XML with Data Values: Typechecking Revisited*

Noga Alon
Tel Aviv University
noga@tau.math.ac.il

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

Frank Neven†
Limburgs Universitair Centrum
frank.neven@luc.ac.be

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

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

Abstract

We investigate the *typechecking* problem for XML queries: statically verifying that every answer to a query conforms to a given output DTD, for inputs satisfying a given input DTD. This problem had been studied by a subset of the authors in a simplified framework that captured the structure of XML documents but ignored data values. We revisit here the typechecking problem in the more realistic case when data values are present in documents and tested by queries. In this extended framework, typechecking quickly becomes undecidable. However, it remains decidable for large classes of queries and DTDs of practical interest. The main contribution of the present paper is to trace a fairly tight boundary of decidability for typechecking with data values. The complexity of typechecking in the decidable cases is also considered.

1 Introduction

Databases play a crucial role in new internet applications ranging from electronic commerce to Web site management to digital government. Such applications have redefined the technological boundaries of the area. The emergence of the Extended Markup Language (XML) as the likely standard for representing and exchanging data on the Web has confirmed the central role of semistructured data but has also redefined some of the ground rules. Perhaps the most important is that XML marks the "return of the schema" (albeit loose and flexible) in semistructured data, in the form of its Data Type Definitions (DTDs), which constrain valid XML documents. The benefits of 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, DTDs 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 which are valid with respect to the specified DTD. This raises the issue of *(static) typechecking*: verifying at compile time that *every* XML document which is the result of a specified query applied to a valid input document, satisfies the output DTD.

The XML standard is still under development. The basic document model, the precise definition of DTD, and the languages for querying and transforming XML documents are in a state of flux. Languages proposed for XML queries and transformations include XML-QL [10], XSLT [5], XPath [7], XQL [22], and Quilt [6]. 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 [1, 2].

In a previous paper [18], a subset of the authors investigated the typechecking problem in a limited framework. The bare-bones structure of XML documents was abstracted as ordered labeled trees. To circumvent the lack of a generally accepted notion of DTD, regular tree languages were used as types. The wide variety of XML query languages was addressed by using a very general model of tree transformer called *k-pebble transducer*, which was shown to subsume the tree manipulation core of XML-QL and XSL. The main result of [18] is that typechecking k-pebble transducers is decidable. That is, given an input type τ_1, an output type τ_2, and a k-pebble transducer T, it is decidable whether $T(\tau_1) \subseteq \tau_2$. However, this result has serious limitations. Chief among them is the absence of data values from the model. In fact, it is easily seen that typechecking

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

†Post-doctoral researcher of the Fund for Scientific Research, Flanders.

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

becomes undecidable when k-pebble transducers are augmented with comparisons on data values.

The present paper extends the investigation of [18] to typechecking of queries with comparisons of data values. We focus on query languages in the style of XML-QL, and on typing using DTDs and their variants. The main contribution of the paper is to trace the boundary of decidability of the typechecking problem in the presence of data values, where the parameters consist of various features of the query language and of the DTDs. On the decidability side, we show that typechecking is decidable for queries with non-recursive path expressions, arbitrary input DTD, and output DTD specifying conditions on the number of children of nodes with a given label. We are able to extend this to DTDs using star-free regular expressions, and then full regular expressions, by increasingly restricting the query language. We also establish lower and upper complexity bounds for our typechecking algorithms. The upper bounds range from PSPACE to non-elementary, but it is open if these are tight. The lower bounds range from CO-NP to PSPACE. On the undecidability side, we show that typechecking becomes undecidable as soon as the main decidable cases are extended even slightly. We mainly consider extensions with recursive path expressions in queries, or with types decoupled from tags in DTDs (also known as specialization). This traces a fairly tight boundary for the decidability of typechecking with data values.

Related work Typechecking XML transformations is an important research problem that has been quite intensively investigated lately. In our prior work [18] we showed that typechecking is decidable for a certain class of transformations. That class is incomparable with the class of transformations discussed here, since it only applies to trees without data values, but on such trees they are more powerful than the XML-QL-style transformations we consider here. Type inference for a more restricted class of XML transformations is considered in [20]. The approach taken there is to extend the types from regular path expressions to context free grammars to be able to express certain inferred types.

A different approach to type checking XML transformations is taken by XDuce [14, 15]. XDuce is a general-purpose functional language in the style of ML [16], whose types are essentially DTDs with types decoupled from tags. Recursive functions can be defined over XML data by pattern matching against regular expressions. XDuce performs static typechecking for these functions, verifying that the output of a function will always be of the claimed output type. However, the typechecking algorithm is only sound, not complete: one can write in XDuce a function that always returns results of the required output type, but that the typechecker rejects. While this is expected in a general-purpose language that can express non-terminating functions, the typechecker fails even on simple functions that are expressible in, say, XML-QL, and for which we know already that type checking is decidable [18]. In other words, XDuce's typechecker could be strengthened. The goal in XDuce however differs from ours: XDuce focuses on making the typechecker practical, both for the application writer and for the language implementer, while our work is meant to study the theoretical limits of typechecking.

Yet another approach to typechecking is taken by YAT [8, 9]. This system for semistructured data has an original type system, based on unordered types. YATL (the query language in YAT, combining datalog with Skolem functions) admits type inference.

Type inference for the variables occurring in a query has been considered in [17]. The problem there consists in finding all types for the variables found in a query: this is different from inferring the output type of an XML transformation.

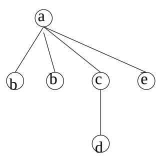
Organization The paper is organized as follows. The first section develops the basic framework, including our abstractions of XML documents and DTDs, and the variant of XML-QL used as a query language. Section 3 presents the decidability and complexity results, and Section 4 the undecidability results. The paper ends with brief conclusions. An appendix contains some of the proofs.

2 Basic Framework

We introduce here the basic formalism used throughout the paper, including our abstractions of XML document, DTD, and XML-QL.

Data trees Data trees are our abstraction of XML documents. They capture the nesting structure of XML elements, their tags, and data values associated with them. We fix an infinite set of data values denoted by $\mathcal{D}$. A *data tree* over finite alphabet Σ is a triple $\langle t, label, val \rangle$ where t is a finite ordered tree, $label$ is a mapping from the nodes of t to Σ, and val is a mapping from the nodes of t to $\mathcal{D}$. Given a tree t, we denote its set of nodes by $nodes(t)$ and its root by $root(t)$. We refer to elements in Σ as *tags* of t and to elements in $\mathcal{D}$ assigned by val as data values of t. Note that we do not restrict the number of children of any given node, so data trees are unranked. We denote the set of data trees over Σ by $\mathcal{T}_{\Sigma,\mathcal{D}}$. The set of finite labeled ordered trees over Σ (without data values) is denoted by $\mathcal{T}_\Sigma$.

Types and DTDs DTDs and their variants provide a typing mechanism for XML documents. We will use several notions of types for data trees. The first corresponds closely to the DTDs proposed for XML documents, and we therefore (by slight abuse) continue to use the same term. A DTD consists of an extended context-free grammar[1] over alphabet Σ (we make no distinction between terminal and non-terminal symbols). A data tree $\langle t, label, val\rangle$ over Σ satisfies a DTD D if the tree $\langle t, label\rangle$ is a derivation tree of the grammar. For example, the tree



is valid w.r.t. to the DTD: $a \to b^*.c.e$; $b \to \varepsilon$; $c \to d^*$; $d \to \varepsilon$; $e \to \varepsilon$

The set of data trees satisfying a DTD τ is denoted by $inst(\tau)$. Note that DTDs place no restriction on the data values in a data tree, and concern exclusively the tags.

Usual DTDs use regular languages to describe the allowed sequences of children of a node. However, weaker specification mechanisms are sufficient in many applications. We consider throughout the paper several such alternative mechanisms, each yielding a restricted kind of DTD. To understand the rationale behind the restrictions, it is useful to consider a logic-based point of view. First, note that strings over alphabet Σ can be viewed as logical structures over the vocabulary $\{<, (O_\sigma)_{\sigma\in\Sigma}\}$ where $<$ is a binary relation and every O_σ is a unary relation. A string $w = a_1 \dots a_n$ is represented by the logical structure $(\{1,\dots,n\}; <, (O_\sigma)_{\sigma\in\Sigma})$ where $<$ is the natural order on $\{1,\dots,n\}$, and for each i, $i \in O_\sigma$ iff $a_i = \sigma$. It is well-known that regular languages are exactly those definable by Monadic Second-Order (MSO) logic[2] on the logical vocabulary of strings [4, 11]. However, this is much more powerful than needed by most DTDs. In many cases, the required properties of valid strings can be expressed simply in First-Order logic (FO). This corresponds to a well-known subset of the regular languages, called *star-free* [24]. There is a language-theoretic characterization of star-free languages: they are precisely described by the *star-free regular expressions*, which are build from single symbols and ϵ using concatenation, union, and complement. We call DTDs using only star-free regular expressions *star-free DTDs*, and we refer to unrestricted DTDs as *regular DTDs*.

We will consider an even simpler class of DTDs, which specify cardinality constraints on the tags of children of a node, but does not restrict their order. Such DTDs are useful either when order is irrelevant, or when the order of tags in the output is hard-wired by the syntax of the query and so can be factored out. We use a logic called SL, inspired by [19]. The syntax of the language is as follows. For every $a \in \Sigma$ and natural number i, $a^{=i}$ and $a^{\geq i}$ are *atomic SL formulas*. Every atomic formula is a formula and the negation, conjunction, and disjunction of formulas are also formulas. A word w over Σ satisfies an atomic formula $a^{=i}$ if it has exactly i occurrences of a, and similarly for $a^{\geq i}$. Satisfaction of Boolean combination of atomic formulas is defined in the obvious way. As an example, consider the SL formula

$$\text{co-producer}^{\geq 1} \to \text{producer}^{\geq 1}.$$

This expresses the constraint that a co-producer can only occur when a producer occurs. One can check that languages expressed in SL correspond precisely to properties of structures over the vocabulary $\{<, (O_\sigma)_{\sigma\in\Sigma}\}$ that can be expressed in FO without using the order relation, $<$. Thus, SL forms a natural subclass of star-free regular expressions. We refer to DTDs using the language SL as *unordered DTDs*.

