XML with Data Values: Typechecking Revisited*

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

Tova Milo
Tel Aviv University
milo@post.tau.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 Document Type Definitions (DTDs), or XML Schemas, which constrain valid XML documents. The benefits of schemas are numerous. Some are analogous to those derived from schema information in relational query processing. Perhaps most importantly to the context of the Web, schemas can be used to validate data exchange. In a typical scenario, a user community would agree on a common schema and on producing only XML documents which are valid with respect to the specified schema. 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 schema.

The typechecking problem takes as input a query and two schemes (or types), one for the input XML documents and one for the output XML documents generated by the query. The goal is to verify whether all the XML documents generated by the query, when applied to documents that conforms to the input type, conform to the output type. In practice, the typechecker is a program

*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.

module that analyses the query and either accepts or rejects it. One approach to typechecking is *type inference*, a technique derived from functional programming languages and first adapted to XML by XDuce [19, 20]. Murata also addresses the type inference problem for transformations expressed with certain tree automata [27]. Given program, (and possibly an input type), the type inference system constructs a most general output type for that program in a bottom up fashion. Typechecking can then be performed by checking that the inferred type is a subset of the given output type. This approach is quite appealing in practice because typechecking is easy to implement and is extendible to a large class of query languages; for example XQuery uses this approach [15].

However, we showed in a previous paper [23] that any type inference system is incomplete, i.e. it cannot compute the most general output type and, as a consequence, the resulting typechecking algorithm may reject some queries that are correct. In practice this is a serious limitation for XML typecheckers, forcing users to turn the typechecker off (when this is an option), or to rewrite the query in non-obvious ways, in an attempt to overcome the typecheckers limitations. The second approach to typechecking, which we advocated in [23], is to design specific techniques that are complete for a given query language. We considered a particular class of tree transformations that can be expressed by so called *k-pebble transducer*, which we showed to be powerful enough to subsume the tree manipulation core of practical XML query languages, including recursive traversals like in XSLT [11], and described a method for typechecking all transformations in this class. The technique, however, is specific only to the particular language considered, i.e. the class of transformations expressed by k-pebble transducers, and does not extend in obvious ways to other languages. The main limitation of k-pebble transducers is that they do not allow joins between data values, which is a feature found in most query languages. We showed in [23] that type checking becomes undecidable if k-pebble transducers are extended with joins between data values. However, this negative result is not worrisome in itself, because the the class of transformations defined by k-pebble transducers with joins is more powerful than what is needed in practice. Thus, the results in [23] leave unexplored a large class of queries of significant practical interest: queries that can express joins by comparing data values, but do less powerful tree restructurings than k-pebble transducers. This class is precisely where practical declarative query languages lie (XML-QL [13], XQuery [8]) and deserves a thorough investigation.

The present paper investigates typechecking of queries with comparisons of data values. We focus on declarative query languages in the style of XML-QL and XQuery and various fragments thereof, with path expressions containing regular expressions, but without recursive functions, and on types consisting of various extensions and restrictions of DTDs. Our findings are mostly negative: typechecking is undecidable for most practical purposes, but we also find a number of cases where typechecking is decidable. The main contribution is to trace the boundary of decidability of the typechecking problem in the presence of data values, for various combinations of query languages and types. 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 type specialization in DTDs (also known as decoupled tags). 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 [23] 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 [31]. 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.

Typechecking XML views of relational databases is considered in [1]. Although the basic framework is quite different from the present paper, this problem is related to the one investigated here. Indeed, some upper bound results from the present papers transfer to [1] and some lower bound results from [1] transfer to the present paper. The relationship between the two papers is discussed in more detail in Section 6.

The type inference approach to typechecking XML transformations has first been introduced by XDuce [19, 20]. XDuce is a general-purpose functional language in the style of ML [21], whose types are essentially DTDs with specialization. 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, as we showed in [23], 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, in the case of XML the typechecker fails even on simple functions that are expressible in, say, join-free XML-QL, and for which we know already that typechecking is decidable [23]. 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 [12, 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 [22]. We are given a declarative query and are asked to find all possible types of the variable assignments: the query's output type is not considered in this problem. This is a different and more limited problem than XDuce's type inference. The analysis in [22] shows that the complexity of this problem ranges from PTIME to NP-complete for various combinations of query languages and output types.

XML types are usually abstracted as regular tree languages: in both XML type formalisms used in practice, DTDs and XML-Schema a type is a regular tree language, but none of these formalisms can express all regular tree languages. Regular tree languages have been traditionally considered for ranked trees [35]. Several techniques for extending them to unranked trees have been proposed in the literature: by using unranked automata [4] as specialized DTD's [31] by encoding unranked trees as binary trees [23] and with hedge automata [28]. These formalisms turn out to be equivalent, i.e. they all define the same class of unranked regular tree languages. In this paper we use specialized DTDs [31].

Organization. The paper is organized as follows. The first section develops the basic framework, including our abstractions of XML documents and DTDs, and our query language formalism. Sections 3 and 4 present the decidability and complexity results, respectively, and Section 5 the undecidability results. Section 6 discusses technical connections between [1] and the present paper. Finally, the paper ends with brief conclusions.

