q_1) \rightarrow (q_2, \mathit{down\text{-}left})$ $\quad$ // done

(a_0, q_1)	$\to (q_3, stay)$	// we are on a leaf: prepare to move up
(a, q_3)	$\to (q_3, \textit{up-right})$	// move up as long as coming from the right
(a, q_3)	$\to (q_4, \textit{up-left})$	// one move up left
(a, q_4)	$\to (q_2, \textit{down-right})$	// one move down right and done
(r, q_3)	$\to (q_f, stay)$	// tree exhausted

Here $a_0 \in \Sigma_0$, $a_2 \in \Sigma_2$ and $a \in \Sigma_0 \cup \Sigma_2 - \{r\}$.

Example 3.5 This example illustrates why k-pebble transducers can express pattern matching. As argued in Section 2.2, patterns are the most essential common denominator of existing XML query languages. Recall that a pattern is a tree labeled with regular expressions. We illustrate with $p = r_1(r_2, r_3(r_4, r_5))$, where $r_1, \ldots, r_5$ are five regular expressions. We can find all matches of this pattern in a input tree, using $k = 6$ pebbles (in general, k is the number of nodes in the pattern, plus one). The transducer enumerates, in lexicographic order, all five-tuples of nodes $(x_1, x_2, x_3, x_4, x_5)$ in the input tree, using five pebbles advanced with the pre-order traversal "subroutine" from the previous example. Then, for each 5-tuple, it uses the sixth pebble to check the following five conditions: $x_1 \in eval(r_1, t)$, $x_2 \in eval(r_2, x_1)$, $x_3 \in eval(r_3, x_1)$, $x_4 \in eval(r_4, x_3)$, $x_5 \in eval(r_5, x_3)$. Each condition has the form $x_i \in eval(r_j, x_k)$ (the first condition involves the root node t, which we denote with x_0), and we can check it by locating first the node x_i with the sixth pebble (using a pre-order tree traversal), then moving up the tree until we reach x_k, and checking the regular expression r_j, in reverse, along the way.

We illustrate with $p = [a.b^*.c]([(a|f).g], [b]([c^*.d], [d^*.c]))$, a pattern with five regular expressions. We can find all matches of this pattern in a input tree, using $k = 6$ pebbles (in general, k is the number of nodes in the pattern, plus one). The transducer enumerates, in lexicographic order, all five-tuples of nodes $(x_1, x_2, x_3, x_4, x_5)$ in the input tree, using five pebbles which are advanced using the pre-order traversal "subroutine" in the previous example. Then, for each 5-tuple, it uses the sixth pebble to check the following five conditions: (1) the path from the root to x_1 matches $a.b^*.c$ (the first regular expression), (2) x_1 is an ancestor of x_2 and the path from x_1 to x_2 matches $(a|f).g$, ..., (5) x_3 is an ancestor of x_5 and the path between them matches $d^*.c$. Each condition involves two nodes x_i, x_j. To check that x_i is an ancestor of x_j and the path between them matches a given regular expression, the transducer uses the sixth pebble to search for x_j (using a pre-order tree traversal), from there it moves the sixth pebble up the tree until it reaches x_i, checking the regular expression in reverse.

Example 3.6 This example illustrates that the size of the output tree may be exponentially larger than that of the input tree. The transducer maps a tree t

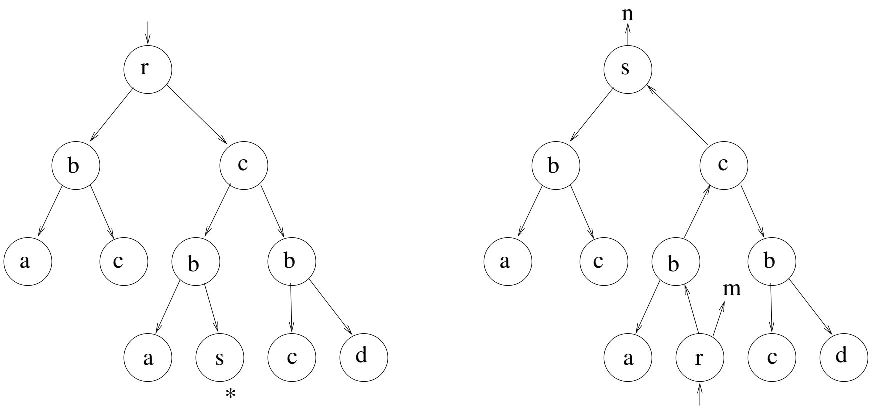

Figure 2: A complex tree transformation: rotating around leaf s. Two new nodes need to be added, labeled m and n respectively. The order of the children in the output tree is read counterclockwise.

into $f(t)$, where f is defined recursively:

$$\begin{array}{rcll} f(a(t_1, t_2)) & = & x(a(f(t_1), f(t_2)), a(f(t_1), f(t_2))) & a \in \Sigma_2 \\ f(a()) & = & x(a(), a()) & a \in \Sigma_0 \end{array}$$

Here $T = (\Sigma, \Sigma \cup \{x\}, \{q_1, q_2, q_3\}, q_1, P)$, where P consists of the following:

$$\begin{array}{ll} (a, q_1) \to (x(q_2, q_2), \mathit{output2}) & a \in \Sigma \\ (a, q_2) \to (a, \mathit{output0}) & a \in \Sigma_0 \\ (a, q_2) \to (a(q_3, q_4), \mathit{output2}) & a \in \Sigma_2 \\ (a, q_3) \to (q_1, \mathit{down\text{-}left}) & a \in \Sigma_2 \\ (a, q_4) \to (q_1, \mathit{down\text{-}right}) & a \in \Sigma_2 \end{array}$$

Example 3.7 This 1-pebble transducer performs a complex rotation of the input tree. Assume that each symbol $a_0 \in \Sigma_0$ has a corresponding symbol $a_2 \in \Sigma_2$, and let s be some symbol (i.e $s_0 \in \Sigma_0, s_2 \in \Sigma_2$). The following transducer finds the first node labeled s_0, and declares it to be the new root. See Fig. 2 for an illustration. The transducer proceeds as follows. First it uses a depth-first traversal to place pebble 1 on the leaf labeled s, and enters state q: these transitions are omitted. We show here the other transitions. We may assume to start in state q, on the leaf labeled s_0. The subscripts in the states $q_{up}, q_{left}, q_{right}$ indicate which way we reached the current node, while the subscripts in $q'_{up}, q'_{left}, q'_{right}$ indicate which way we should go next. Here r is the symbol of the root.

$$\begin{array}{lll} (s_0, q) & \to (r_2(q', q'_{up}), \mathit{output2}) \text{ new root} & \\ (s_0, q') & \to (m_0, \mathit{output0}) & \\ & \text{produces additional node } m & \\ (a, q'_{up}) & \to (q_{left}, \mathit{up\text{-}left}) & a \neq r \end{array}$$

$$
\begin{array}{lll}
(a, q'_{up}) & \to (q_{right}, \textit{up-right}) & a \neq r \\
(r, q'_{up}) & \to (n, \textit{output0}) & \\
 & \text{produces additional node } n & \\
(a_2, q_{left}) & \to (a_2(q'_{right}, q'_{up}), \textit{output2}) & \\
(a_2, q_{right}) & \to (a_2(q'_{up}, q'_{left}), \textit{output2}) & \\
(a_2, q_{up}) & \to (a_2(q'_{left}, q'_{right}), \textit{output2}) & \\
(a_0, q_{up}) & \to (a_0, \textit{output0}) & \\
(a_2, q'_{left}) & \to (q_{up}, \textit{down-left}) & \\
(a_2, q'_{right}) & \to (q_{up}, \textit{down-right}) &
\end{array}
$$

In particular, this example illustrates that a 1-pebble transducer can reverse a string (assuming the string is encoded as a right-linear binary tree).