We have so far defined DTDs and several restrictions. We next consider an orthogonal *extension* of basic DTDs, also present in more recent DTD proposals such as XML-Schemas. This is motivated by a severe limitation of basic DTDs: their definition of the type of a given tag depends only on the tag itself and not on the context in which it occurs. For example, this means that the singleton $\{t\}$ where t is the tree[3] $a(b(c), b(d))$ cannot be described by a DTD, because the "type" of the first b differs from that of the second b. This naturally leads to an extension of DTDs with *specialization* (also called decoupled types) which, intuitively, allows defining the type of a tag by several "cases" depending on the context. Formally, we have:

Definition 2.1 *A* specialized *DTD over Σ is a tuple $\tau = (\Sigma, \Sigma', \tau', \mu)$ where*

- *Σ and Σ' are finite alphabets;*

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

[2] MSO is first-order logic augmented with quantification over sets.

[3] We denote a tree with root r and sequence of subtrees $t_1, \dots, t_n$ by $r(t_1, \dots, t_n)$.

- τ' *is a DTD over* Σ'*; and*
- μ *is a mapping from* Σ *to* Σ'.

A tree t over Σ *satisfies a specialized DTD* τ*, if* $t \in \mu(inst(\tau'))$.

Intuitively, Σ' provides for some a's in Σ a set of specializations of a, namely those $a' \in \Sigma'$ for which $\mu(a') = a$. We also denote by μ the homomorphism induced on strings and trees by μ. Interestingly, it turns out that specialized DTDs are precisely equivalent to regular tree automata over unranked trees [3, 20]. This is more evidence that specialized DTDs are a robust and natural specification mechanism. We will consider specialization in conjunction with regular DTDs, star-free DTDs, and unordered DTDs.

The query language QL The query languages we consider are subsets of XML-QL [10]. The queries have a `where` clause and a `construct` clause. The `where` clause is a tree pattern whose nodes are variables and whose edges are labeled by regular path expressions. Additionally, there may be comparisons of data values associated with the variables. The `where` clause is used to extract bindings of the variables, by matching the pattern into the input tree. The `construct` clause specifies how to construct an answer tree from the bindings. Nesting is allowed: the `construct` clause may use sub-queries. We later consider several extensions and restrictions of the language.

We next define our basic query language, denoted QL. We fix an alphabet Σ and assume given an infinite set Var of variables (denoted $x, y, z, \ldots$, possibly subscripted).

Definition 2.2 *Let* $\bar{z} = z_1, \ldots, z_k$ *be sequence of variables in Var, and denote* $Z = \{z_1, \ldots, z_k\}$*. A* QL *query* $q(\bar{z})$ *is a pair* $\langle W, C\rangle$ *such that the following hold.*

- W *(the* `where` *clause) consists of a finite tree and a set of conditions. The tree has nodes in* Var*, disjoint from* Z*, and edges labeled by a regular expression over* Σ*. We denote by* $var(W)$ *the set of variables occurring in* W*, and consider some canonical ordering*[4] *on* $var(W)$*. The conditions are (in)equalities of the form* $x = (\neq)\alpha$ *where* $x \in var(W) \cup Z$ *and* α *is in* $var(W) \cup Z \cup \mathcal{D}$.
- C *(the* `construct` *clause) is a finite tree labeled as follows:*
  - *internal nodes are labeled by expressions of the form* $f(\bar{x})$ *where* $\bar{x}$ *is a sequence of distinct variables* $x_1 \ldots x_n$ *from* $var(W) \cup Z$*,* $n \geq 0$*, and* $f \in \Sigma \cup \{x_1, \ldots, x_n\}$*;*
  - *leaf nodes are labeled by an expression as above or by another QL query* $q(\bar{x})$*, where* $\bar{x}$ *is, as above, a sequence of distinct variables from* $var(W) \cup Z$.

  Additionally, if a node is labeled by $f(\bar{x})$ *and has a child labeled* $g(\bar{y})$*, where* $g \in \Sigma \cup \{x_1, \ldots, x_n\}$*, or* $q(\bar{y})$*, where q is a QL query, then each variable in* $\bar{x}$ *must occur in* $\bar{y}$.

[4] For example the order in which variables appear in the depth first traversal of the tree.

We call $\bar{z}$ the query's free variables. Our discussion in this paper is about queries without free variables that may, however, have free variables in subqueries, hence we must consider them in our definitions. We require the construct clause C of the outermost query to have a root node labeled $f()$, with $f \in \Sigma$ (i.e. without variables).

The semantics of QL is defined as follows. We use below q and r to denote QL queries and f, g are assumed to be in $\Sigma \cup \{x_1, \ldots, x_n\}$. A QL query q (without free variables) defines a mapping from $\mathcal{T}_{\Sigma,\mathcal{D}}$ to $\mathcal{T}_{\Sigma}$. Since we need to define recursively the semantics of subqueries with free variables, let $q(\bar{z})$ be a query $\langle W, C\rangle$ and let $T = \langle t, label, val\rangle$ be an input data tree. For each mapping $\gamma : Z \rightarrow nodes(T)$ we shall define an output forest $q_\gamma(T)$, in two steps: first we define the set of bindings of W into T that extend γ, and next the construction of $q_\gamma(T)$ from the bindings (and the queries nested within C). For the topmost query, the output forest will be a tree.

Consider a query $q(\bar{z})$ with `where` clause W and `construct` clause C, a tree $T = (t, label, val)$, and $\gamma : Z \rightarrow nodes(t)$. A γ*-binding* is a mapping β from $var(W) \cup Z$ to $nodes(t)$, extending γ, such that $\beta(root(W)) = root(t)$ and for each edge from x to y in W with label r, the sequence of Σ-labels on the path from $\beta(x)$ to $\beta(y)$ in t (exclusive of $\beta(x)$ and inclusive of $\beta(y)$) spells a word in r. Additionally, the conditions on data values are satisfied. That is, if $x = (\neq)\alpha$ is a condition of W, then $val(\beta(x)) = (\neq)\ val(\beta(\alpha))$ if α is a variable, and $val(\beta(x)) = (\neq)\ \alpha$ if α is a data value. We denote by $Bind_\gamma(q, t)$ the set of γ-bindings from W to t. It will sometimes be useful to view $Bind_\gamma(q, t)$ as a relation with attributes $var(W)$, to which the usual relational algebra operations can be applied.

Note that since t is an ordered tree, the nodes in t can be totally ordered by a depth-first, left-to-right traversal. Since the variables $var(W)$ are also ordered, this induces a lexicographic ordering of the sets of bindings $Bind_\gamma(q,t)$.

The output forest $F_\gamma = q_\gamma(T)$ is be constructed from $Bind_\gamma(q,t)$ and C as follows. Each node u in C contributes to the following set of nodes in F_γ:

- if u is labeled $f(\bar{x})$ then $nodes(F_\gamma)$ includes all pairs of the form $(u, \beta(\bar{x}))$, for all $\beta \in Bind_\gamma(q,t)$. Each such node is labeled f if $f \in \Sigma$, and $label(\beta(f))$ if f is a variable in $\bar{x}$.

- if u is labeled with a subquery, $r(\bar{x})$, then $nodes(F_\gamma)$ includes all nodes of $r_{\beta|\bar{x}}(T)$, for all $\beta \in Bind_\gamma(q,t)$.

Each edge (u,v) in C contributes to a set of edges in F_γ, as follows. Let u be labeled $f(\bar{x})$.

- if v is labeled $g(\bar{y})$, then for every $\beta \in Bind_\gamma(q,t)$, the node $(v, \beta(\bar{y}))$ is a child of $(u, \beta(\bar{x}))$. The children of u are ordered by the bindings $\beta(\bar{y})$.

- if v is labeled $r(\bar{z})$, then for every $\beta \in Bind_\gamma(q,t)$ the roots of the forest $r_{\beta|\bar{y}}(T)$ are children of $(u, \beta(\bar{x}))$, ordered by the bindings $\beta \mid \bar{y}$.

Example 2.3 Consider XML documents holding information about movies (titles, directors, actors, and possibly reviews), described by the (partial) DTD: $root \to movie^*$; $movie \to title.director.review^*$; $title \to actor^*$; $actor \to name.\Sigma^*$; $director \to \epsilon$; $review \to \epsilon$. Figure 1 represents a QL query collecting the titles of movies by W.Allen, their actors (grouped under title), all available information about each actor (grouped under the actor with the same tags as in the input), and the reviews, if any. Note that a title cannot appear in the answer if there is no actor in it, because the `where` clause of the query requires an actor. However, a title does appear even if there is no review for it. This is achieved by collecting the reviews for each title using the nested query Q1.

Remark Note that our definition does not address how data values are to be included in the output tree. For instance, in Example 2.3 it would make sense to extract the actual values of titles, text of reviews, etc. There are many ways to specify this, but the issue is irrelevant to our investigation since the DTDs we consider do not restrict data values. Our framework can be augmented with any mechanism for producing data values in the output without affecting our typechecking results.

The typechecking problem Given an input DTD τ_1, an output DTD τ_2 (possibly specialized) and a query q, we say that q *typechecks* (with respect to τ_1 and τ_2) iff $q(inst(\tau_1)) \subseteq inst(\tau_2)$. We will show in Section 4 that typechecking for the full QL and unrestricted regular DTDs is undecidable. Therefore, we are led to consider restricted decidable cases.

3 Decidability Results

We present in this section our decidability results on typechecking QL queries, under various restrictions on QL and output DTDs. There are three main decidability results, involving increasingly *restricted* fragments of QL and increasingly *powerful* output DTDs:

1. non-recursive QL (QL using only non-recursive regular path expressions), and unordered output DTDs;

2. non-recursive QL without tag variables in the `construct` clause, and star-free output DTDs;

3. non-recursive QL without tag variables and no "projection", and regular output DTDs.

This highlights an interesting trade-off between the query language and the DTDs. The undecidability results of Section 4 show that our decidability results are quite tight.

Our first result concerns non-recursive QL and unordered output DTDs.

Theorem 3.1 *The typechecking problem for non-recursive QL queries, regular input DTDs, and unordered output DTDs is decidable in* EXPSPACE.