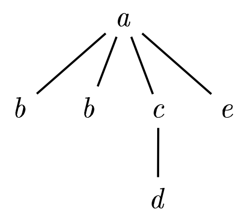
2 Basic Framework

We introduce here the basic formalism used throughout the paper, including our abstractions of XML documents, DTDs, and queries.

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 $T = \langle t, label, val \rangle$ 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 $root(T)$. Since the tree is ordered, we can define a unique depth-first traversal on its nodes, and denote with $<$ the resulting total order on $nodes(T)$; in particular, if x is an ancestor of y then $x < y$. We refer to elements in Σ assigned by *label* 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. When modeling XML trees only leaf nodes make sense to carry data values, and we will simply ignore all the other values. 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 $\mathcal{T}_\Sigma$. For each $T \in \mathcal{T}_{\Sigma,\mathcal{D}}$, we denote with $s(T) \in \mathcal{T}_\Sigma$ its *structure*, i.e. the tree obtained by dropping all data value labels. We will often write T instead of $s(T)$ when no confusion arises. We sometimes denote a tree with root r and sequence of subtrees $T_1, \ldots, T_n$ by $r(T_1, \ldots, T_n)$.

Types and DTDs. As usual, we define XML types in terms of the tree structure alone, and ignore data values: an XML type, $\mathcal{T}$, is a set of trees $\mathcal{T} \subseteq \mathcal{T}_\Sigma$. Each type $\mathcal{T}$ also defines a set of trees with data values, namely $s^{-1}(\mathcal{T}) = \{T \mid s(T) \in \mathcal{T}\}$, and we will often write $\mathcal{T}$ instead of $s^{-1}(\mathcal{T})$ when no confusion arises. We discuss in the sequel various specification methods for XML types that we consider in this paper.

The basic specification method is (an abstraction of) DTDs. A DTD consists of an extended context-free grammar over alphabet Σ (we make no distinction between terminal and non-terminal symbols). In an extended CFG, the right-hand sides of productions are regular expressions over the terminals and non-terminals. 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 with respect to the DTD: $a \to b^*.c.e;\quad b \to \varepsilon;\quad c \to d^*;\quad d \to \varepsilon;\quad e \to \varepsilon.$

The set of data trees satisfying a DTD τ is denoted by $inst(\tau)$. Strictly speaking, this is a set of trees without data values (i.e. a subset of $\mathcal{T}_\Sigma$) but, as we discussed, we can view it as a set of trees with data values (i.e. a subset of $\mathcal{T}_{\Sigma,\mathcal{D}}$).

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 $w = a_1 \ldots a_n$

is represented by the logical structure $(\{1,\ldots,n\};<,(O_\sigma)_{\sigma\in\Sigma})$ where $<$ is the natural order on $\{1,\ldots,n\}$, 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 on the logical vocabulary of strings [5, 14]. MSO is first-order logic augmented with quantification over sets. 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* [35]. There is a language-theoretic characterization of star-free languages: they are precisely described by the *star-free regular expressions*, which are built from single symbols, $\emptyset$, 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 [29]. 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. For conciseness, we denote by ϵ the formula $\bigwedge_{a\in\Sigma} a^{=0}$. 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

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

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-Schema. This is motivated by a 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 $a(b(c),b(d))$ cannot be described by a DTD, because the "type" of the first b differs from that of the second b. Of course, T has a DTD (in fact, several), but none defines precisely the set $\{T\}$. 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;*
- 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. It has been shown [31, 4] that specialized DTDs are precisely equivalent to regular tree automata over unranked trees defined in [4], which in turn are equivalent to hedge automata [28]. This is more evidence that specialized DTDs (and the other formalisms) 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. A query in the language QL defines a function $\mathcal{T}_{\Sigma,\mathcal{D}} \rightarrow \mathcal{T}_{\Sigma}$. That is, the function takes as input a tree *with* data values, since we want to allow this function to compare different data values for equality, and returns a tree *without* data values, since our discussion is restricted to studying the type of the output tree. To make such a formalism practical, one needs to extend it to include data values in the output tree: this is orthogonal to our discussion on typechecking and we do not consider such an extension here.

A query Q in QL consists of a tree template in which each node is labeled with a tag and a formula: $Q \in \mathcal{T}_{Tag \times \mathcal{L}}$. A *tag* is either a symbol in Σ or a variable from a fixed a set of variables, *Var*: that is, $Tag = \Sigma \cup \mathit{Var}$. First, we define a set of formulas, $\mathcal{L}$. Then we define QL formally.