3.3 Complexity and Expressiveness

Besides their role in typechecking (Section 4), k-pebbles transducers are of independent interest for expressing tree transformations. We show here that their data complexity is in PTIME. In the case of a deterministic transducer T this means that there exists a PTIME algorithm that "computes" $T(t)$ for an input tree t. Note that the generated output $T(t)$ can have size exponential in t (Example 3.6). Still, the algorithm will produce a polynomial-size encoding of $T(t)$, as a DAG. In the case of non-deterministic k-pebbles transducers we need to be more careful what the PTIME data complexity means. In particular T can produce an infinite set of outputs for a given t: $T(t) = \{t_1, t_2, \ldots\}$. We show:

Proposition 3.8 *Let T be a fixed k-pebble transducer. Then for each input tree t, (1) the set $T(t)$ is a regular tree language, and (2) one can construct in* PTIME *(in the size of t) a tree automaton A_t that accepts the language $T(t)$.*

Proof: (Sketch) For the sequel we fix some k-pebble tree transducer $T = (\Sigma, \Sigma', Q, q_0, P)$ and some input tree t. Recall that $T(t)$ is the set of output trees. We will construct a top-down tree automaton A_t, with *silent transitions*, which accepts precisely the language $T(t)$. A silent transition has the form $(a, q) \to q'$, with the meaning that the automaton may change its state from q to q' without advancing the head if the underlying symbol is a. Obviously such transitions do not increase the automaton's expressive power. $A_t \stackrel{\text{def}}{=} (\Sigma', \Gamma \cup \{q_f\}, \gamma_0, \Sigma' \times \{q_f\}, P')$, where Γ is the set of configurations of T on t, q_f is a new symbol denoting the unique final state, and γ_0 is the initial T-configuration. We define the productions P' next. Recall the rewriting $\gamma \to t'$, where $\gamma \in \Gamma$ and where $t' \in T_{\Sigma'}(\Gamma)$ had one of the following three forms: (1) t' is a T-configuration γ', (2) t' is an output symbol $a \in \Sigma'$, (3) t' is a tree with three nodes $a'(\gamma_1, \gamma_2)$, $a' \in \Sigma', \gamma_1, \gamma_2 \in \Gamma$. Define P' to consists of the following:

$$
\begin{array}{ll}
(a', \gamma) \to \gamma' & \text{(for every rewriting } \gamma \to \gamma'\text{)} \\
(a', \gamma) \to q_f & \text{(for every rewriting } \gamma \to a'\text{)}
\end{array}
$$

$(a', \gamma) \rightarrow (\gamma_1, \gamma_2)$ (for every rewriting $\gamma \rightarrow a'(\gamma_1, \gamma_2)$)

The first two are silent transitions, the last is a regular transition in the top-down automaton A_t. If t has n nodes, then there are $O(n^k)$ configurations, hence A_t can be computed in PTIME in the size of t.

It is easy to check that A_t accepts some tree t' iff $t' \in T(t)$. We have to prove that A_t accepts some tree t' iff $t' \in T(t)$. We start with "if". Given $t' \in T(t)$ we have:

$$t'_0 = \gamma_{0,T,t} \Rightarrow t'_1 \Rightarrow \ldots \Rightarrow t'_n = t'$$

where $t'_i \in T_{\Sigma'}(\Gamma)$, for $i = 0, \ldots, n$. Let Δ be the set of A_t-configurations on t'. For each t'_i, $i = 1, \ldots, n$, define $w_i \in \Delta^*$ to be the following sequence w_i of A_t-configurations. Let $x_1, \ldots, x_p$ be all the leaf nodes in $t'_i \in T_{\Sigma'}(\Gamma)$ which are labeled with elements from Γ, and let $\gamma_1, \ldots, \gamma_p$ be their labels. Notice that $x_1, x_2, \ldots$ are also nodes in t', since the sequence $t'_1, t'_2, \ldots, t'_n$ is an increasing sequence of trees. Then define $w_i \stackrel{\text{def}}{=} (\gamma_1, x_1) \ldots (\gamma_p, x_p)$. Notice that $w_0 = \delta_0$ (the initial A_t-configuration) and $w_n = \varepsilon$. The following holds:

$$w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n$$

Indeed, each rewriting $t'_{i-1} \Rightarrow t'_i$ determines a rewriting $w_{i-1} \Rightarrow w_i$ (this can be proven by analyzing the three cases which determined the rewriting $t'_{i-1} \Rightarrow t'_i$; details omitted). This proves that A_t accepts t'.

Conversely, let t' be accepted by A_t. By definition, there exists a rewriting:

$$\delta_{0,A,t} = w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n = \varepsilon$$

Consider some $w_i = (\gamma_1, x_1) \ldots (\gamma_p, x_p)$, for $i = 0, n$. Define $t'_i \in T_{\Sigma'}(\Gamma)$ to be obtained as follows. For each $j = 1, p$, remove from t' the subtree rooted at node x_j; the node x_j becomes a leaf, which we label with γ_j. Obviously $w_0 = \gamma_{0,T,t}$ (the initial T-configuration) and $w_n = t'$ (since nothing is removed from t'). It is easy to check the following T-rewritings:

$$t'_0 \Rightarrow t'_1 \Rightarrow \ldots \Rightarrow t'_n$$

□

Proposition 3.8 can be used to effectively answer several questions about $T(t)$. First, consider the problem of constructing $T(t)$ given a deterministic transducer T, and a tree t. This can be answered in PTIME in the size of t: indeed, A_t serves as a DAG encoding of the result $T(t)$. If required, $T(t)$ can be actually computed from A_t in time polynomial in the size of both t and $T(t)$.

Next, consider the decision problem: given t, t', is $t' \in T(t)$? This obviously can be answered in PTIME in the size of t and t', by "running" the automaton A_t on t'. Finally, we can enumerate $T(t) = \{t_1, t_2, \ldots\}$: this corresponds to enumerating all trees generated by a regular tree grammar, and can be done in amortized PTIME cost in the size of the input and the output.

It easily follows from Proposition 4.6 and Theorem 4.7 below that k-pebble transducers cannot express all PTIME transformations. For example, transformations that involve cardinality checking (e.g. "output 'a' if the input tree is a right linear tree labeled with $b^n.c^n$) are not expressible by k-pebble transducers. The exact characterization of their expressive power remains open.