Proof: The decidability is shown by bounding the size of inputs that need to be checked to detect a violation of the output DTD. Let q be a non-recursive QL query, τ_1 a regular input DTD, and τ_2 an unordered output DTD. Suppose $T \in inst(\tau_1)$ and $q(T) \not\models \tau_2$. We show that there exists $T_0 \in inst(\tau_1)$ with at most exponentially many nodes (w.r.t. q, τ_1, τ_2), and $q(T_0) \notin inst(\tau_2)$. We construct T_0 from T as follows. Since $q(T)$ violates τ_2, there exists a path $n_1 \ldots n_k$ from the root of $q(T)$ to a node n_k with tag $a \in \Sigma$ such that the sequence w of children of n_k violates the SL formula φ_a specified by τ_2. Thus, w satisfies $\neg\varphi_a$. Clearly, $\neg\varphi_a$ can be written as $\bigvee_l C_l$ where each C_l is of the form $a_1^{*_1 i_1} \wedge \ldots \wedge a_h^{*_h i_h}$ where the a_j are distinct symbols in Σ, $*_j \in \{\geq, =\}$ and each i_j is an integer bounded by the maximum integer occurring in φ_a. Since w satisfies $\neg\varphi_a$, it satisfies at least one C_l, say $a_1^{*_1 i_1} \wedge \ldots \wedge a_h^{*_h i_h}$. Let $v_1 \ldots v_n$ be a subsequence of w that satisfies $C_l^{=} = a_1^{= i_1} \wedge \ldots \wedge a_h^{= i_h}$. Each node on the path $n_1 \ldots n_k$ from root to n_k, and

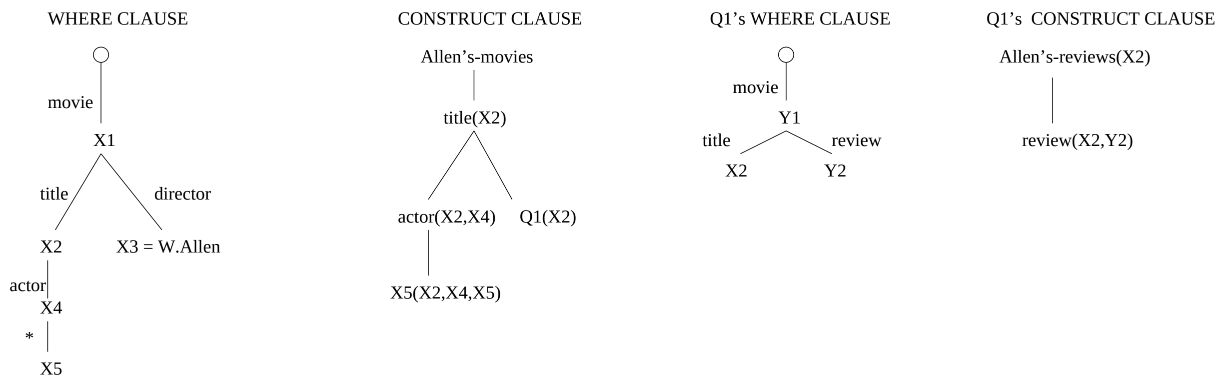


Figure 1: The Woody Allen query

among the $v_1 \ldots v_n$, arises from some binding of a subset of $var(q)$. Let B consist of the set of nodes of T in the images of these bindings, or on some path from the root of T to such a node. Note that $|B| \leq |q|^2(|q|+|\tau_2||\Sigma|)$ ($k \leq |q|, n \leq |\tau_2||\Sigma|$, the number of nodes in each binding is $\leq |q|$, and the number of nodes from root to each node in a binding is bounded by $|q|$)[5]. We will use the following observation. Let T' be any tree whose restriction to depth up to $|q|$ is a subtree of T, and that contains all nodes in B. Note that $q(T')$ still contains the node n_k and its children $v_1 \ldots v_n$, since all needed bindings are still present in T'. Thus, the sequence v of children of n_k in $q(T')$ satisfies $a_1^{\geq i_1} \wedge \ldots \wedge a_h^{\geq i_h}$ Furthermore, if $*_j$ is equality, then v satisfies $a_j^{=i_j}$, since every binding of q in T' is also a binding of q in T, and the sequence of children of n_k in $q(T)$ satisfies $a_j^{=i_j}$. It follows that v satisfies C_l and so $q(T')$ violates τ_2.

Let T_0 be a minimal tree whose restriction to depth up to q is a subtree of T, that contains all nodes in B, and that satisfies τ_1. Since T_0 satisfies the property just described, $q(T_0)$ violates τ_2. An upper bound on the number of nodes in T_0 is found as follows. For each node n in T_0, consider the sequence of its children written as $u_1 b_1 u_2 \ldots b_m u_{m+1}$ where $b_i \in B$, $1 \leq i \leq m$, and u_j contains no nodes in B, $1 \leq j \leq m+1$. Since T_0 is minimal, the size of each u_j is bounded by $|\tau_1|$ (more precisely by the number of states in the automaton for the regular language describing the allowed sequences of children of n in τ_1). Since nodes in B occur in T at depth at most $|q|$, the number of nodes of T_0 at depth up to $|q|$ is bounded by $[(|B|+1)|\tau_1|]^{|q|}$. Finally, observe that paths of T_0 whose nodes are at depth larger than $|q|$ contain no paths with repeated labels in Σ (otherwise we could pump down T_0 thus contradicting its minimality). Thus, the number of nodes at depth $> |q|$ is bounded by the maximum number of nodes at depth $\leq |q|$ times $|\tau_1|^{|\Sigma|}$. This yields a total bound of $[(|B|+1)|\tau_1|]^{|q|} \times (1 + |\tau_1|^{|\Sigma|})$ for the size of T_0. In summary, T_0 satisfies τ_1, $q(T_0)$ violates τ_2, and the size of T_0 is polynomial in $|\tau_1|$ and $|\tau_2|$ and exponential in $|q|$ and $|\Sigma|$. Also, computing $q(T_0)$ is polynomial in T_0 and PSPACE in q. This yields the decidability result and an EXPSPACE upper bound for the complexity of typechecking.

□

Remark A careful examination of the analysis in the proof of Theorem 3.1 provides a more refined complexity bound for typechecking q with respect to input DTD τ_1 and output unordered DTD τ_2. The complexity is EXPSPACE with respect to $|q|$, the maximum size of a regular expression in τ_1, and the maximum integer used in τ_2.

Our second decidability result, corresponding to (2) above, further restricts the QL queries but extends the output DTDs.

Theorem 3.2 *The typechecking problem for non-recursive QL queries without tag variables in the* `construct` *clause, regular input DTDs, and star-free output DTDs, is decidable in* EXPSPACE.

Proof: The proof is by reduction to the case when the output DTD is unordered, for which we can use Theorem 3.1. The key observation is the following:

(†) if r is a star-free regular expression and $a_1, \ldots, a_n$ are distinct symbols in Σ, then there exists an SL sentence σ, using integers whose size (in unary representation) is linear in r and computable in EXPTIME from r and $a_1, \ldots, a_k$, such that

$$r \cap a_1^* \ldots a_k^* \;=\; L(\sigma) \cap a_1^* \ldots a_k^*.$$

[5] The inequality $n \leq |\tau_2||\Sigma|$ assumes the integers in τ_2 are written in unary.

The proof of (†) is by a straightforward induction on the structure of r. Now consider a non-recursive QL query q without tag variables in the `construct` clause, and a star-free output DTD τ_2. Suppose first that

(∗) all siblings in the `construct` clause of q have *distinct* tags

(we eliminate this restriction below). Since q has no tag variables, all sequences of sibling nodes in answers to q are of the form $a_1^* \ldots a_k^*$ for distinct $a_i \in \Sigma$. From (†) it follows that q typechecks with respect to τ_2 iff it typechecks with respect to τ_2' for an unordered DTD τ_2' computable in EXPTIME from τ_2. The decidability and EXPSPACE upper bound then follow from Theorem 3.1. The fact that τ_2' is exponential in τ_2 is not a problem since the size of the integers used in τ_2' is only linear in the size of τ_2 (see also Remark following Theorem 3.1).

Suppose now that q violates (∗). We construct from q a query $\bar{q}$ by replacing in the `construct` clause all tags $a_1, \ldots, a_k$ by *distinct* $b_1, \ldots b_k$, and we use the following variant of (†):

(‡) Let $a_i, b_i \in \Sigma$, $i \in [1, k]$, where the b_i are distinct, and let h be the homomorphism mapping b_i to a_i. If r is a star-free regular expression then there exists an SL sentence σ, using integers whose size (in unary representation) is linear in r and computable in EXPTIME from r, $a_1, \ldots, a_k$, and $b_1, \ldots, b_k$, such that

$$r \cap a_1^* \ldots a_k^* \;=\; h(L(\sigma) \cap b_1^* \ldots b_k^*).$$

Using (‡) it is clear that q typechecks with respect to τ_2 iff $\bar{q}$ typechecks with respect to an unordered DTD $\bar{\tau}_2$ computable in EXPTIME from τ_2. The decidability and EXPSPACE upper bound follow as above. □

Our third decidability result removes all restrictions on output DTDs, allowing full regular DTDs. However, it requires an additional restriction on QL queries. Intuitively, this limits the projections which can be performed by the query. We formalize this as follows.

Definition 3.3 *Let q be a QL query and τ an input DTD. The query q is* projection free *with respect to τ iff it is equivalent, on all inputs satisfying τ, to the query obtained from q by replacing, in each nested query $\bar{q}(\bar{z}) =< \bar{W}, \bar{C} >$ of q (including q itself), each node $f(\bar{x})$ in $\bar{C}$ by $f(var(\bar{W}) \cup \bar{z})$.*

Intuitively, this means that for every node $f(\bar{x})$, the bindings of all the variables not in $\bar{x}$ functionally depend on those of $\bar{x}$. Obviously, there are many syntactic sufficient conditions on q and τ ensuring that q is projection-free w.r.t. τ. We illustrate this by an example.

Example 3.4 Consider the *cinema* DTD in Example 2.3. The query in the same example is *not* projection free (for example, the node *title(X2)* is a violation since, for one given title, there may be several actor children bound to $X4$. Replacing *title(X2)* by *title(X4)* will thus yield a different output). On the other hand, consider the query q in Figure 2. The query collects the actors in W.Allen movies, together with the title of the movie. Additionally, the nested query q' produces for each such actor all other titles (not by W.Allen) where the actor acts. This query is projection free: Since every *actor* node has a unique *title* parent, using *actor(X3,X2)* rather than *actor(X3)* yields no difference. Also, when the nested query is evaluated, the binding for $X3$ is fixed. There is only one ancestor *title* and *movie* nodes for any given $Y3$, and similarly, for each binding of $Y1$ there is a unique *director* child $Y4$. So *title(X3,Y2,Y3)* could be replaced by *title(X3,Y1,...,Y4)*.