Formulas in $\mathcal{L}$ will be interpreted over the input tree to the query, $T \in \mathcal{T}_{\Sigma,\mathcal{D}}$. There are two kinds of atomic formulas: path expressions and comparisons. A path expression is of the form $\psi = X \ R \ Y$, where $X, Y \in \mathit{Var}$ and R is a regular expression over Σ. Such expressions are interpreted as follows: ψ is true in T if there exists a path from node X to node Y whose sequence of labels (excluding the first node and including the last node) on that path belongs to R. A comparison formula is of the form $\psi = X \ \mathit{op} \ V$ where $X \in \mathit{Var}$, $\mathit{op} \in \{=, \neq\}$, and $V \in \mathit{Var} \cup \mathcal{D}$. Comparison formulas are interpreted as follows: ψ is true in T if X's data value is equal to V's data value. Further, formulas in $\mathcal{L}$ are conjunctive formulas over atomic formulas. More precisely, $\varphi \in \mathcal{L}$ is an expression of the form:

$$\varphi \quad = \quad \exists Y_1 \ldots \exists Y_k (\psi_1 \wedge \ldots \wedge \psi_m) \tag{1}$$

The variables $Y_1, \ldots, Y_k$ are called existential, or bound variables. All other variables are called free variables. We denote the set of free variables with $\mathit{free}(\varphi)$. As usual, existential variables correspond to projections, and we say that a formula φ is projection-free when it has no existential variables. When we consider several formulas $\varphi_1, \varphi_2, \ldots$, we always assume that they have disjoint sets of existential variables (otherwise we rename these variables).

We often write a formula as a datalog rule, by dropping the existential quantifiers and instead listing the free variables in the rule's head. That is, φ in (1) can be written as:

$$P(X_1, \ldots, X_n) \quad :- \quad \psi_1, \ldots, \psi_m \tag{2}$$

where P is some identifier, and $\mathit{free}(\varphi) = \{X_1, \ldots, X_n\}$.

We can now formally define the language QL:

Definition 2.2 *A query Q in QL is a tree $Q \in \mathcal{T}_{Tag \times \mathcal{L}}$ such that the following conditions hold. We denote with $tag(v)$ and ϕ_v the tag and the formula associated to a node v, respectively.*

- *For any two nodes v, v' if v is an ancestor of v' then $\mathit{free}(\phi_v) \subseteq \mathit{free}(\phi_{v'})$.*
- *If r is the root node, then $\phi_r \equiv \mathit{true}$ and $\mathit{free}(\phi_r) = \emptyset$.*
- *If $tag(v) \in \mathit{Var}$ then $tag(v) \in \mathit{free}(\phi_v)$.*

A query is called *projection-free* if all its formulas are projection-free.

Example 2.3 Figure 1 illustrates a query in QL. Here Q is a tree where each node is labeled with a rule $label(\bar{X}) \leftarrow \psi_1, \ldots, \psi_n$. Here, $label \in Tag$ and $\psi_1, \ldots, \psi_n$ is datalog rule with free variables $\bar{X}$. We refer to the nodes with labels root, title; actor, V, AllensReviews, and reviews as $v_0, v_1, \ldots, v_5$, respectively. Where the datalog rule is missing it is assumed to be *true*. All path expressions occurring in datalog rules are very simple and consist of a single tag (e.g. *movie*, *director*, etc), with the exception of _ which is the wildcard (a convenient abbreviation meaning any tag). This query is not projection-free, since several formulas have existential variables.

Next, we define next the query's semantics, for which we first need some notation. Let T be some input tree in $\mathcal{T}_{\Sigma,\mathcal{D}}$. If $V \subseteq Var$ is a set of variables, then a V-substitution (or V-binding) is a mapping $\theta : V \to nodes(T)$. We fix a distinguished variable $X_0 \in Var$ which we always interpret as the root, $\theta(X_0) = root(T)$. Recall that $nodes(T)$ is totally ordered by a depth-first traversal. We extend this order to a lexicographic order on V-substitutions (for this we fix a total order on Var). If $V' \supseteq V$ and θ' is a V'-substitution then we say that θ' extends θ if $\theta(X) = \theta'(X)$ for all $X \in V$. Finally, if v is a node in some query $Q \in QL$, and $root(Q) = v_0, v_1, \ldots, v_{n-1}, v_n = v$ is the path from the root to v, then we denote by $\Phi_v = \phi_{v_0} \wedge \phi_{v_1} \wedge \ldots \wedge \phi_{v_n}$, the conjunction of all formulas labeling v's ancestors, including v. Clearly, $free(\phi_v) = free(\Phi_v)$. Now we can define Q's semantics:

Definition 2.4 *Given a query $Q \in QL$ and a tree $T \in \mathcal{T}_{\Sigma,\mathcal{D}}$, the output $Q(T)$ is a tree $T' \in \mathcal{T}_{\Sigma}$ defined as follows:*

- *$nodes(T')$ consists of all pairs (v, θ) such that $v \in nodes(Q)$, $\theta : free(\phi_v) \to nodes(T)$, and $T \models \Phi_v[\theta]$;*
- *$nodes(T')$ are ordered by the relation $(v, \theta) < (v', \theta')$ if $v < v'$ or $v = v'$ and $\theta < \theta'$. Here $v < v'$ denotes the node ordering in Q, while $\theta < \theta'$ is the lexicographic order discussed above;*
- *$edges(T') = \{((v, \theta), (v', \theta')) \mid (v, v') \in edges(Q), \theta' \text{ extends } \theta\}$; and,*
- *if $tag(v) \in \Sigma$, then $label(v, \theta) = tag(v)$; if $tag(v) \in Var$, then $label(v, \theta) = label(\theta(v))$.*