4 Typechecking k-pebble Transducers

Now that we have presented our types and transformation language, we are ready to present the solution for the typechecking problem. We first recall the typechecking and type inference problems, and the connection between them.

4.1 Typechecking and type inference

Recall that a type is $\tau = inst(A)$ for some tree automaton A. We denote $T(\tau) \stackrel{\text{def}}{=} \bigcup_{t \in \tau} T(t)$.

Definition 4.1 *(Typechecking) A k-pebble tree transducer T* typechecks *with respect to an input type τ_1 and an output type τ_2, if $T(\tau_1) \subseteq \tau_2$.*

The problem of verifying if T typechecks with respect to types τ_1, τ_2 is called the *typechecking problem.*

The *type inference problem* is closely related to typechecking. Given a k-pebble transducer T and an input type τ_1, the type inference problem is to construct, if possible, a type τ_2' such that $T(\tau_1) = \tau_2'$. Note that a solution to the type inference problem immediately yields a solution to the typechecking problem. To verify that T typechecks with respect to input τ_1 and output τ_2, first infer $\tau_2' = T(\tau_1)$, then check that $\tau_2' \subseteq \tau_2$, using the fact that inclusion of regular tree languages is decidable. Unfortunately, as illustrated by Example 4.2, type inference is not possible even for very simple types and transformations.

Example 4.2 Let $Q1$ be the following XML-QL query:

```
<result> WHERE <root> <a> $X </a> <a> $Y </a>
                </root>
         CONSTRUCT <b/>
</result>
```

Consider the input DTD τ_1 given by $root := a^*$. Thus, the input XML documents are of the form:

```
<root><a/> . . . <a/> </root>
```

with n occurrences of `<a/>`, for some $n \geq 0$ (in XML `<a/>` is an abbreviation for `<a> </a>`). On such a document, query $Q1$ binds each of the two variables `$X` and `$Y` to some `<a/>` element, and for each binding emits a `<b/>` output element. Hence, the query's outputs have the form:

```
<result> <b/> <b/> . . . <b/> </result>
```

with n^2 occurrences of `<b>`. In short, the query maps an input a^n to the output b^{n^2}. Clearly the set τ of all results is not described by a DTD, since DTDs can describe only regular languages. In practice one often settles for a DTD approximating τ, like $result := b^*$. This is definitely not the best approximation: in fact no best approximation for non-regular tree languages τ exists, because for any regular set $\tau' \supset \tau$ and for any tree $t \in \tau' - \tau$, the set $\tau' - \{t\}$ is still regular and a better approximation than τ'.

Example 4.3 Let $Q2$ be the following XSLT query:

```
<xsl:template match="root">
  <result>
     <b/>
     <xsl:apply-patterns/>
     <b/>
     <xsl:apply-patterns/>
     <b/>
     <xsl:apply-patterns/>
  </result>
</xsl:template>
<xsl:template match="a">
     <a/>
</xsl:template>
```

Assuming the same DTD for the input XML documents as in the previous example, this query writes a b element, then copies the entire input, and repeats this three times. Hence, it maps an input a^n to the output $ba^n ba^n ba^n$. Again, this set cannot be described by a DTD.

Given the failure of type inference, one might consider several alternatives, such as:

- Use an upper approximation of the output $T(\tau_1)$, like b^* above. This results in some correct transformations being rejected by the typechecker.
- Use a stronger type system to express more sets of the form $T(\tau_1)$: we need to exercise care, because for too powerful type systems the test $T(\tau_1) \subseteq \tau_2$ may become undecidable. This approach is pursued in [32], using target types called *specialized context-free DTDs*, consisting of extended DTDs that specify the allowed children of a node as a context-free language. It is shown in [32] that type inference can be performed for simple XML-QL selection queries. As a side effect, this provides a typechecking test for this limited class of queries, since inclusion of context-free languages into regular languages is decidable. It fails, however, for XML-QL queries with more complex `CONSTRUCT` clauses. It remains open whether it can be extended using even stronger types.

The failure of type inference has a surprising twist: *inverse* type inference turns out to be possible. Returning to Example 4.2, consider query $Q1$ and assume the output DTD to be $(b.b)^*$: i.e., we require to have an even number of b's. It is easy to see that $(a.a)^*$ is the DTD describing precisely the input documents mapped by $Q1$ into output documents with an even number of b's.

More precisely, the *inverse type inference problem* is the following: given a k-pebble transducer T and an output type τ, construct a type τ^{-1} such that $\tau^{-1} = \{t \mid T(t) \subseteq \tau\}$: the type τ^{-1} is called the *inverse type* for T and τ. The solution to the inverse type inference problem immediately yields a solution to the typechecking problem: to typecheck T with respect to input τ_1 and output τ_2, first compute the inverse type τ_2^{-1} for T and τ_2, then check that $\tau_1 \subseteq \tau_2^{-1}$. This establishes the main result of the paper:

Theorem 4.4 *It is decidable, given a k-pebble transducer T, an input type τ_1, and an output type τ_2, whether T typechecks with respect to τ_1 and τ_2.*

The remainder of the section is dedicated to solving the inverse type inference problem, thus establishing Theorem 4.4. We do this in three stages:

Step 1 define an acceptor variant of the transducer, called *k-pebble automaton*;

Step 2 show that for each k-pebble transducer T and type τ, the complement of $\{t \mid T(t) \subseteq \tau\}$ is recognized by some k-pebble automaton; and,

Step 3 prove that every k-pebble automaton recognizes a regular tree language.

We develop the three steps next.

Step 1. The k-pebble automaton works similarly to the k-pebble transducer, but produces no output: its function is simply to accept or reject its input. Formally:

Definition 4.5 *A k-pebble tree automaton is a 4-tuple (Σ, Q, q_0, P), where:*

- *Σ, Q, q_0 are as in a k-pebble transducer.*
- *P is a set of transitions. Move transitions are as in a k-pebble transducer. Output transitions are replaced by the following:*