We can show the following result, the most technically challenging of this section.

Theorem 3.5 *The typechecking problem for projection-free non-recursive QL queries without tag variables, regular input DTDs, and regular output DTDs, is decidable.*

Proof: Decidability is shown by bounding the size of inputs that need to be checked to detect a violation of the output DTD. The proof uses Ramsey's Theorem [13, 21] and is quite intricate. Due to space limitations, we present the proof in Section A of the appendix. □

It remains open whether Theorem 3.5 holds without the projection-free restriction.

Complexity Theorems 3.1 and 3.2 provide an upper bound of EXPSPACE on the complexity of typechecking non-recursive QL queries with respect to unordered output DTDs, or non-recursive QL queries without tag variables and star-free output DTDs. However, it remains open whether this complexity is tight.

There are significant special cases in which the complexity of typechecking can be brought down to PSPACE. We consider the case when the input DTDs are of bounded depth (which implies that queries are also of bounded depth). This is a restriction of practical interest, since many applications use shallow

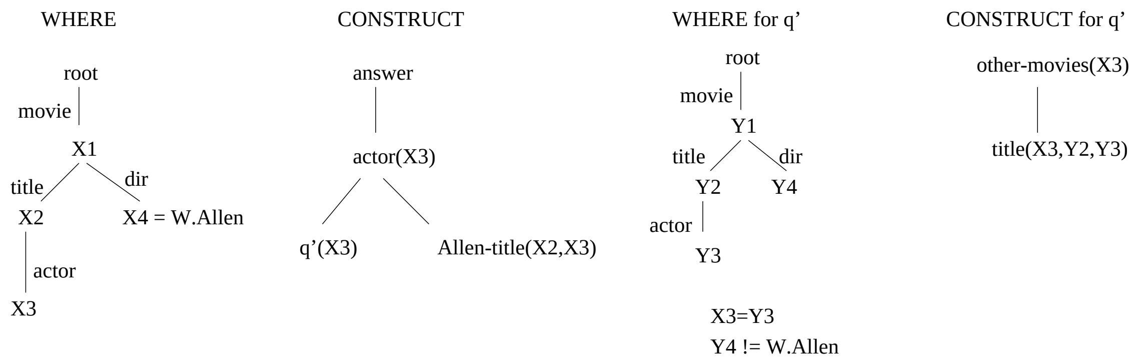


Figure 2: A projection-free query.

DTDs. For example, relational databases can naturally be represented by DTDs of depth[6] at most 2. We can show the following using the proofs of Theorems 3.1 and 3.2.

Corollary 3.6 *Let Σ and $M > 0$ be fixed. (i) Typechecking non-recursive QL queries with respect to input DTDs of depth $\leq M$ and unordered output DTDs is in* PSPACE*; (ii) Typechecking non-recursive QL queries without tag variables with respect to input DTDs of depth $\leq M$ and star-free DTDs is in* PSPACE*.*

Once again, it remains open whether the above complexity is tight. However, we can show the following lower bounds.

Theorem 3.7 *Typechecking non-recursive QL queries without tag variables with respect to input DTD of depth ≤ 2, and unordered output DTDs is:*

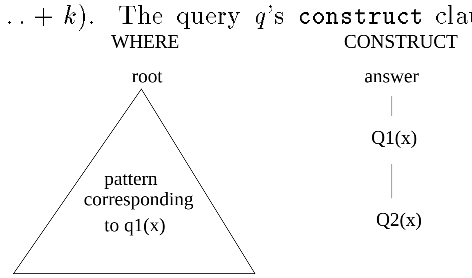
(i) CO-NP*-hard for QL queries without conditions on data values;*

(ii) DP*-hard[7] for QL queries with equalities on data values;*

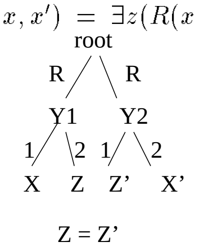
(iii) Π_2^p-hard for QL queries with equalities and inequalities on data values.

Proof: For (i), we reduce validity of propositional CNF formulas to the typechecking problem. Let φ be a CNF propositional formula using variables $x_1, \ldots, x_n$. Consider the input DTD $root \rightarrow X_1 \ldots X_n; X_i \rightarrow (zero + one),\ \ 1 \leq i \leq n; zero \rightarrow \epsilon; one \rightarrow \epsilon$. The query q is represented in Figure 3. Basically, each nested query q_i returns X_i iff X_i has a child labeled "one" in the input. (The pattern in the `where` clause of q is only used to ensure that the set of bindings of q is non-empty.) The SL formula for the output DTD is obtained from φ by replacing each positive literal x_i by $X_i^{=1}$ and each negative literal $\neg x_i$ by $X_i^{=0}$, $1 \leq i \leq n$. Clearly, φ is valid iff q typechecks w.r.t. τ_1 and τ_2.

The proof of (ii) is by simultaneous reduction of propositional validity and conjunctive query containment, which is known to be NP-complete. The query consists of the concatenation of two independent sub-queries, one corresponding to propositional validity and the other to conjunctive query containment. The subquery corresponding to propositional validity (and its corresponding SL formula) is the one described in (i). We briefly describe the subquery corresponding to conjunctive query containment. Consider two conjunctive queries $q_1(\bar{x}), q_2(\bar{y})$ over relation R of arity k. We build an input DTD τ_1, QL query q, and output unordered DTD τ_2 as follows. The DTD τ_1 is $root \rightarrow R^+$, $R \rightarrow (1 + \ldots + k)$. The query q's `construct` clause is:



The pattern in the `where` clause corresponds in the natural way to $q1(\bar{x})$. For example, the pattern corresponding to $q_1(x, x') = \exists z(R(x, z) \wedge R(z, x'))$ is:



Thus, $Q1$ occurs in the answer once for each $\bar{x}$ in the answer to q_1 on R. The query $Q2$ is a nested

[6] The root has depth zero.

[7] Recall that DP properties are of the form $\sigma_1 \wedge \sigma_2$ where $\sigma_1 \in$ NP and $\sigma_2 \in$ CO-NP.

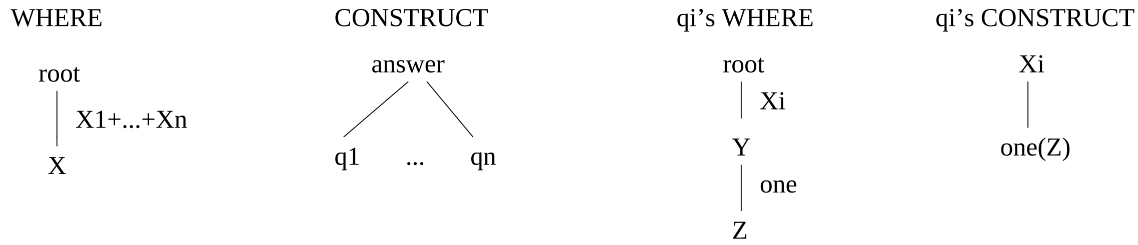


Figure 3: The query for propositional validity

query with root $Q2$ which collects bindings corresponding to $q_2(\bar{y})$, where additionally $\bar{x}$ and $\bar{y}$ have the same associated data values. Thus, $q_1 \subseteq q_2$ iff the answer to $Q2$ is non-empty for each $\bar{x}$ in the answer to q_1. The unordered output DTD is simply $answer \rightarrow Q_1^{\geq 0}, \quad Q_1 \rightarrow Q_2^{\geq 1}$.

The proof of (iii) is by reduction of containment of conjunctive queries with inequalities, which is known to be Π_2^p-complete [23]. The reduction is similar to that of (ii). □

For star-free output DTDs, we can show a PSPACE lower bound, even without conditions on data values:

Theorem 3.8 *Typechecking non-recursive QL queries without conditions on data values with respect to input DTD of depth ≤ 2 and star-free output DTDs (using FO sentences with unary integers) is* PSPACE-*complete.*

Proof: We use a reduction from the Quantified 3SAT problem, known to be PSPACE-complete [12] (see Appendix). □

4 Undecidability Results

So far, we have shown that typechecking QL queries is decidable under various restrictions on the query language and output DTD. In particular, all decidability results require *non-recursive* QL queries, and output DTDs *without specialization*. In this section we show that these restrictions are largely necessary for decidability. Specifically, we show the following:

1. allowing *specialization* in output DTDs leads to undecidability of typechecking even for highly restricted queries and DTDS (QL queries with path expressions limited to single symbols, no inequality tests on data values, no tag variables, unordered input DTDs of depth two, and unordered output DTDs);

2. nested queries can be traded in the above against disjunction in path expressions (expressions of the form a or $a+b$, for $a, b \in \Sigma$) and tag variables; and

3. allowing *recursive path expressions* in QL queries yields undecidability of typechecking even for very simple output DTDs without specialization.

In conjunction with the previous decidability results, this yields a fairly tight boundary of decidability for typechecking.

The first result shows that allowing specialization in output DTDs quickly leads to undecidability of typechecking, even under stringent assumptions. We call a QL query *conjunctive* if every path expression in the `where` clause is a single symbol in Σ.