Example 2.5 Continuing Example 2.3, assume that the input tree, T, is an XML document holding information about movies (titles, directors, actors, and possibly reviews), described by the (partial) DTD:

$$root \to movie^*; \quad movie \to title.director.review^*;$$

$$title \to actor^*; \quad actor \to name.\Sigma^*;$$

$$director \to \epsilon; \quad review \to \epsilon.$$

The query in Figure 1 collects W. Allen's movie titles, their actors (grouped under title), all available information about each actor (grouped under the actor with the same tags as in the input), and their reviews, if any. The output tree has a single root node, labeled *root*, with several children labeled *title*: there will be one such child for every value X_2 satisfying the formula:

$$\Phi_{v_1}(X_2) \quad :- \quad X_0 \ movie \ X_1, X_1 \ title \ X_2, X_1 \ director \ X_3, X_3 = \text{“W.Allen”}$$

Intuitively there is one *title* in the output for each movie directed by *“W.Allen”*. Each corresponds to a unique value of X_2 and has several *actor* children, one for each binding of U satisfying the formula:

$$\Phi_{v_2}(X_2, U) \quad :- \quad \Phi_{v_1}(X_2), X_2 \ actor \ U$$

$$\text{root}()$$

$$\text{title}(X_2) \leftarrow X_0 \text{ movie } X_1,\ X_1 \text{ title } X_2,\ X_1 \text{ director } X_3,\ X_3 = \text{“ W. Allen”}$$

$$\text{actor}(X_2, U) \leftarrow X_2 \text{ actor } U \qquad \text{AllensReviews}(X_2)$$

$$V(X_2, U, V) \leftarrow U \_\, V \qquad \text{reviews}(X_2, Y_2) \leftarrow Y_1 \text{ title } X_2,\ Y_1 \text{ review } Y_2$$

Figure 1: The Woody Allen query

In addition each *title* node has exactly one child labeled *AllensReviews*, because $\Phi_{v_4} = \Phi_{v_1}$.

For each *actor* node there will be a number of children, one for each binding of V satisfying the formula:

$$\Phi_{v_3}(X_2, U, V) \quad :- \quad \Phi_{v_2}(X_2),\ U \_\, V$$

The tag of that node will be the same as V's tag. Recall that _ denotes here the regular expression which matches any tag, i.e. V is simply bound to all U's children, in effect copying all actors' children from the input tree to the output tree.

Finally, nodes of type *AllensReviews* will have several children labeled *reviews*, one for each binding of Y_2 satisfying the formula:

$$\Phi_{v_5}(X_2, Y_2) \quad :- \quad \Phi_{v_4}(X_2),\ Y_1 \ \textit{title}\ X_2,\ Y_1 \ \textit{review}\ Y_2$$

The intention here is for Y_2 to collect all reviews of X_2's parent. We already bound X_1 to X_2's parent in the formula φ_{v_1}, but the variable X_1 is not free making it inaccessible for ϕ_{v_5}. Hence we use a new variable Y_1 to be bound to X_2's parent. Since we interpret such queries only over trees, Y_1 will be bound to the same node as X_1.

QL can express a large subset of the queries expressible in XML-QL [13] and XQuery [8], although the latter have block structure while QL does not. For example the query in Example 2.5 can be expressed equivalently in XQuery [8] as:

```
<root>
  { for $X2 in /movie/title[director="W.Allen"]
    return <title>
              { for $U in $X2/actor
                return <actor>
                          { for $V in $U/*
                            return { element {$V} { } }
                          }
                       </actor>
              }
              <AllensReviews>
                 { for $Y2 in $X2/../review
                   return <reviews/>
                 }
              </AllensReviews>
```

```
            </title>
    }
</root>
```

Notice how the block structure in XQuery can be expressed in QL by a judicious use of free variables. QL can express that way a rather large fragment of XQuery.

Remark. As already mentioned, our definition does not address how data values are to be included in the output tree. For instance, in Example 2.5 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 we only study the type of the output tree and 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 5 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 where path expressions define *finite* languages), and unordered output DTDs;
2. non-recursive QL without tag variables and star-free output DTDs;
3. non-recursive, "projection-less" QL without tag variables, and regular output DTDs. We will formally define projection-less queries later on. For now, recall that a projection-free query is one with no existential variables. A "projection-less" query is one that is equivalent, under a given input DTD, to its projection-free variant obtained by dropping all the existential quantifiers.

The above results highlight an interesting trade-off between the query language and the DTDs. The undecidability results of Section 5 show that our decidability results are quite tight.

All three results are proven using the following idea. If there exists some input tree T such that $Q(T)$ does not conform to the required output type, then there exists a subtree $T_0 \subseteq T$ whose size is bounded by a function that depends only on Q and the input and output types, such that $Q(T_0)$ also fails to conform to the output type. Hence, in order to typecheck Q it suffices to compute $Q(T_0)$ for all trees T_0 up to a given size, and check that each result conforms to the output type. Before giving the formal proofs, we illustrate the basic intuition on the following simple example.