$(a_0, \bar{b}, q^{(i)}) \rightarrow (branch0)$

$(a_2, \bar{b}, q^{(i)}) \rightarrow ((q_1^{(i)}, q_2^{(i)}), branch2)$

The *branch0* and *branch2* transitions are analogous to the transitions of tree automata. The *branch0* transition stops the current computation branch and accepts. The *branch2* transition spawns two independent computations, in states $q_1^{(i)}$ and $q_2^{(i)}$ respectively; the input head is not moved. We define now formally the set of trees *inst*(A) accepted by A. We consider configurations γ of A on t as for k-pebble transducers, and denote Γ and γ_0 the set of all configurations, and the initial configuration respectively. The partial computation of an automaton A on a tree t is described by a word in Γ^*, denoting the configurations reached so far by the different branches. We define now the A-rewriting $\rightarrow$ from Γ to Γ^*, similar to the T-rewriting for transducers. Each transition in P contributes to $\rightarrow$. For every move transitions we define $\gamma \rightarrow \gamma'$ as for transducers. For every transition $(a, \bar{b}, q^{(i)}) \rightarrow (branch0)$ that applies to γ, we define $\gamma \rightarrow \varepsilon$. For every transition $(a, \bar{b}, q^{(i)}) \rightarrow ((q_1^{(i)}, q_2^{(i)}), branch2)$ that applies to $\gamma = (i, q^{(i)}, \bar{x})$ we define $\gamma \rightarrow \gamma_1\gamma_2$, where $\gamma_1 = (i, q_1^{(i)}, \bar{x}), \gamma_2 = (i, q_2^{(i)}, \bar{x})$. We next define the A-rewriting $\Rightarrow$ from Γ^* to Γ^* to be the congruential closure of $\rightarrow$. (1) if $\gamma \rightarrow w$, then $\gamma \Rightarrow w$, and (2) if $w_1 \Rightarrow w_2$ then $w.w_1 \Rightarrow w.w_2$ and $w_2.w \Rightarrow w_2.w$. A tree t is *accepted* by A, iff $\gamma_0 \stackrel{*}{\Rightarrow} \varepsilon$. The tree language $inst(A)$ accepted (or *recognized*) by A consists of all trees accepted by A.

Step 2. Inverse type inference can be achieved using k-pebble automata.

Proposition 4.6 *For each k-pebble transducer T and type τ there exists a k-pebble automaton A such that $inst(A) = \overline{\{t \mid T(t) \subseteq \tau\}}$. Here the bar denotes the complement of the set.*

Proof: Notice that $\overline{\{t \mid T(t) \subseteq \tau\}} = T^{-1}(\bar{\tau}) = \{t \mid T(t) \cap \bar{\tau} \neq \emptyset\}$, hence A can be obtained by composing T with the top-down automaton B describing the type $\bar{\tau}$. This is possible since B consumes the tree top-down, exactly in the order in which T produces it. Formally, let $T = (\Sigma, \Sigma', Q^T, q_0^T, P^T)$, and let $B = (\Sigma, Q^B, q_0^B, Q_F^B, P^B)$ be some top-down automaton accepting $\bar{\tau}$ (the complement of a regular set is also regular). A will be the "product automaton" of T and B, more precisely $A = (\Sigma, Q^T \times Q^B, (q_0^T, q_0^B), P)$. We define next the transitions P. Recall that T has *move* and *output* transitions, while A will have *move* and *branch* transitions.

- For every *move* transition in P^T:

  $$(a, \bar{b}, q_1^T) \to (q_2^T, d)$$

  and for every state $q^B \in Q^B$, A has a move transition

  $$(a, \bar{b}, (q_1^T, q^B)) \quad \to \quad ((q_2^T, q^B), d) \tag{3}$$

- For every output transition in P^T:

  $$(a, \bar{b}, q^T) \to (a_0', \mathit{output0})$$

  and for every state $q^B \in Q^B$ for which $(a_0', q^B) \in Q_F^B$ we have in A a *branch0* transition:

  $$(a_0, \bar{b}, (q^T, q^B)) \quad \to \quad (\mathit{branch0}) \tag{4}$$

- For every output transition in P^T:

  $$(a, \bar{b}, q^T) \to (a'(q_1^T, q_2^T), \mathit{output2})$$

  and for every branch transition in P^B:

  $$(a', q^B) \to (q_1^B, q_2^B)$$

  we have in A a *branch2* transition:

  $$(a, \bar{b}, (q^T, q^B)) \quad \to \quad ((q_1^T, q_1^B), (q_2^T, q_2^B), \mathit{branch2}) \tag{5}$$

To conclude the proof we need to show that for every tree t, $T(t) \cap inst(B) \neq \emptyset$ iff $t \in inst(A)$. This is intuitively obvious, since A simulates T in parallel with the computation of B on T's output. A formal proof requires us to resort to the definitions of $T(t)$, $inst(B)$, and $inst(A)$, and the details are more intricate.

We start with the "if" part, i.e. we are given $t \in inst(A)$. Then there exists an A-rewriting:

$$\gamma_{0,A,t} = w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n = \varepsilon$$

where, for each $i = 0, n$, $w_i \in \Delta^*_{A,t}$, i.e. w_i is a sequence of A-configurations on t; here $\gamma_{0,A,t}$ is the initial A-configuration. For every $i = 0, n$ we will construct a tree $t'_i \in T_{\Sigma'}(\Gamma_{T,t})$ such that the following rewritings hold in T:

$$\gamma_{0,T,t} = t'_0 \Rightarrow t'_1 \Rightarrow \ldots \Rightarrow t'_n = t'$$

and $t' \in T_{\Sigma'}$. Furthermore, we also construct for each $i = 0, n$ a sequence of B-configurations $v_i \in \Delta^*_{B,t'}$ such that the following rewriting holds in B:

$$\delta_{0,B,t'} = v_0 \stackrel{*}{\Rightarrow} v_1 \stackrel{*}{\Rightarrow} \ldots \stackrel{*}{\Rightarrow} v_n = \varepsilon$$

This proves the claim that there exists t' such that $t' \in T(t)$ and $t' \in inst(B)$.

We show how to construct t'_i and v_i, inductively on $i = 0, n$. Each B-configuration in v_i refers to a node in t', so apparently we cannot construct v_i before having $t' = t'_n$. We could either construct $t'_0, t'_1, \ldots, t'_n$ first, before $v_0, v_1, \ldots, v_n$, or notice that v_i only needs the nodes in t' which are already present in t'_i. Also, we rely on a tight connection between w_i, t'_i, and v_i: namely if we replace each A-configuration $\gamma = (j, (q^T, q^B), \bar{x})$ in w_i with $\bar{\gamma} = (j, q^T, \bar{x})$, then we obtain precisely the sequence of T-configurations on the leaves of t'_i. Moreover, if we replace each A-configuration $(j, (q^T, q^B), \bar{x})$ in w_i with (j, q^B, x) where x is the corresponding leaf node in t'_i, then we obtain precisely the sequence of B-configurations v_i.