Theorem 4.1 *Typechecking is undecidable for conjunctive QL queries without tag variables and without inequality, unordered input DTDs of depth ≤ 2, and unordered output DTDs with specialization.*

Proof: The proof is by reduction of the implication problem for FDs and inclusion dependencies. Let D be a set of FDs and inclusion dependencies over some k-ary relation R, and f an FD over R. We construct an input DTD τ_1, a QL query q, and an output DTD τ_2 (satisfying the restrictions in the statement) such that q typechecks with respect to τ_1 and τ_2 iff $D \models f$. The input DTD is: $root \rightarrow R^{>0}; \quad R \rightarrow 1^{=1} \wedge \ldots \wedge k^{=1}; \quad i \rightarrow \epsilon, 1 \leq i \leq k$. Intuitively, R encodes a k-ary relation. The query q is the concatenation of several parts, each of which is used to verify (in conjunction with the output DTD) whether the dependencies in D and f are satisfied by R. We illustrate these using an example for each kind of dependency. Consider an inclusion dependency, say $R[12] \subseteq R[23]$. The corresponding portion of the query is illustrated in Figure 4. Note that $R[12] \subseteq R[23]$ is satisfied iff every R_{12} node has at least one child labeled R_{23}. Now consider an FD, say $1 \rightarrow 2$. The corresponding portion of the query is represented in Figure 5. Note that the FD is satisfied iff every *pair* node has at least one child labeled *eq*. Clearly, one can state using a specialized unordered DTD that either some dependency in D is violated, or f is satisfied. Thus, q typechecks iff $D \models f$. □

Similar undecidability results can be shown for slightly different combinations of features, which

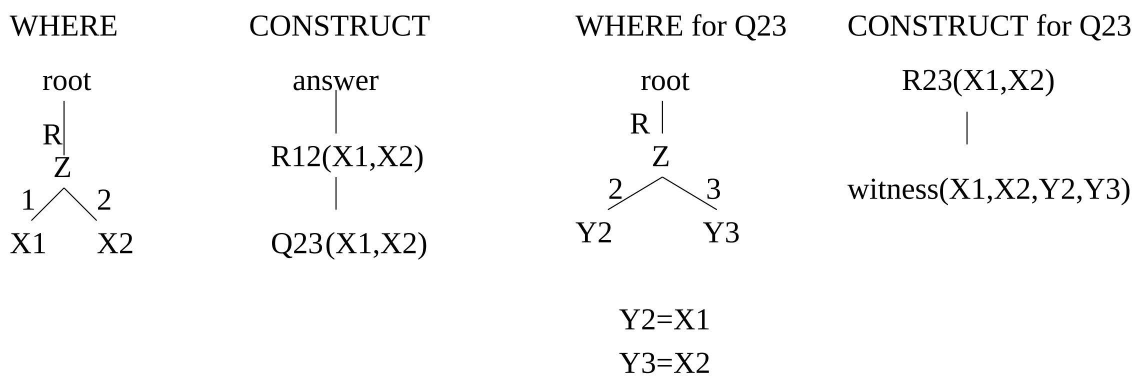


Figure 4: Query for inclusion dependency

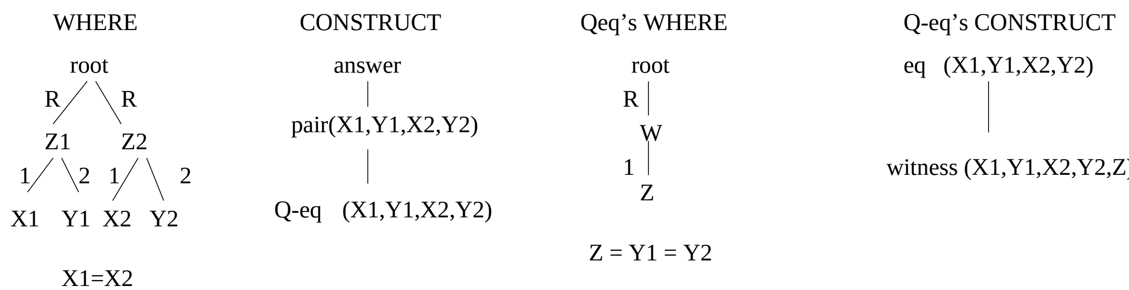


Figure 5: Query for functional dependency

highlight rather subtle trade-offs. For example, one might wonder if nested queries are essential to undecidability. The following result shows that they are not: nested queries can be traded against disjunctions in path expressions and tag variables in the `construct` clause. We call a QL program *disjunctive* if the path expressions in the `where` clause are of the form a or $a+b$ where a, b are single symbols.

Proposition 4.2 *Typechecking is undecidable for disjunctive QL queries with tag variables, without inequality, and without nested queries, unordered input DTDs of depth* ≤ 2*, and unordered output DTDs with specialization.*

Proof: The proof is again by reduction from the implication problem for fds and inclusion dependencies (see Appendix). □

All the decidability results of Section 3 assume nonrecursive QL queries. We next show that removing this restriction immediately causes undecidability of typechecking, even with very simple output DTDs.

Theorem 4.3 *Typechecking is undecidable for QL queries and any output DTD that requires a nonempty sequence of children under the root.*

Proof: We reduce the Post Correspondence Problem (PCP) to typechecking a QL query with respect to an output DTD requiring a nonempty sequence of children under the root (see Appendix). □

5 Conclusions

The main contribution of the present paper is to shed light on the feasibility of typechecking XML queries that make use of data values in XML documents. The results trace a fairly tight boundary of decidability of typechecking. In a nutshell, they show that typechecking is decidable for XML-QL-like queries without recursion in path expressions, and output DTDs without specialization. As soon as recursion or specialization are added, typechecking becomes undecidable.

The decidability results highlight subtle trade-offs between the query language and the output DTDs: decidability is shown for increasingly *powerful* output DTDs ranging from unordered and star-free to regular, coupled with increasingly *restricted* versions of the query language. Showing decidability is done in all cases by proving a bound on the size of counterexamples that need to be checked. The technical machinery required becomes quite intricate in the case of regular output DTDs and involves a combinatorial argument based on Ramsey's Theorem. For the decidable cases we also consider the complexity of typechecking and show several lower and upper bounds.

The undecidability results show that specialization in output DTDs or recursion in queries render typechecking unfeasible. If output DTDs use specialization, typechecking becomes undecidable even under very stringent assumptions on the queries and DTDs. Similarly, if queries can use recursive path expressions, typechecking becomes undecidable even for very simple output DTDs without specialization.

Several questions are left for future work. We showed decidability of typechecking for regular output DTDs and queries restricted to be *projection free*. It is open whether the latter restriction can be removed. With regard to complexity, closing the remaining gaps between lower and upper bounds remains open.

Beyond the immediate focus on typechecking, we believe that the results of the paper provide considerable insight into XML query languages, DTD-like typing mechanisms for XML, and the subtle interplay between them.

References

[1] 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/`.

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

[3] 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/`.

[4] J. R. Büchi. Weak second-order arithmetic and finite automata, Z. Math. Logik Grundl. Math., vol. 6, pp. 66–92, 1960.

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

[6] D. Chamberlin, J. Robie, and D. Florescu Quilt: An XML Query Language for Heterogeneous Data Sources, `WebDB 2000, (to be published in Lecture Notes in Computer Science, Springer-Verlag, 2000)`.

[7] J. Clark. Xml path language (xpath), 1999. `http://www.w3.org/TR/xpath`.

[8] S. Cluet, C. Delobel, J. Simeon and K. Smaga. Your mediators need data conversion! In *Proc. ACM-SIGMOD Int'l. Conf. on Management of Data*, pp. 177-188, 1998.

[9] V. Christophides, S. Cluet and J. Simeon. On Wrapping Query Languages and Efficient XML Integration. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pp. 141-152, 2000

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

[11] H.-D. Ebbinghaus and J. Flum. Finite Model Theory, Springer, second edition, 1999.

[12] M.R.Garey and D.S.Johnson. Computers and Intractibilitiy: A Guide to the Theory of NP-Completeness. Freeman, San Francisco, 1979.

[13] R.L. Graham, B.L. Rothschild and J. H. Spencer, *Ramsey Theory* (Second Edition), Wiley, New York, 1990.

[14] XDuce: An XML Processing Language (Preliminary Report). H. Hosoya and B. C. Pierce, WebDB'2000.

[15] Regular Expression Pattern Matching for XML. H. Hosoya and B. C. Pierce. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2001

[16] R. Milner, M. Tofte and R. Harper. The Definition of Standard ML. MIT Press, 1990.

[17] T. Milo and D. Suciu. Type Inference for Queries on Semistructured Data. In *Proc. ACM Symposium on Principles of Database Systems*, pp. 215–226, 19 99.

[18] T.Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 11-22, 2000.

[19] F. Neven and T. Schwentick. XML Schemas without Order. Unpublished manuscript, 1999.

[20] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 35-46, 2000.

[21] F. P. Ramsey, On a problem of formal logic, *Proceedings of the London Mathematical Society*, 30(2):264–286, 1929

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

[23] R. van der Meyden. The complexity of querying infinite data about linearly ordered domains *JCSS*, 54(1):113-135, 1997.

[24] W. Thomas. Languages, automata, and logic. In *Handbook of Formal Languages*, eds. G. Rozenberg and A. Salomaa, vol.3, ch.7, Springer, 1997.

Appendix

A Proof of Theorem 3.5

The proof is based on a corollary of Ramsey's Theorem [13, 21] (the theorem and corollary are provided below). Decidability is shown by bounding the size of inputs that need to be checked to detect a violation of the output DTD.

Assume we are given some projection-free nonrecursive QL query q'. By the definition of *projection free*, q' is equivalent to some query q obtained from q' by replacing in each nested query $\bar{q}(\bar{z}) = < \bar{W}, \bar{C} >$ of q' (including q' itself), each node $f(\bar{x})$ in $\bar{C}$ by $f(var(\bar{W}) \cup \bar{z})$. So it suffices to prove the theorem for q. Without loss of generality, we can assume that all the path expressions in q are single labels or disjunctions of such labels. (If not we add extra variables).

Recall that for a query $q(\bar{x}) = \langle W, C \rangle$, $var(W)$ denotes the set of variables in W, and $Bind\_gamma(q, T)$ denotes the set of γ-bindings from W to T. For the discussion below we need to generalize these notions and then apply them to nodes in the (nested) construct clauses of the query. We also to define the auxiliary notions of node *tags* and *variables*.

Definition A.1 *Let $q_0 = \langle W_0, C_0 \rangle$ be a QL query. Let c be a node in (a possibly nested) construct clause C_i of q_0, with $q_0 = \langle W_0, C_0 \rangle, \ldots, q_i = \langle W_i, C_i \rangle$ being the (nested) queries on the path from the root query q_0 to C_i. Let T be some input tree.*
We define $var^*(q_l) = \Sigma_{l=0\ldots l} var(W_l)$
We define $Bind^*(q_l, T)$, $l = 0 \ldots i$, as follows.
$Bind^*(q_0, T) = Bind_{\emptyset}(q_0, T)$
$Bind^*(q_i, t) = \bigcup \{Bind_{\gamma}(q_{i-1}, T) \mid \gamma \in Bind^*(q_{i-1}, T)\}$
Finally, for the node c, $vars(c)$ and $Binds(c, T)$ are defined as follows. When c has a label of the form $f(\bar{x})$, $vars(c) = var^*(q_i)$ and $Binds(c, T) = Bind^*(q_i, T)$. Otherwise, when c is labeled by some nested query $q_{i+1}(\bar{x})$, $vars(c) = var^*(q_{i+1})$ and $Binds(c, T) = Bind^*(q_{i+1}, T)$.