Example 3.1 Consider the query Q in Figure 2. The nodes with labels a, b, c, d, and e are referred to as $v_0, \ldots, v_4$, respectively. Assume that we need to typecheck Q against the following output DTD:

$$\begin{aligned} a &::= b* \\ b &::= c* \\ c &::= (d, d, d, d*, (e, e?)?) \end{aligned}$$

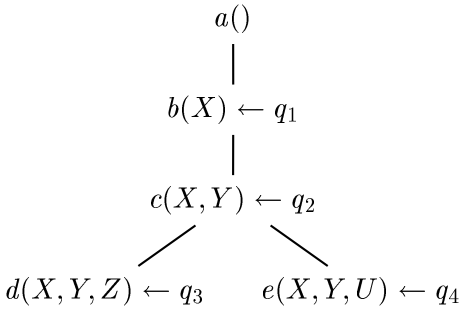


Figure 2: A query for Example 3.1.

The DTD requires a c element to have at least three d children and at most two e children. Suppose $Q(T)$ fails to conform to this DTD. Assume there is a node u labeled c in $Q(T)$ that violates the output DTD. There are two cases: either u has two or fewer d children, or u has three or more e children. In both cases we start by finding a subtree $T_0 \subseteq T$ such that u is also in $Q(T_0)$. Since u is witnessed by a substitution $\theta : \{X, Y\} \rightarrow \mathit{nodes}(T)$ we only need two nodes in T to make this substitution possible; we also need some extra nodes in T to serve as witnesses for the existential variables in $\Phi_{v_2} = \phi_{v_2} \wedge \phi_{v_1}$ (namely one node for each existential variable in $\mathit{bound}(\Phi_{v_2})$); and we need all ancestors of these nodes to ensure that T_0 is a tree. The number of nodes in T_0 is at most $(\mid q_1 \mid + \mid q_2 \mid) \times \mathit{height}(T)$. In the first case, when u has two or fewer d children in $Q(T)$, then we argue that u has also two or fewer d-children in $Q(T_0)$ (it cannot have more than in $Q(T)$), hence $Q(T_0)$ violates the output DTD. In the second case, when u has three or more e children, then we choose arbitrarily three such children, u_1, u_2, u_3. They are witnessed by three different extensions of θ to the variable U, hence we only need $3 \mid q_4 \mid$ nodes in T to allow these three nodes to be constructed: by adding these and all their ancestors to T_0, we ensure that u still has the three children u_1, u_2, u_3 in $Q(T_0)$, i.e. $Q(T_0)$ also violates the output DTD, and its size is $\leq (\mid q_1 \mid + \mid q_2 \mid + 3 \mid q_4 \mid)\mathit{height}(T)$. Except for the factor $\mathit{height}(T)$ everything else depends only on the query and the output DTD (the number 3 was derived from the output DTD). Finally we explain how to reduce the $\mathit{height}(T)$ factor. Consider the input DTD. If it is not recursive, then $\mathit{height}(T)$ is also bound. Otherwise we use a form of a pumping lemma applied to the input DTD and all paths expressions in q_1, q_2, q_3, q_4 to argue that we can bound the height of T_0. In all cases some additional elements may need to be added to T_0 to make it conform to the input DTD. At the end we obtain a tree T_0 with a bounded size, such that $Q(T_0)$ does not conform to the output DTD.

The example only illustrates the basic intuition. Rigorous proofs differ for each special case we consider, and are provided in the remainder of the section.

Our first result concerns non-recursive QL and unordered output DTDs. The size of a query q, denoted by $|q|$, and of a DTD τ, denoted by $|\tau|$, is simply the number of characters in their definition.

Theorem 3.2 *The typechecking problem for non-recursive QL queries, regular input DTDs, and unordered output DTDs is decidable in* CO-NEXPTIME.

Proof: Let q be a non-recursive QL query, τ_1 a regular input DTD, and τ_2 an unordered output DTD. Suppose $T \in \mathit{inst}(\tau_1)$ and $q(T) \notin \mathit{inst}(\tau_2)$. We show that there exists $T_0 \in \mathit{inst}(\tau_1)$ with at most exponentially many nodes (with respect to $|q|, |\tau_1|, |\tau_2|$), and $q(T_0) \notin \mathit{inst}(\tau_2)$. We construct T_0 from T as follows. Since $q(T)$ violates τ_2, there exists a path $n_1 \dots n_k$ from the root of $q(T)$ to a node n_k with tag $a \in \Sigma$ such that the sequence w of the 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 (not necessarily continuous) subsequence of w that satisfies $C_l^{=} = a_1^{=i_1} \wedge \ldots \wedge a_h^{=i_h}$. Let $var(q)$ denote the set of all variables in all formulas in q, both bound and free variables. Each node on the path $n_1 \ldots n_k$ from root to n_k, and among the nodes $v_1 \ldots v_n$ mentioned above, was constructed due to 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 $|B| \leq |q|^2(|q| + |\tau_2||\Sigma|)$. This is because $k \leq |q|$ and $n \leq |\tau_2||\Sigma|$, 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 $|q|$. The inequality $n \leq |\tau_2||\Sigma|$ assumes the integers in τ_2 are written in unary.

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 the bindings in T that contributed to the creation of n_k in $q(T)$ still exist 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). Thus, the number of children of any node in T_0 is bounded by $[|B| + (|B|+1)|\tau_1|]$. 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 $[|B| + (|B|+1)|\tau_1|]^{|q|}$. 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 $[\ |B| + (|B|+1)|\tau_1|]^{|q|} \times (1 + |\tau_1|^{|\Sigma|})$ for the size of T_0.