For $i = 0$ define $t'_0 \stackrel{\text{def}}{=} \gamma_{0,T,t}$ (the initial T-configuration) and $v_0 \stackrel{\text{def}}{=} \delta_{0,B,t'}$ (the initial B-configuration). For $i > 0$ consider the A-rewriting $w_{i-1} \Rightarrow w_i$. By definition $w_{i-1} = w'.\gamma.w''$, $w_i = w'.u.w''$, and $\gamma \rightarrow u$. We construct t'_i, v_i from t'_{i-1}, v_{i-1} by cases on the type of the rewriting $\gamma \rightarrow u$. Case 1: the rewriting was generated by a move transition (equation (3)), hence $u = \gamma' \in \Delta_{A,t}$. Define $v_i \stackrel{\text{def}}{=} v_{i-1}$, and t'_i is obtained from t'_{i-1} by replacing the label $\bar{\gamma}$ with $\bar{\gamma}'$. Case 2: the rewriting was generated by a *branch0* transition (equation (4)), hence $u = \varepsilon$. Then v_i is obtained from v_{i-1} by erasing the configuration corresponding to γ, and t'_i is obtained from t'_{i-1} by replacing the leaf label γ with $a'_0 \in \Sigma'_0$ (where a' is the output symbol of the transition in T, see notations in the definition of A's transitions, P). Case 3: the rewriting was generated by a *branch2* transition (equation (5)), hence $u = \gamma_1\gamma_2$. Recall that $\gamma = (j, (q^T, q^B), \bar{x})$. Then, using the notations in the definition of A's transitions, t'_i is obtained from t'_{i-1} by replacing the leaf node x labeled $\bar{\gamma}$ with $a'(\bar{\gamma}_1, \bar{\gamma}_2)$: i.e. we labeled x with a', and introduced two new leaf nodes x_1, x_2, labeled $\bar{\gamma}_1 = (j, q^T_1, \bar{x})$ and $\bar{\gamma}_2 = (j, q^T_2, \bar{x})$ respectively. Similarly, v_i is obtained from v_{i-1} by replacing the configuration $\delta = (j, q^B, x)$ corresponding to γ with $\delta_1\delta_2$, where $\delta_1 = (j, q^B_1, x_1)$, $\delta_2 = (j, q^B_2, x_2)$. On can check in each of these cases that $t'_{i-1} \Rightarrow t'_i$, and that either $v_{i-1} = v_i$ or $v_{i-1} \Rightarrow v_i$, hence $v_{i-1} \stackrel{*}{\Rightarrow} v_i$.

We now sketch the "only if" direction. Let $t' \in T(t)$ such that $t' \in inst(B)$. We have a T rewriting:

$$\gamma_{0,T,t} = t'_0 \Rightarrow t'_1 \Rightarrow \ldots \Rightarrow t'_n = t'$$

with $t'_0, \ldots, t'_n \in T_{\Sigma'}(\Gamma_{T,t})$. We also have a B rewriting:

$$\delta_{0,B,t'} = v_0 \Rightarrow v_1 \Rightarrow \ldots \Rightarrow v_m = \varepsilon$$

Out of these two construct a A-rewriting:

$$\gamma_{0,A,t} = w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n = \varepsilon$$

proving that $t \in inst(A)$. For that we first need to "align" the second derivation to the first derivation, then we apply the construction from the "if" case in reversed.

For each $v_i \in \Delta^*_{B,t'}$ denote $\nu(v_i)$ the set of nodes in t' occurring in the configurations of v_i. The sets $\nu(v_i) - \nu(v_{i-1})$, $i = 1, m$ are a disjoint cover of the set $nodes(t')$, and each has exactly one element, which implies that t' has m nodes. This defines a certain order in which the nodes in t' are visited by the B-rewriting: $x_1, x_2, \ldots, x_m$, with $\{x_i\} = \nu(v_i) - \nu(v_{i-1})$, $i = 1, m$.

Considering the T-rewriting, let $\mu(t'_i)$ be the set of nodes in t'_i labeled with elements in $\Gamma_{T,t}$. The sets $\mu(t'_i) - \mu(t'_{i-1})$, $i = 1, n$ form a disjoint cover of $nodes(t')$ and each has 0 or 1 elements. It follows that the T-rewriting also defines an order on the nodes in t', corresponding to the order in which they were introduced. Since both $\Rightarrow$ rewritings are confluent, we may assume that this is the same order $x_1, \ldots, x_m$ as above.

Next we expand the B rewriting to:

$$\delta_0 = v_{j_0} \stackrel{*}{\Rightarrow} v_{j_1} \stackrel{*}{\Rightarrow} \ldots \stackrel{*}{\Rightarrow} v_{j_n} = \varepsilon$$

such that $0 = j_0 \leq j_1 \leq \ldots \leq j_n = m$, and for every $i = 1, m$, $j_i = j_{i-1}$ if $\mu(t'_i) - \mu(t'_{i-1}) = \emptyset$, and $j_i = j_{i-1} + 1$ if $\mu(t'_i) - \mu(t'_{i-1}) \neq \emptyset$. It follows now that $\nu(v_{j_i}) = \mu(t'_i)$ for every $i = 0, n$. This gives us the required alignment.

Finally, we construct w_i from t'_i and v_{j_i} as follows. For a T-configuration $\gamma = (p, q^T, \bar{x})$ and a B-configuration $\delta = (q^B, x)$, denote (γ, δ) the A-configuration $(p, (q^T, q^B), \bar{x})$. Let $\gamma_1.\gamma_2 \ldots \gamma_p$ be the T-configurations on the leaves of t'_i. Let v_{j_i} be $\delta_1.\delta_2 \ldots \delta_p$ (must be the same number p because $\mu(t'_i) = \nu(v_{j_i})$). Then define $w_i = (\gamma_1, \delta_1) \ldots (\gamma_p, \delta_p)$. The proof of the fact that $w_{i-1} \Rightarrow w_i$ in A is done by a straightforward case analysis. □

Step 3. Finally, we can show the following result, which is the cornerstone of our approach to inverse type inference, and technically the most interesting.

Theorem 4.7 *k-pebble tree automata accept precisely the regular tree languages.*

Proof: Clearly, k-pebble automata can simulate classical non-deterministic top-down tree automata using a single pebble, so each regular tree language is accepted by some k-pebble automaton. The converse is considerably harder. To prove it, we show how to construct for each k-pebble tree automaton A an MSO (Monadic Second-Order logic) formula φ_A s.t. for every tree t, A accepts t iff $t \models \varphi_A$. Since MSO formulas define precisely the regular tree languages [34], the theorem follows.

The classic result that MSO on trees defines precisely the regular tree languages assumes a certain representation of trees as first-order structures. A tree t is represented by a structure $(D, succ1, succ2, (R_a)_{a\in\Sigma})$, where D is the set of nodes, $succ1(x, y)$ and $succ2(x, y)$ denote the fact that y is the left (right) child of x, and $R_a(x)$ holds if node x has label a. As a warm-up, consider a simple example: the descendant relation among nodes of t. This is defined by the MSO formula:

$$\varphi(x, y) = \forall S.\ (\ S(x) \wedge \mathit{closed}(S) \Longrightarrow\ S(y))$$

where

$$\begin{aligned} \mathit{closed}(S) = \quad & (\forall u\ \forall v\ (S(u) \wedge succ_1(u, v) \Longrightarrow S(v))) \\ & \wedge\ (\forall u\ \forall v\ (S(u) \wedge succ_2(u, v) \Longrightarrow S(v))) \end{aligned}$$

Clearly, $\varphi(x, y)$ holds iff y is a descendant of x in t. Indeed, x's descendants can be described as the smallest set S containing x and closed under $succ_1$ and $succ_2$. Note the universal quantification of S.