Definition A.2 *For a node c in the construct clause, with a label of the form $f(\bar{x})$, we call f the* tag *of c, and call $\bar{x}$ the* variables *of c. For a node c labeled by some nested query q_i, the* tag *of c (resp. the* variables *of c) is the tag (resp. variables) of the root note in the construct clause of q_i.*

To simplify the presentation, we will assume first that, for every node c in the construct clause of q, the tags of c's children are all distinct. The general case where some tags may repeat is considered afterwards.

Let τ_1, τ_2 be input and output DTDs respectively. Suppose $T \in inst(\tau_1)$ and $q(T) \not\models \tau_2$. Since $q(T)$ violates τ_2, there exists a path $n_1 \ldots n_k$ from the root of $q(T)$ to a node n_k with some tag $a \in \Sigma$ such that the sequence w of children of n_k violates the regular expression φ_a specified by τ_2. Note that n_k was constructed due to some node c in some (possibly nested) *construct* clause of q. If $a_1, \ldots, a_n$ are the tags of the children of c in the construct clause, this w satisfies the language $\hat{\varphi_a} = \neg\varphi_a \cap a_1^* \ldots a_n^*$. It is easy to see that $\hat{\varphi_a}$ is in fact a union of languages, each described by a vector of n triplets of natural numbers $(k_1, i_1, j_1), \ldots, (k_n, i_n, j_n)$, restricting the number of the a_ls, $l = 1 \ldots n$, in every word to be $k_l + i_l \ mod \ j_l$, when $j_l > 0$, and be exactly k_l when $j_l = 0$.

The following immediately follows.

Proposition A.3 *Let q be a QL query, let T be some input tree, and let τ_2 be some output DTD. $q(T)$ violates τ_2 iff there exists, in the construct part of q, a node c with some tag a and children $c_1, \ldots, c_n$ having tags $a_1, \ldots, a_n$ and variables $\bar{x}_1, \ldots, \bar{x}_n$, and a vector $(k_1, i_1, j_1), \ldots, (k_n, i_n, j_n)$ in $\hat{\varphi_a} = \neg\varphi_a \cap a_1^* \ldots a_n^*$, s.t. $Binds(c, T)$ contains a tuple v where for all $l = 1 \ldots n$, $\pi_{\bar{x}_i}(\sigma_{vars(c)=v}(Binds(c_i, T)))$ is of size $k_l + i_l \ mod \ j_l$ when $j_l > 0$, and of size k_l when $j_l = 0$.*

For brevity, we will refer in the sequel to the property of $Binds(c, T)$ containing a tuple v s.t. for all $l = 1 \ldots n$, $\pi_{\bar{x}_i}(\sigma_{vars(c)=v}(Binds(c_i, T)))$ is of size $i_l \ mod \ j_l$ when $j_l > 0$, and of size i_l when $j_l = 0$, as the *modulo property*.

It thus remains to show that, given an input DTD τ_1, a query q, some node c in the `construct` part, and a vector $(i_1, j_1), \ldots, (i_n, j_n)$ for c as defined above, one can decide if the modulo property holds for some tree $T \in inst(\tau_1)$.

To prove this it suffices to show that if such a T exists, then there exists a "small" such T, namely a tree T whose size is bounded by some given function of the sizes of the τ_1, q, and the numbers $(k_1, i_1, j_1), \ldots, (k_n, i_n, j_n)$. Then, to decide if the above holds we simply need to check all the trees up to that size. We will show below that such a bound on the tree size indeed exists.

Without loss of generality, assume that τ_1 contains no redundant symbols. First observe that if none of the regular expressions in τ_1 contains $*$ then it suffices to look at trees of size $O(\tau_1^{|q|})$. This is because q looks at paths of of a bounded length. Thus all we need to check are trees of the corresponding maximal depth and with a bounded width. (The actual violating instance can then be obtained by replacing each of the tree leaves by some arbitrary derivation tree for the leaf label.) We thus assume in the sequel that at least some of the regular expressions in τ_1 contain $*$.

Next, note that T may be large due to its depths or to its width. Since the query looks at a bounded depth in T, all nodes beyond depth $|q|$ are essentially irrelevant. So we only need to look at trees up to depth q. (As above, the actual violating instances can later be obtained from these trees by replacing each of the tree leaves by some arbitrary derivation tree for the leaf label.) For brevity, we will abuse below the standard terminology and from now on whenever we say a *tree* we mean a tree of depth $\leq |q|$. Whenever we say that *a tree T satisfies* τ_1 we mean that T can be extended, by adding nodes only in depth greater than $|q|$ to a tree that satisfies τ_1.

Let $T \in inst(\tau_1)$ be the smallest tree having the modulo property (or one of them, if several such trees exist). Let N be the set of nodes in T consisting of the following.

1. all the nodes in the vector v,
2. for every c_i where the relation $R_i = \sigma_{vars(c)=v}(Binds(c_i, T)))$ is not empty, the nodes in some vector $r_i \in R_i$,
3. for every c_i with corresponding triplet (k_i, i_i, j_i), all the nodes in some sub-relation of of size k_i of $\pi_{\bar{x}_i}\sigma_{vars(c)=v}(Binds(c_i, T)))$, and
4. all the nodes on the paths from the root to the nodes in items 1-3 above.

Note that the size of N is bounded by $(|q| + \tau^{|q|}) \times |q|$. Our goal is to show that if T is bigger than a given size then it contains a set of nodes x, not including any of the nodes in N, such that the tree T' obtained from T by removing x still belongs to $inst(\tau_1)$ and has the modulo property. This will contradict the minimality of T.

Clearly if x is not chosen carefully the resulting tree may not belong anymore to $inst(\tau_1)$. We will thus be interested only in deletion of nodes that leave the tree in τ_1. For that we define the notion of a *deletable unit.*

Definition A.4 *Given a tree $T \in inst(\tau_1)$ and a node n in T, a consecutive subsequence $\bar{n}$ of n's children is called a* deletable unit *if (i) the tree T' obtained from T by deleting $\bar{n}$ (and the subtrees rooted at them) still belongs to $inst(\tau_1)$, and (ii) $\bar{n}$ does not contain a sub sequence having the above property.*

We can show the following.

Proposition A.5 *Let U be a maximal devision of T into non-overlapping deletable units, not containing any of the nodes in N. The number of the deletable units in U is no less than $|T|/((|\tau_1| \times (|N| + 1))^{|q|})$.*

Proof: (Sketch) Note that by the pumping lemma the size of each deletable unit is at most $|\tau_1|$, and the number of siblings separating between any two consecutive deletable units is also at most $|\tau_1|$. Similarly, a node in N has a sequence of at most $|\tau_1|$ following(proceeding) siblings not including any node which is either in N or belongs to some deletable unit. Finally note that if a sibling not in a deletable unit has more than $\tau_1 \times |N|$ children then at least some of them must belong to a deletable unit. Combining all these we get the above lower bound. □

Corollary A.6 *For every number x, each tree $T \in inst(\tau_1)$ of size larger than $x \times ((|\tau_1| \times (|N|+1))^{|q|})$ has at least x deletable units not including any of the nodes in N.*

We will use this corollary below to derive the required bound on the size of T. In particular we will show that if T is larger than a given size then it contains enough deletable units so that some can be removed without effecting the modulo property.

For every deletable unit $u \in U$ we associate a vector $t_u = (t_u^1, \ldots t_u^n)$ where t_u^l, $l = 1 \ldots n$, is the number of vectors in $\pi_{\bar{x}_l}(\sigma_{vars(c)=v}(Binds(c_l, T)))$ that contain some node in u, modulo j_l.

Similarly, for every set of deletable units $s \subset U$ of size $k \leq |q|$ we associate a vector $t_s = (t_s^1, \ldots t_s^n)$ where for all $l = 1 \ldots n$, t_s^l is the number of vectors $\pi_{\bar{x}_l}(\sigma_{vars(c)=v}(Binds(c_l, T)))$ that contain some node in s, modulo j_l.

From Ramsey's theorem (which will be given below), and in particular from Corollary A.8, it follows that for every number x, if T is larger than some given bound (a function of the number x, the given $j_l's$, $|\tau_1|$, and $|q|$) then there exist a set X of x deletable units in T, not including any of the nodes in N s.t. for every number $r \leq |x|$, all subsets of X of size r have the same associated vector (there may be different vectors for different r's, but all subsets of the same size r have the same vector).

Let $x = \Pi_{l=1 \ldots n} j_l \times k!$ (where $k = |q|$). If T is larger than the above Ramsey bound, then it contains a set X of x deletable units with the above property. Now, consider the tree T' obtained from T by removing all the nodes in X. Note that T' still satisfies τ_1, and since none of the nodes in N was deleted, the following hold.

- $v \in Binds(c, T')$.
- For every c_i where, in the corresponding (k_i, i_i, j_i) triplet, $j_i = 0$, $\pi_{\bar{x}_i}(\sigma_{vars(c)=v}(Binds(c_i, T')))$ contains exactly i_i tuples.
- For every c_i where $\bar{x}_i$ is the empty vector $\pi_{\bar{x}_i}(\sigma_{vars(c)=v}(Binds(c_i, T'))) = \pi_{\bar{x}_i}(\sigma_{vars(c)=v}(Binds(c_i, T)))$. (Namely, the relations are either both empty of both contain a single 0-ary tuple.) [8]

To show that T' has the modulo property, it remains to prove that for every c_i where $\bar{x}_i$ is not the empty vector, the relations $\pi_{\bar{x}_i}(\sigma_{vars(c)=v}(Binds(c_i, T')))$ is of size $i_i \ mod \ j_i$. Rather than computing the exact size of this relation, we will compute the number of vectors deleted from $\pi_{\bar{x}_i}(\sigma_{vars(c)=v}(Binds(c_i, T)))$. If we show that this number, modulo j_i, is zero then we are done.