Hence, our procedure testing that q does *not* typecheck with respect to input DTD τ_1 and output DTD τ_2 simply consists of guessing a T_0 of exponential size and then verifying whether (1) T_0 satisfies τ_1, and (2) $q(T_0)$ violates τ_2. Both (1) and (2) can be tested in exponential time. Indeed, let N be the size of the input. Then (1) can be checked in time linear in T_0 and N, which is exponential in N: just check for every node that its sequence of children matches the specified regular expression. For testing (2), we first construct the output tree. As there are at most $|T_0|^N$ possible bindings, this takes time at most exponential in N. Next, we transform each SL formula in DNF, which can be done in time exponential in N and which can result in formulas that are at most of exponential size. So in the worst case, we have to check for an exponential number of nodes, an exponential number of disjuncts. $\square$

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

Theorem 3.3 *The typechecking problem for non-recursive QL queries without tag variables, with regular input DTDs, and star-free output DTDs, is decidable in* CO-NEXPTIME.

Proof: The proof is by reduction to the case when the output DTD is unordered, for which we can use Theorem 3.2. 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 φ_r using integers whose value is bounded by $|r|$ and computable in EXPTIME from r and $a_1, \ldots, a_k$, such that

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

The proof of (†), is by induction on the structure of r. We provide it for completeness at the end of the current proof.

Now consider a non-recursive QL query q without tag variables and a star-free output DTD τ_2. Suppose first that

(∗) all siblings nodes in 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 CO-NEXPTIME upper bound then follow from Theorem 3.2. 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, and the size of the counterexample in the proof of Theorem 3.2 is exponential only in $|q|$, $|\tau_1|$, and the integers occurring in the SL formulas in τ_2.

Suppose now that q violates (∗). We construct from q a query $\bar{q}$ by replacing in q 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 φ_r, using integers whose value is bounded by $|r|$ and computable in EXPTIME from r, $a_1, \ldots, a_k$, and $b_1, \ldots, b_k$, such that

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

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 CO-NEXPTIME upper bound follow as above.

To conclude, we provide the proof of (‡). The proof of (†) immediately follows, taking h to be the identity mapping.

More specifically, we define inductively φ_r as a disjunction of statements of the form $b_1^{*_1 i_1} \wedge \cdots \wedge b_n^{*_n i_n}$ (abbreviated for convenience $b_1^{*_1 i_1} \cdots b_n^{*_n i_n}$) where each $*i_j \in \{=, \geq\}$ and i_j are natural numbers. Moreover, $i_1, \ldots, i_n \leq |r|$, the size of φ_r is exponential in $|r|$, and φ_r can be computed in time exponential in $|r|$.

The inductive definition proceeds as follows. We distinguish several cases:

1. If $r = \emptyset$ or $r = \varepsilon$ then φ_r is the empty disjunction and consists of the only disjunct $b_1^{=0} \cdots b_n^{=0}$, respectively.

2. If $r = a_i$, then φ_r is a disjunctions of terms of the form $b_1^{=j_1} \cdots b_n^{=j_n}$, having one disjunct per each b_k such that $h(b_k) = a_i$, with $j_k = 1$ and $j_\ell = 0$ for $\ell \neq k$.

3. If $r = r_1 + r_2$ then φ_r consists of the disjuncts in φ_{r_1} and φ_{r_2}.

4. If $r = r_1 \cdot r_2$, then φ_r consists of the disjuncts obtained from φ_{r_1} and φ_{r_2} as follows. Add
$$b_1^{*_1 i_1} \cdots b_n^{*_n i_n}$$
each time there is a disjunct
$$b_1^{*_1^1 i_1^1} \cdots b_n^{*_n^1 i_n^1}$$
in φ_{r_1} and a disjunct
$$b_1^{*_1^2 i_1^2} \cdots b_n^{*_n^2 i_n^2}$$
in φ_{r_2} such that there is a $j \in \{1, \ldots, n\}$ for which the following holds
   - $i_\ell^1 = 0$ for all $\ell > j$;
   - $i_\ell^2 = 0$ for all $\ell < j$;
   - $*_j =$ '$=$' if both $*_j^1$ and $*_j^2$ are '$=$', otherwise $*_j$ is $\geq$;
   - $i_j = i_j^1 + i_j^2$;
   - $*_\ell = *_\ell^1$ and $i_\ell = i_\ell^1$ for $\ell < j$; and
   - $*_\ell = *_\ell^2$ and $i_\ell = i_\ell^2$ for $\ell > j$.

   For instance, the concatenation of $a^{=1}b^{\geq 2}c^{\geq 0}$ and $a^{\geq 0}b^{=1}c^{=3}$ gives rise to $a^{=1}b^{\geq 3}c^{=3}$.