Also by way of warm-up, consider an *and/or* tree whose internal nodes are labeled $\wedge, \vee$ and whose leaves are labeled $0, 1$. It is easy to see that there is a 1-pebble automaton accepting precisely those trees that evaluate to 1 when viewed as Boolean circuits. An MSO formula φ defining the same set of trees is obtained by "traversing" the tree bottom-up, from leaves to root, and defining the set of nodes whose subtrees evaluate to 1 using a unary relation S:

$$\varphi = \forall S.\ ((\ \forall x.R_1(x) \Longrightarrow S(x)) \wedge \mathit{reverse\text{-}closed}(S)) \Longrightarrow S(\mathit{root})$$

where

$\mathit{reverse\text{-}closed}(S) = \forall x \forall y\ .(R_\vee(x)\ \wedge\ (succ_1(x, y) \vee succ_2(x, y))\ \wedge S(y)) \Longrightarrow S(x) \wedge\ \forall x \forall y \forall z\ .(R_\wedge(x) \wedge succ_1(x, y) \wedge succ_2(x, z)\ \wedge\ S(y)\ \wedge\ S(z)) \Longrightarrow S(x)$

We reduce the simulation problem of a k-pebble automaton to the *Alternating Graph Accessibility Problem*, AGAP [22]. An alternating graph (or *and/or*) graph is a graph $G = (V, E)$ whose nodes V are partitioned into *and*-nodes and *or*-nodes: $V = V_\wedge \cup V_\vee$. The problem consists in deciding whether a node $x \in V$ is *accessible*, where accessibility is defined as follows: an *and*-node is accessible if all its successors are accessible; an *or*-node is accessible if at least one of its successors is accessible. The set of accessible nodes x is defined in MSO by:

$$\varphi(x) \quad = \quad \forall S.(\mathit{reverse\text{-}closed}(S) \Rightarrow S(x)) \tag{6}$$

where *reverse-closed*(S) is:

$$\begin{array}{rl} & \forall y.(D_\vee(y) \wedge \exists z.(E(y,z) \wedge S(z)) \Rightarrow S(y)) \\ \wedge & \forall y.(D_\wedge(y) \wedge \forall z.(E(y,z) \Rightarrow S(z)) \Rightarrow S(y)) \end{array} \tag{7}$$

Given a k-pebble automaton A and a tree t, construct the following *and/or* graph $G_{A,t} = (V, E)$. Its *or*-nodes are all configurations of A on t, $V_\vee = \Gamma$; its set of *and*-nodes consists of a unique node ε plus all pairs of configurations (γ_1, γ_2) sharing the same $\bar{x}$ component: $V_\wedge = \{\varepsilon\} \cup \{(\gamma_1, \gamma_2) \mid \gamma_1 = (i, q, \bar{x}), \gamma_2 = (i, q', \bar{x}), i = 1, k, q, q' \in Q_i, \bar{x} \in (nodes(t))^i\}$. The set of edges E contains two kinds of edges: first all edges of the form (γ, u) for which $\gamma \to u$ where $\to$ is the A-rewriting from Γ to Γ^* (recall that u is either a configuration γ', or a pair of configurations $\gamma_1\gamma_2$, or ε); second all edges of the form $((\gamma_1, \gamma_2), \gamma_i)$, for $i = 1, 2$. It follows directly from the definitions of *inst*(A) and of AGAP that a tree t is in *inst*(A) iff the initial configuration γ_0 is accessible in the graph $G_{A,t}$. Hence it only remains to show that we can express the AGAP problem on $G_{A,t}$ in MSO.

The difficulty lies in the fact that the nodes in $G_{A,t}$ are tuples of nodes from the input structure t (a configuration $\gamma = (i, q, \bar{x})$ is represented by the i-tuple $(x_1, \ldots, x_i)$; q does not depend on the tree t and will be encoded separately; a pair (γ_1, γ_2) is also represented by an i-tuple, since γ_1, γ_2 have the same $\bar{x}$). Hence the set S in Eq. (6) is no longer unary. To circumvent that, we rely on a special property of $G_{A,t}$. Namely if two nodes described by an i-tuple $(x_1, \ldots, x_i)$ and a j-tuple $(y_1, \ldots, y_j)$ are connected by an edge, then either $i = j$, or $i = j + 1$, or $i = j - 1$, and the tuples agree on all but the last position. This follows from the stack discipline on pebbles in A (only the last pebble can be moved) and allows us to quantify independently, on different portions of the graph. The construction of the MSO formula below relies on this observation, and we outline this construction next.

We denote $A = (\Sigma, Q, q_0, P)$, $Q = Q_1 \cup Q_2 \cup \ldots \cup Q_k$ and we enumerate the states in Q such that $Q = \{q_0, q_1, \ldots, q_n\}$, and: $Q_1 = \{q_0, \ldots, q_{n_1}\}$, $Q_2 = \{q_{n_1+1}, \ldots, q_{n_2}\}$, ..., $Q_k = \{q_{n_{k-1}+1}, \ldots, q_{n_k}\}$. We show first the case when $k = 1$, so A uses a single pebble, to illustrate how one encodes the state in a configuration and how to encode the transitions. In this case configurations can be assimilated with pairs (q_j, x), and the MSO formula φ defining acceptance by A uses a different unary relation S_j for each state $q_j \in Q$. Namely φ_A is:

$$\varphi_A = \forall S_0.\forall S_1 \ldots \forall S_{n_1}(\textit{reverse-closed} \Longrightarrow S_0(\textit{root})) \tag{8}$$

where *reverse-closed* is a sentence stating that $S_0, S_1, \ldots, S_{n_1}$ are closed under reverse transitions of A according to the *and/or* semantics. Thus, φ_A states that the initial configuration of A (pebble at the root in state q_0) is accessible in the *and/or* graph $G_{A,t}$. It follows that φ_A holds iff A accepts the tree.

The formula *reverse-closed* is a direct representation of the transitions in A (and edges in $G_{A,t}$) in MSO. For each of A's transitions *reverse-closed* includes a corresponding conjunct:

$$\textit{reverse-closed} = \bigwedge_{p \in P} \psi_p$$

We illustrate ψ_p for some transitions:

- for $p = ((a, q_u) \to (q_v, \mathit{stay}))$,
$$\psi_p = \forall\, x.(R_a(x) \wedge S_v(x)) \Longrightarrow S_u(x)$$
- for $p = ((a, q_u) \to (q_v, \mathit{down\text{-}left}))$,
$$\psi_p = \forall x.\ \forall y.(R_a(x) \wedge \mathit{succ}_1(x, y) \wedge S_v(y)) \Longrightarrow S_u(x)$$
- for $p = ((a, q_u) \to (q_v, q_w, \mathit{branch2}))$,
$$\psi_p = \forall x.(R_a(x) \wedge S_v(x) \wedge S_w(x)) \Longrightarrow S_u(x)$$
- for $p = ((a, q_u) \to (\mathit{branch0}))$,
$$\psi_p \quad = \quad \forall x.R_a(x) \Longrightarrow S_u(x) \tag{9}$$

To see that φ_A holds iff A accepts the tree, it suffices to notice the similarity between Eq. (6) and Eq. (8). For that, one needs to observe how the formula for *reverse-closed* in a general graph (Eq. (7)) becomes the formula above when instantiated to $G_{A,t}$. For example notice that each *and*-node in $G_{A,t}$ has exactly two successors (hence the universal quantifier in (7) becomes an $\wedge$ in *branch2*).