Observe that since the query is projection free, each deleted node affects precisely the tuples in $\pi_{\bar{x}_i}(\sigma_{vars(c)=v}(Binds(c_i, T)))$ in which it appears. The total effect of the deletion of all the nodes in x is described by following *inclusion-exclusion* formula which we explain next.

$$n_l = x \times c_l^1 - \binom{x}{2} \times c_l^2 + \binom{x}{3} \times c_l^3 - \ldots +/- \binom{x}{|X_l|} \times c_l^{|X|}$$

The formula first counts separately for each deletable unit in x how many tuples where deleted when the node is removed. Note that according to the conclusion from Ramsey's theorem mentioned above, the number of images destroyed for each node are the same - this is essentially the value of the l entry in the vectors associated with subsets of X of size 1. Let c_l^1 be this number. Thus the total sum of destroyed images is $x \times c_l^1$. Now, note that by the above we counted a bit too much: some tuples, (i.e. those containing two or more nodes in x) where counted several times. To fix this we deduct for every pair of nodes the number of tuples in which the two nodes both appear. (There are $\binom{x}{2}$ such pairs of nodes, each appearing, again according to the conclusion of Ramsey Theorem, in c_l^2 images, (where c_l^2 is the value of the l entry in the vectors associated with subsets of X of size 2). So we deduct $\binom{x}{2} \times c_l^2$. Now, note again that we deducted too much: some tuples (i.e. those containing 3 or more nodes in x) where counted several times. To fix that we add for every triplet of nodes the number of tuples in which the three nodes all appear. As above this is $\binom{x}{3} \times c_l^3$, for some constant c_l^3. Now again we added too much so we deduct for the four-or-more images, etc. Since the maximal number of nodes in a tuple is bounded by $|q|$ the inclusion/exclusion sum can stop when we get to that size.

[8] This is because in the definition of N we picked (in item 2) a tuple from each non empty relation R_i. So, if R_i was not empty the projecting results in one tuple - the empty tuple.

Since we chose x to be the number $x = \Pi_{l=1\ldots n} j_l \times k!$ (where $k = |q|$), each element in the above sum divides by j_l, so the total number n_l of X_l assignments that we lost divides by j_l. It follows that T' still has the modulo property, which contradicts the minimality of T.

To conclude, lets see Ramsey Theorem, and a corollary of the theorem which leads to the above result.

Theorem A.7 *(Ramsey's Theorem) ([13], see also [21], pp.7-9) For every three finite natural numbers k, x, w there exists a finite number $R(k, x, w)$ s.t. for every set Y of elements of size $|Y| \geq R(k, x, w)$ and every coloring of the family of all the subsets of Y of size k with w colors, Y contains a subset $X \subset Y$ of size $|X| = x$ where all Xs subsets of size k have the same color.*

The following variant is an easy consequence of Ramsey's Theorem.

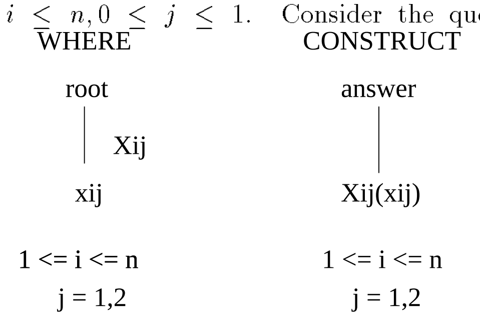
Corollary A.8 *For every three finite natural numbers k, x, w there exists a finite number $R'(k, x, w)$ s.t. for every set Y of elements of size $|Y| \geq R'(k, x, w)$ and every coloring of the family of all the subsets of Y of size $\leq k$ with w colors, Y contains a subset $X \subset Y$ of size $|X| = x$ where for all $k' \leq k$, all Xs subsets of size k' have the same color (there may be different colors for different k's).*

Now, observe that in the proof above we simply need to consider each possible vector attached to a set of deletable unites as a color. The number w of available colors then is simply $j_1 \times \ldots \times j_n$. The $k = |q|$, and $x = \Pi_{l=1\ldots n} j_l \times k!$. Now, recall from Corollary A.6 that every tree T of size larger than $R'(k, x, w) \times ((|\tau_1| \times (|N| + 1))^{|q|})$ has at least $R'(k, x, w)$ deletable units not including any of the nodes in N. The rest follows immediately from Corollary A.8.

Remark: Recall that we assumed at the beginning of the proof that, for every node c in the construct clause of q, the tags of c's children are all disjoint. To conclude the proof we need to consider the case of redundant tags. Otherwise, let a be the tag of c and $a_1, \ldots, a_k$ the tags of its children. As in the proof of Theorem 3.2 we construct from q a query $\bar{q}$ by replacing in the `construct` clause all tags $a_1, \ldots, a_k$ by *distinct* $b_1, \ldots b_k$, and replace the regular expression φ_a by $h^{-1}(\varphi_a) \cap b_1^* \ldots b_k^*$ where h is a mapping that maps each b_i to a_i. It is easy to see that the new query typechecks w.r.t to the modified DTD iff the original one does.

B Other Proofs

Proof of Theorem 3.8 We use a reduction from the Quantified 3SAT problem, known to be PSPACE-complete [12]. Let $\psi = Q_1 x_1 \ldots Q_n x_n \varphi(x_1, \ldots, x_n)$ where $Q_i \in \{\forall, \exists\}, 1 \leq i \leq n$, and φ is an instance of 3SAT (conjunction of disjunctions of 3 literals) using variables among $x_1, \ldots, x_n$. Consider the input DTD: *root* $\to X_{10}X_{11} \ldots X_{n0}X_{n1}$, $X_{ij} \to \epsilon$, $1 \leq i \leq n, 0 \leq j \leq 1$. Consider the query q:



Clearly, q simply copies the input, outputting the string $X_{10}X_{11} \ldots X_{n0}X_{n1}$ of length $2n$. Now consider the output DTD describing the children of *answer* by the sentence $\bar{\psi} = Q_1 i_1 \in \{1, 2\} \ldots Q_j i_j \in \{2j-1, 2j\} \ldots Q_n i_n \in \{2n-1, 2n\}\bar{\varphi}$ where $\bar{\varphi}$ is obtained from φ by replacing each positive literal x_j by $i_j = 2j$ and each negative literal $\neg x_j$ by $i_j = 2j - 1$, and all integers are unary. Clearly, $\bar{\psi}$ is polynomial in ψ and $\bar{\psi}$ is satisfied by strings of length $2n$ iff ψ is true. Since q outputs a string of length $2n$, it follows that q typechecks iff ψ is true.

Proof of Proposition 4.2 We use again a reduction from the implication problem for fds and inclusion dependencies. We illustrate the reduction using an example. Suppose R is a k-ary relation, $D = \{R[12] \subseteq R[23], 12 \to 3\}$, and $f = 3 \to 1$. The idea of the reduction is similar to that for Theorem 4.1 except that we represent explicitly projections of R on subsets of its attributes involved in the fds or incds. For the example, we use the following input DTD:

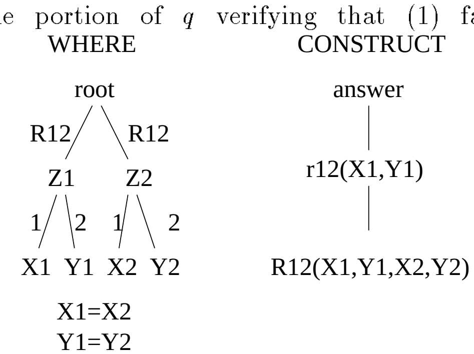
$$
\begin{array}{l}
root \to R^{>0} \wedge R_{12}^{>0} \wedge R_{23}^{>0} \wedge R_{13}^{>0} \wedge R_{123}^{>0} \\
R \to 1^{=1} \wedge \ldots \wedge k^{=1}; i \to \epsilon, 1 \leq i \leq k \\
R_{12} \to 1^{=1} \wedge 2^{=1} \wedge a_1^{=1} \wedge a_2^{=1} \\
R_{23} \to 2^{=1} \wedge 3^{=1} \wedge a_1^{=1} \wedge a_2^{=1} \\
R_{13} \to 1^{=1} \wedge 3^{=1}; R_{123} \to 1^{=1} \wedge 2^{=1} \wedge 3^{=1}.
\end{array}
$$

The a_1, a_2 children are needed under R_{12} and R_{23} for technical reasons related to checking inclusion dependencies; they are not needed for projections used to check fds alone. The query q consists of several parts which, in conjunction with the output DTD, test that either one of the following fails:

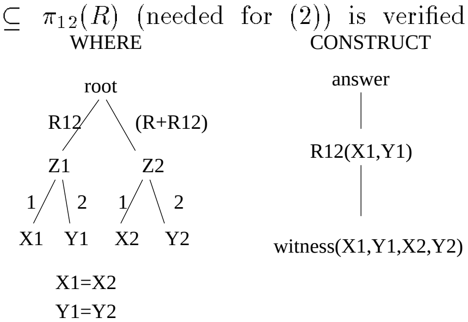
1. the pairs of data values of the children of R_{12} labeled $1, 2$ are distinct for different R_{12} nodes;
2. the set of pairs of data values of the children of R_{12} labeled $1, 2$ equals $\pi_{12}(R)$;
3. for each node labeled R_{12}, the pair of data values of the children of R_{12} labeled a_1 and a_2 is the same as that for the children labeled 1 and 2;
4. the analogs of (1)-(3) for R_{23};
5. the analogs of (1)-(2) for R_{13} and R_{123};
6. the inclusion $\pi_{12}(R) \subseteq \pi_{23}(R)$;
7. the fd $1 \rightarrow 3$;

or the fd $3 \rightarrow 1$ holds.