5. If $r = \neg s$, then φ_r is the intersection of the negation of the disjuncts of φ_s. The negation of a single disjunct $b_1^{*_1 i_1} \cdots b_n^{*_n i_n}$ is equivalent to the disjunction obtained by adding for every $j = 1, \ldots, n$, the disjuncts
$$b_1^{*_1 i_1} \cdots b_j^{=0} \cdots b_n^{*_n i_n},\ b_1^{*_1 i_1} \cdots b_j^{=1} \cdots b_n^{*_n i_n},\ \ldots,\ b_1^{*_1 i_1} \cdots b_j^{=i_j - 1} \cdots b_n^{*_n i_n}.$$
If $*_j =$ '$=$', then we also add
$$b_1^{*_1 i_1} \cdots b_j^{\geq i_j + 1} \cdots b_n^{*_n i_n}.$$
We now have an intersection of sets of disjuncts. By De Morgan's laws we transform this to a disjunction of conjunctions. Every conjunction can then be transformed to a single expression as follows. Suppose we have the conjunction $\bigwedge_k b_1^{*_1^k i_1^k} \cdots b_n^{*_n^k i_n^k}$. This conjunction can be omitted, as it is contradictory, when there is a j and $\ell_1 \neq \ell_2$ with
   - $*_j^{\ell_1} = *_j^{\ell_2} =$ '$=$', and $i_j^{\ell_1} \neq i_j^{\ell_2}$; or
   - $*_j^{\ell_1} =$ '$=$', $*_j^{\ell_2} =$ '$\geq$', and $i_j^{\ell_1} < i_j^{\ell_2}$.

   Otherwise the conjunction is equivalent to $b_1^{*_1 i_1} \cdots b_n^{*_n i_n}$, where for each $j = 1, \ldots, n$,
   - $*_j =$ '$=$' and $i_j = i_j^\ell$ for some ℓ for which $*_j^\ell =$ '$=$'; and
   - $*_j =$ '$\geq$' and $i_j = \max\{i_j^\ell \mid \ell\}$ when no $*_j^\ell$ is '$=$'.

From the construction it readily follows that the integers are $\leq |r|$. Further, as there are at most $(2|r|)^n$ possible disjuncts, the size of φ_r is exponential in $|r|$. However, the above algorithm is double exponential in the size of $|r|$. Indeed, the conversion from conjunctive to disjunctive normal form can result in an exponential blow-up of the formula size. Therefore, we next give a brute force algorithm to compute φ_r in exponential time.

$$\text{root}()$$

$$\text{title}(X_0, X_1, X_2, X_3) \leftarrow X_0 \text{ movie } X_1,\ X_1 \text{ title } X_2,\ X_1 \text{ director } X_3,\ X_3 = \text{“ W. Allen”}$$

$$\text{actor}(X_0, X_1, X_2, X_3, U) \leftarrow X_2 \text{ actor } U \qquad \text{AllensReviews}(X_0, X_1, X_2, X_3)$$

$$V(X_0, X_1, X_2, U, V) \leftarrow U * V \qquad \text{reviews}(X_0, X_1, X_2, X_3, Y_1, Y_2) \leftarrow Y_1 \text{ title } X_2,\ Y_1 \text{ review } Y_2$$

Figure 3: A projection-free query.

We have just shown that the integers in the expression are bounded above by $|r|$. Therefore, we can define φ_r as the expression

$$\bigvee\{b_1^{*_1 i_1} \cdots b_n^{*_n i_n} \mid (*_j = \text{‘='} \text{ and } i_j \leq |r|) \text{ or } (*_j = \text{‘} \geq' \text{ and } i_j = |r| + 1), \text{ and } h(b_1^{i_1} \cdots b_n^{i_n}) \subseteq L(r)\}.$$

As there are only exponentially many strings $b_1^{*_1 i_1} \cdots b_n^{*_n i_n}$ and $b_1^{i_1} \cdots b_n^{i_n} \subseteq L(r)$ can be tested in PSPACE, the result follows. To see the latter, we can translate the star-free regular expression into an equivalent FO sentence whose size is linear in the size of r and test whether $b_1^{*_1 i_1} \cdots b_n^{*_n i_n}$ satisfies this formula. This is well-known to be in PSPACE. □

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 3.4 *Let q be a QL query and τ an input DTD. The query q is* projection-less *with respect to τ iff it is equivalent, on all inputs satisfying τ, to the projection-free query q' obtained from q by dropping all the existential quantifiers.*

Obviously, there are many syntactic sufficient conditions on q and τ ensuring that q is projection-free with respect to with respect to τ. We illustrate this by an example.

Example 3.5 Consider the *movie* DTD of Example 2.5 and the query in q in Figure 1. Obviously q not projection-free, but we show that it is projection-less. For that we construct the query q' in Figure 3, by promoting all existential variables to free variables (i.e. including them in the datalog rule's head). Clearly q' is projection-free. We argue now that q and q' are equivalent over input documents of the given DTD. For that it suffices to show for each formula the query q that, if there exists some binding of its existential variables that makes that formula true, then that binding is unique. We only illustrate this on the formula ϕ_{v_1} given by the datalog rule:

$$\textit{title}(X_2) \quad :- \quad X_0 \textit{ movie } X_1, X_1 \textit{ title } X_2, X_1 \textit{ director } X_3, X_3 = \textit{“W.Allen”}$$

where X_0, X_1, X_3 are existential variables. Given a binding of X_2, clearly bindings for X_0, X_1 are unique: X_0 is the root and X_1 is the parent of X_2. A binding for X_3 is also unique, because the DTD requires a *movie* to have a unique *director*. This completes the proof for ϕ_{v_1}, and the other formulas are treated similarly. It follows that q and q' are equivalent and, consequently, that q is projection-less.