We now extend Equation (8) to the case when k is arbitrary. We will define a predicate *reverse-closed*$^{(i)}$, for each $i = 1, k$, stating that $S_{n_{i-1}+1}, \ldots, S_{n_i}$ are closed under reverse transitions of A. Then, the MSO formula equivalent to A will be φ_A in Equation (8), with *reverse-closed* replaced with *reverse-closed*$^{(1)}$. The predicate *reverse-closed*$^{(i)}$ assumes that pebbles $1, 2, \ldots, i-1$ are fixed, and their positions described by the free variables $x_1, \ldots, x_{i-1}$; it also has free variables $S_0, S_1, \ldots, S_{n_{i-1}}$. The predicate only considers moves affecting pebbles $i, i+1, \ldots, k$. Partition A's transitions into $P = P_1 \cup \ldots \cup P_k$, where P_i is the set of transitions of the form $(a, \bar{b}, q^{(i)}) \to \ldots$. Then:

$$\mathit{reversed\text{-}closed}^{(i)} = \bigwedge_{p \in P_i} \psi_p$$

For move transitions p, ψ_p is the same as for *reverse-closed* above, except that now it also inspect the presence/absence of the previous $i-1$ pebbles. For example, for $i = 3$ and $p = ((a, 01, q_u) \to (q_v, \mathit{stay}))$ corresponding ψ_p is

$$\psi_p = \forall\, x.(R_a(x) \wedge (x \neq x_1 \wedge x = x_2) \wedge S_v(x)) \Rightarrow S_u(x).$$

In general, given $b \in \{0, 1\}$ we write $x =_b y$ to mean $x = y$ when $b = 1$ and $x \neq y$ when $b = 0$; given $\bar{b} \in \{0, 1\}^{i-1}$, we write $\mathit{pebbles}_{\bar{b}}(x)$ for the formula $\bigwedge_{j=1,i-1} x =_{b_j} x_j$. Then, for $p = ((a, \bar{b}, q_u^{(i)}) \to (q_v^{(i)}, \mathit{stay}))$,

$$\psi_p = \forall\, x.(R_a(x) \wedge \mathit{pebbles}_{\bar{b}}(x) \wedge S_v(x)) \Longrightarrow S_u(x).$$

The other move and branch transitions are similar.

The new transitions are the *pick* and *place* transitions. These determine the following conjuncts in *reverse-closed*$^{(i)}$:

- for $p = ((a, \bar{b}, q_u^{(i)}) \to (q_v^{(i+1)}, place))$,

$$\psi_p = \forall x_i.(R_a(x_i) \land pebbles_{\bar{b}}(x_i) \land \varphi^{(i+1)} \Longrightarrow S_u(x_i))$$

  where

$$\varphi^{(i+1)} = \forall S_{n_i+1} \ldots \forall S_{n_{i+1}}(reverse\text{-}closed^{(i+1)} \Longrightarrow S_v(root))$$

  Note the resemblance of $\varphi^{(i+1)}$ to Equation (8): here q_v acts as an initial state for pebble $i+1$.

- for $p = ((a, \bar{b}, q_u^{(i)}) \to (q_v^{(i-1)}, pick))$,

$$\psi_p = \forall x_i.(R_a(x_i) \land pebbles_{\bar{b}}(x_i) \land S_v(x_{i-1}) \Longrightarrow S_u(x_i))$$

  Note the resemblance to Equation (9): here q_u acts as a terminal state (*branch0*) for pebble i.

This completes the proof of the translation of A into MSO. The resulting formula φ_A has size exponential in k (due to the replication of the $\varphi^{(i)}$ subformulas), but the quantifier depth is k (for first-order variables x_i) and $\mid Q \mid$ (for second order variables S_j). Note that the stack discipline imposed on the use of pebbles is essential to the construction in the proof. □

The algorithm resulting from our typechecking technique for k-pebble transducers is hyperexponential in k (so non-elementary). In fact, this is also a lower bound, as stated next. Note that the lower bound holds even under rather stringent assumptions.

Theorem 4.8 *[35] The typechecking problem for deterministic k-pebble transducers is non-elementary, even for fixed input and output types.*

Proof: We use a reduction of the emptiness problem for star-free generalized regular expressions (formed using alphabet symbols, union, concatenation, and complement) which is known to be non-elementary [36]. We use a fixed input type τ_1 that allows encoding a string $w \neq \epsilon$ over Σ as a binary tree $enc(w)$ over $\Sigma \cup \{-\}$ as follows: $enc(a) = a(), enc(av) = a(-, enc(v))$. It is easily seen that for every star-free generalized regular expression r one can construct in PTIME a deterministic k-pebble automaton A_r without branching accepting $\{enc(w) \mid w \in lang(r)\}$. Thus, r is empty iff $inst(A_r)$ is empty. Now construct a deterministic k-pebble transducer T_r that given an input $t \in \tau_1$ outputs $b()$ if A_r rejects t and $b(c()c())$ if A_r accepts t. Consider the fixed output type defining the singleton $\{b()\}$. It follows that T_r typechecks iff $lang(r) = \emptyset$. □

As a useful side-effect of the above proof, we can state:

Corollary 4.9 *The emptiness problem for deterministic k-pebble automata without branching is non-elementary.*

The number of pebbles in the pebble transducer is the main source of complexity in the typechecking procedure. This is not unexpected. Indeed, as noted by [21] for the case of strings, automata with k pebbles (more restricted than ours) are *extremely concise*: their equivalent automata without pebbles are larger by a tower of exponentials of height k. Automata with pebbles are convenient and very economical, but the complexity of usual decision problems can be expected to be harder in proportion to their conciseness.

5 Extensions

The main limitations of our typechecking approach are the high complexity and the lack of data values. Both can be dealt with in restricted cases: of special interest are transformations expressed in the usual XML query languages. A full treatment is beyond the scope of this paper; here we mention some initial results indicating that both limitations can be overcome in significant practical cases.

Complexity of typechecking As discussed earlier, typechecking k-pebble transducers is non-elementary. Thus, typechecking in its full generality appears to be prohibitively expensive. However, our approach can be used in restricted cases of practical interest for which typechecking can be reduced to emptiness of automata with *very few pebbles*. Fortunately, even one or two pebbles can be quite powerful. For example, typechecking selection XML-QL queries without joins (i.e., queries that extract the list of bindings of a variable occurring in a tree pattern such as described in Section 2.2 and Example 3.5) can be reduced to emptiness of a 1-pebble automaton with exponentially many states (yielding a total complexity of 2-EXPTIME). Another interesting special case consists of XML-QL queries with tree patterns and constructed answers restricted so that grouping is done one variable at a time. For such queries, the automaton uses two pebbles and exponentially many states, if the output type is a DTD. (If grouping is done at most m variables at a time, the automaton uses $2m$ pebbles.)