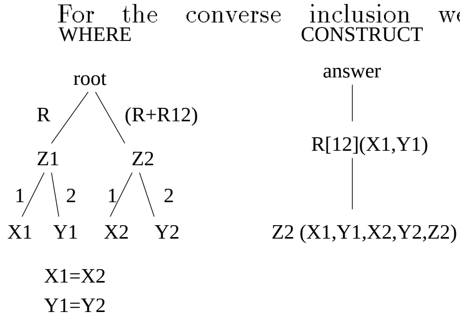
The portion of q verifying that (1) fails is:



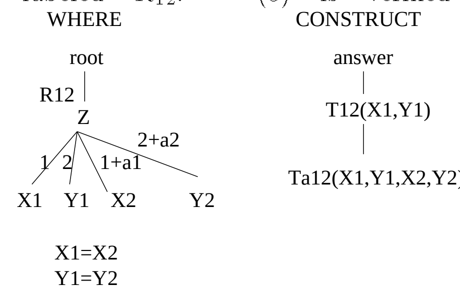
If some r_{12} node has more than one R_{12} node, then (1) fails. The inclusion $R_{12} \subseteq \pi_{12}(R)$ (needed for (2)) is verified using:



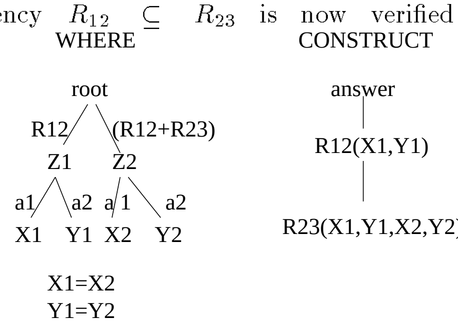
Assuming that (1) holds, $R_{12} \subseteq \pi_{12}(R)$ fails iff some R_{12} node has exactly one *witness* node. For the converse inclusion we use:



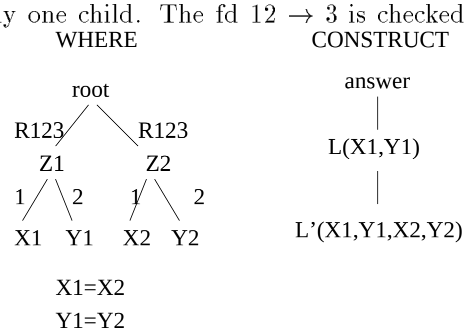
Note that here we use the tag variable $Z2$; the inclusion $\pi_{12}(R) \subseteq R_{12}$ is violated iff there is some node labeled $R[12]$ with no child labeled R_{12}. (3) is verified using:



A violation occurs iff some T_{12} node does not have exactly four children. The inclusion dependency $R_{12} \subseteq R_{23}$ is now verified using:



The inclusion is violated iff some R_{12} node has exactly one child. The fd $12 \rightarrow 3$ is checked using:



The fd is violated iff some L-node has more than one L'-child. Satisfaction of the fd $3 \rightarrow 1$ is checked similarly. The output specialized DTD states, as described above, that at least one of (1)-(7) fails, or the fd $3 \rightarrow 1$ holds. Thus, q typechecks iff $D \models f$.

Proof of Theorem 4.3 We reduce the Post Correspondence Problem (PCP) to typechecking a QL query with respect to an output DTD requiring a nonempty sequence of children under the root. Let $u_1, \ldots, u_k$ and $v_1, \ldots, v_k$ be an instance of the PCP, where $u_i, v_i \in \{a, b\}^+$. We encode a solution to the PCP as a linear data tree (a single path). For simplicity, we represent the path as a string where to each position is associated a symbol (the label of the node) and a data value (the data value of the node). We write such a string as $b_1(v_1) \ldots b_t(v_t)$ where the b_i are symbols and the v_i data values (which may be omitted). Suppose $i_1 \ldots i_m$ is a solution to the PCP, and $u_{i_1} \ldots u_{i_m} = v_{i_1} \ldots v_{i_m} = a_1 \ldots a_n$. The encoding of the solution is a string $x\$y\#$ where x and y specify how $a_1 \dots a_n$ is parsed as $u_{i_1} \dots u_{i_m}$, and $v_{i_1} \dots v_{i_m}$, respectively. For each $i, 1 \le i \le n$, the string x contains four consecutive positions $w(i)s(j)i_j a_i$, a_i occurs within u_{i_j}. The string y is the analogous for $v_{i_1} \dots v_{i_m}$. As an example, consider the instance of the PCP:

u_1	u_2	u_3	v_1	v_2	v_3
aba	aab	bb	a	$abab$	$babba$

and its solution $1, 3, 2, 1$. Note that $u_1 u_3 u_2 u_1 = v_1 v_3 v_2 v_1 = ababbaababa$. The corresponding encoding is the string (read top-down and left-to-right):

$w(1)s(1)1a$	$w(1)s(1)1a$
$w(2)s(1)1b$	$w(2)s(2)3b$
$w(3)s(1)1a$	$w(3)s(2)3a$
$w(4)s(2)3b$	$w(4)s(2)3b$
$w(5)s(2)3b$	$w(5)s(2)3b$
$w(6)s(3)2a$	$w(6)s(2)3a$
$w(7)s(3)2a$	$w(7)s(3)2a$
$w(8)s(3)2b$	$w(8)s(3)2b$
$w(9)s(4)1a$	$w(9)s(3)2a$
$w(10)s(4)1b$	$w(10)s(3)2b$
$w(11)s(4)1a$	$w(11)s(4)1a$
$\$$	$\#$

The input DTD we use is the following:

$root \to w$	$w \to s$	$s \to 1 + \dots + k$
$i \to a + b$	$(1 \le i \le k)$	$a \to w + \$ + \#$
$b \to w + \$ + \#$	$\$ \to w$	$\# \to \epsilon$

The query q is the concatenation of several queries, each of which checks for a *violation* of the correct form for the encoding of a solution. Then q typechecks iff every input yields a violation, so the PCP instance has no solution.

It is easily seen that a tree satisfying τ_1 whose corresponding string is of the form

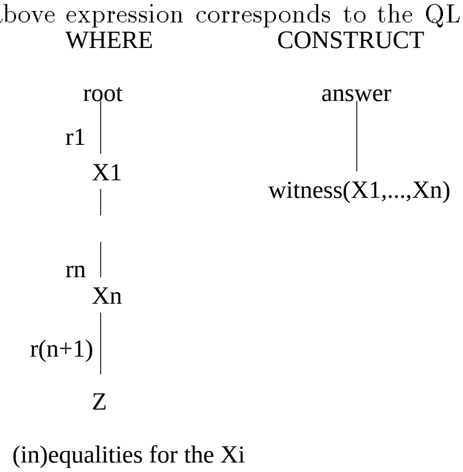
$$w(\alpha_1)s(\beta_1)i_{\beta_1}a_1 \dots w(\alpha_n)s(\beta_n)i_{\beta_n}a_n\$$$
$$w(\alpha'_1)s(\beta'_1)i_{\beta'_1}a_1 \dots w(\alpha'_m)s(\beta'_m)i_{\beta'_m}a'_m\#$$

is an encoding of a solution to the PCP iff it satisfies the following properties:

1. $\alpha_i \ne \alpha_j$ and $\alpha'_i \ne \alpha'_j$ for all $i \ne j$.
2. $\alpha_1 = \alpha'_1$.
3. for each $i < n$, if $\alpha_i = \alpha'_i$, then $i < m$ and $\alpha_{i+1} = \alpha'_{i+1}$.
4. symmetric to (3) for m and n.
5. for each $i (1 \le i \le n)$, $a_i = a'_i$.

   Note that (1)-(4) imply that $n = m$, $\alpha_i = \alpha'_i$, $1 \le i \le n$, *and* $a_1 \dots a_n = a'_1 \dots a'_n$
6. for each fixed β_j, the set
   $$\{i \mid w(i)s(\beta_j)i_{\beta_j} \text{is a substring}\}$$
   is an interval $[\alpha_{j_1}, \alpha_{j_2}]$ of the totally ordered set $\{\alpha_1 \dots \alpha_n\}$, and $a_{j_1} \dots a_{j_2} = u_{i_{\beta_j}}$.
7. analogous property for β'_j and $v_{i_{\beta'_j}}$.
8. $\beta_1 = \beta'_1$;
9. for each i_1, j_1, if $\beta_{i_1} = \beta'_{j_1}$ and there exists some $i_2 > i_1$, such that $\beta_{i_1} \ne \beta_{i_2}$ then there exists some $j_2 > j_1$ such that $\beta'_{j_1} \ne \beta'_{j_2}$.
10. symmetric to (9) for β'_{j_1} and β_{i_1};
11. for each i_1, j_1, if $\beta_{i_1} = \beta'_{j_1}$ then $\beta_{i_2} = \beta'_{j_2}$ where $i_2 = min\{i \mid i > i_1, \beta_i \ne \beta_{i_1}\}$ and $j_2 = min\{j \mid j > j_1, \beta'_j \ne \beta'_{j_1}\}$.
12. for each i, j, if $\beta_i = \beta'_j$ then $i_{\beta_i} = i_{\beta'_j}$.

The queries making up q check that at least one of (1)-(12) fails. We denote queries by expressions of the form $q = r_1.X_1 \dots r_n.X_n.r_{n+1}$ and a conjunction of (in)-equalities among the X_i that apply to the data values. Here the r_i are regular expressions over the alphabet $\Sigma = \{a, b, \$, \#, w, s, 1, \dots, k\}$. Given a string $a_1(v_1) \dots a_m(v_m)$, a binding of q is a mapping ν from $\{X_1, \dots, X_n\}$ to $\{1, \dots, m\}$ such that $a_1 \dots a_{\nu(X_1)} \in r_1$, for every $i, 1 \le i < n$, $a_{\nu(X_i)+1} \dots a_{\nu(X_{i+1})} \in r_n$, and for *some* $v \le m$, $a_{\nu(X_n)+1} \dots a_v \in r_{n+1}$. Additionally, the specified (in)equalities among the data values must hold. The above expression corresponds to the QL query:



To illustrate, consider properties (1)-(4). The queries detecting a violation of (1) are $*w.X.*w.Y.*.\$, X = Y$ (recall that (in)equalities apply to the data values) and $*.\$.*w.X.*w.Y$, $X = Y$. The query detecting a violation of (2) is $w.X.*.\$.w.Y$, $X \ne Y$. Violations of (3) and (4) are detected by the following queries:

- $*.w.X_1.\Sigma^3.w.Y_1.*\$.*.w.X_2.\Sigma^3.w.Y_2$, $X_1 = X_2, Y_1 \neq Y_2$ (for two indexes of equal value occurring before and after \$, their successors are not equal);

- $*.w.X_1.\Sigma^3.w.Y_1.*\$.*.w.X_2.\Sigma^3\#$, $X_1 = X_2$ (for two indexes of equal value occurring before and after \$, the first has a successor and the second does not);

- $*.w.X_1.\Sigma^3\$*.w.X_2.\Sigma^3.w.Y_2$, $X_1 = X_2$ (for two indexes of equal value occurring before and after \$, the second has a successor and the first does not).

Clearly, (3) and (4) are satisfied iff all queries return the empty answer. The queries used for the remaining properties are similar.