Data Values To model `#PCDATA` in XML we assume an infinite alphabet D of data values in addition to Σ. Trees are now in $T_\Sigma(D)$, while types are regular tree languages over $T_\Sigma(\{d\})$, i.e. where all data values are treated as a distinguished leaf symbol d. The k-pebble transducers are extended with three new kinds of transitions (see also Section 1): a comparison predicate $x = y$ that takes two data values x, y and enters either some state q_1 (when $x = y$) or q_2 (when $x \neq y$), unary comparisons on data values, and an output transition which copies a data value from the input tree to the output tree. All XML-QL queries, including those with selections and joins on data values, can be expressed with the extended transducers. The last two kinds of transitions do not affect typechecking. For example, to handle unary predicates we apply the technique in [1]: if the machine uses m predicates, then replace the infinite set of data values with 2^m new constants. The comparison predicate $x = y$ however renders typechecking undecidable in general, because of a reduction from the finite satisfiability problem for first-order logic: the equality predicate allows to evaluate any first-order formula on standard encodings of first-order structures as labeled trees with data values.

However, the typechecking techniques in this paper can be extended to an important class of queries with data value joins. Intuitively, these are queries where all equality tests performed are independent of each other. Consequently, all truth assignments to the equality tests are consistent. As far as typechecking is concerned, the actual equality tests can therefore be replaced by nondeterministic guesses of their truth value, without the risk of inconsistent guesses. In practice, some queries expressed in languages such as XML-QL can be implemented in this manner. To illustrate, assume a relational schema consisting of three relations `Person(pid, name)`, `Worksin(pid, did)`, `Dept(did, name)`, with `pid, did` keys in `Person, Dept` respectively, and consider a three way join Q = `Person` $\bowtie$ `Worksin` $\bowtie$ `Dept`. Such joins are typical in XML-QL queries exporting a relational database to an XML view [19]. The following extended k-pebble transducer T implements Q and performs only independent comparisons. In an outermost loop T iterates `w` over `Worksin`. At each step it iterates `p` over `Person` checking `w.pid = p.pid`: T *stops the inner loop* when an equality is found. Next it iterates over `Dept` and stops when an equality is found. Each comparison done by T is independent on all previous comparisons, i.e. both outcomes $x = y$ or $x \neq y$ are consistent with all previous comparisons. Notice that other implementations of the three way join may not have the independence property. Thus, T can be simulated by a non-deterministic transducer T' with inputs in $T_\Sigma(\{d\})$ by replacing every predicate $x = y$ with output states q_1, q_2 with a non-deterministic choice between q_1 and q_2. Every nondeterministic choice of T' corresponds to a possible run of T (on certain data values) and we can apply the typechecking techniques from this paper to T'.

6 Conclusions

We have developed a general, robust framework for typechecking XML transformers. The k-pebble transducer provides an abstraction that can express existing XML transformation languages such as XML-QL and a subset of XSLT. The regular tree languages capture current DTDs and several proposed extensions. Thus, our framework for typechecking is likely to remain relevant as the XML standards evolve.

Future work has to address the complexity of typechecking, and the treatment of data values, in the context of restricted classes of XML transformations. Restrictions are often acceptable in practice, and may lead to efficient typechecking algorithms. We believe that k-pebble transducers provide a good framework for such a study, as suggested by our preliminary results in this direction.

Acknowledgment We would like to thank Nils Klarlund for his comments. Thanks also to Thomas Schwentick for pointing out the connection between k-pebble automata and star-free generalized regular expressions, yielding the

lower bound in Theorem 4.8.

References

[1] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 122–133, 1997.

[2] A. Aho and J. Ullman. Translations on a context free grammar. *Information and Control*, 19(19):439–475, 1971.

[3] D. Beech, S. Lawrence, M. Maloney, N. Mendelsohn, and H. Thompson. Xml schema part 1: Structures, May 1999. `http://www.w3.org/TR/xmlschema-1/`.

[4] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *Proceedings of the International Conference on Database Theory*, pages 296–313, 1999.

[5] C. Beeri and Y. Tzaban. SAL: an algebra for semistructured data and XML. In *Proceedings of WebDB*, Philadelphia, PA, June 1999.

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

[7] P. Biron and A. Malhotra. Xml schema part 2: Datatypes, May 1999. `http://www.w3.org/TR/xmlschema-2/`.

[8] 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/`.

[9] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[10] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

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

[12] J. Clark. XSL transformations (XSLT) specification, 1999. available from the W3C, `http://www.w3.org/TR/WD-xslt`.

[13] J. Clark and M. Makoto. Relax ng specification, 2001. available from `http://www.oasis-open.org/committees/relax-ng/`.

[14] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion ! In *Proceedings ACM-SIGMOD International Conference on Management of Data*, pages 177–188, 1998.

[15] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the Eights International World Wide Web Conference (WWW8)*, pages 77–91, Toronto, 1999.

[16] J. Engelfriet, H. Hoogenboom, and J. V. Best. Trips on trees. *Acta Cybernetica*, 14:51–64, 1999.

[17] 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`.

[18] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: experience with a Web-site management system. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 414–425, 1998.

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

[20] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Information Systems*, 8(2):117–132, March 1997.

[21] N. Globerman and D. Harel. Complexity results for multi-pebble automata and their logics. In *Proceedings of ICALP*, pages 73–82, Jerusalem, Israel, 1994.

[22] R. Greenlaw, H. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation. P-Completeness Theory.* Oxford University Press, New York, Oxford, 1995.

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

[24] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of ICFP*, pages 11–22, 2000.

[25] B. Ludaescher, Y. Papakonstantinou, P. Velikhov, and V. Vianu. View definition and dtd inference for xml. In *Workshop on Semistructured Data and Nonstandard Data Formats*, January 1999.

[26] S. Maneth and F. Neven. Structured document transformations based on xsl. In *In Proc. Eighth Int'l. Workshop on Database Programming Languages*, Scotland, 1999.

[27] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of VLDB*, pages 315–326, Edinburgh, UK, September 1999.

[28] T. Milo and D. Suciu. Type inference for queries on semistructured data. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 215–226, 1999.

[29] F. Neven and T. Schwentick. Query automata. In *Symposium on Principles of Database Systems*, pages 205–214, Philadelphia, PA, 1999.

[30] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE International Conference on Data Engineering*, pages 251–260, March 1995.

[31] Y. Papakonstantinou and P. Velikhov. Enhancing semistructured data mediators with document type definitions. In *Int'l Conf. on Data Engineering*, 1999.

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

[33] J. Robie. The design of xql, 1999. `http://www.texcel.no/whitepapers/xql-design.html`.

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

[35] T. Schwentick. Personal communication, 2000.

[36] L. Stockmeier. *The complexity of decision problems in automata theory and logic*. PhD thesis, Thesis, Massachusetts Institute of Technology, 1974.

[37] 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.