Restricting queries to be projection-less is clearly a limitation. However, as this example suggests, the limitation may be less severe in practice when the input DTD has certain uniqueness requirements.

For projection-less queries we prove next the following result, the most technically challenging of this section.

Theorem 3.6 *The typechecking problem for projection-less non-recursive QL queries without tag variables, with regular input DTDs, and regular output DTDs, is decidable.*

It remains open whether Theorem 3.6 holds without the projection-less restriction.

The proof of Theorem 3.6 uses Ramsey's Theorem [18, 32] and requires developing some technical machinery. We dedicate the remainder of the section to this development.

Assume we are given some projection-less non-recursive QL query q'. By the definition of *projection-less*, q' is equivalent to the projection-free query q obtained from q' by dropping all the existential quantifiers. So it suffices to prove the theorem for q.

To simplify the presentation, we first assume that all the path expressions in q are single labels or disjunctions of such labels, and prove the theorem for this class of queries. Once the proof for this restricted case is established, we explain how to extend it to the general case.

Recall that, for a node v in q, Φ_v is the conjunction of all formulas labeling v's ancestors, including v. We also use the following notation.

Definition 3.7 *For a query $Q \in QL$, $v \in nodes(Q)$, and a tree $T \in \mathcal{T}_{\Sigma,\mathcal{D}}$.*

- $vars(v) = free(\phi_v)$.
- $Binds(v,T) = \{\theta \mid \theta : vars(v) \rightarrow nodes(T), \text{ and } T \models \Phi_v[\theta] \}$.

For convenience, we will view in the sequel $Binds(v,T)$ as a relation whose attributes are the variables names. We will apply to it usual relational algebra operators: projection on a subset X of $vars(v)$, denoted $\pi_X(Binds(v,T))$, and selection $\sigma_{x\theta y}(Binds(v,T))$ producing the set of tuples in $Binds(v,T)$ satisfying the condition $x\theta y$, where $x \in vars(v)$, and $y \in vars(v)$ or y is a constant, and $\theta \in \{=, \neq\}$.

To further simplify the presentation, we will assume first that, for every $v \in q$ the tags of v'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 the tags of the children of n_k is not in the regular language defined by the regular expression r_a specified by τ_2. Note that, by the definition of $q(T)$, n_k was contributed by some node $c \in nodes(q)$. If $a_1, \ldots, a_n$ are the tags of the children of v, w is in the language $\hat{r_a} = \neg r_a \cap a_1^* \ldots a_n^*$. We will use a characterization of the words in $\hat{r_a}$ in terms of the number of symbols $a_1, \ldots, a_n$ they contain. To this end, we prove the following.

Lemma 3.8 *Let $a_1, \ldots, a_n$ be distinct symbols and let $\nu = (i_1, j_1), ..., (i_n, j_n)$ be a vector of n pairs of natural numbers. We denote by L_ν the language consisting of all words of the form $a_1^{\alpha_1} \ldots a_n^{\alpha_n}$ where $\alpha_m \equiv i_m \bmod j_m$ when $j_m > 0$ and $\alpha_m = i_m$ if $j_m = 0$. For each regular language r over alphabet $\{a_1, \ldots, a_n\}$, there exists a finite set $Vec(r)$ of vectors of pairs of natural numbers as above such that $r \cap a_1^* \ldots a_n^* = \bigcup_{\nu \in Vec(r)} L_\nu$.*

Proof: Let $M = (Q, \Sigma, \delta, q_0, F)$ be a non-deterministic finite-state automaton accepting r, where Q is the set of states, $\Sigma = \{a_1, \ldots, a_n\}$, $\delta : Q \times \Sigma \to 2^Q$ is the transition mapping, q_0 is the start state and F the set of final states. The mapping δ is extended in the usual way to $Q \times \Sigma^*$. Let $\mathbf{Q}$ be the set of all sequences $\vec{q}$ of states $q_0 \ldots q_n$ for which there exists some word $w = u_1 \ldots u_n$ in $r \cap a_1^* \ldots a_n^*$, where $u_1 \in a_1^*, \ldots, u_n \in a_n^*$, such that $q_m \in \delta(q_{m-1}, u_m)$, $1 \leq m \leq n$. For each pair of states p, q and alphabet symbol a_m let $L_{pq}^m = \{a_m^h \mid q \in \delta(p, a_m^h)\}$, and for each vector $\vec{q} = q_0 \ldots q_n \in \mathbf{Q}$ let $L_{\vec{q}} = L_{q_0 q_1}^1 \ldots L_{q_{n-1} q_n}^n$. Clearly,

$$r \cap a_1^* \ldots a_n^* = \bigcup_{\vec{q} \in \mathbf{Q}} L_{\vec{q}}.$$

Next, consider any of the languages L_{pq}^m. This is a regular language over the singleton alphabet $\{a_m\}$. By the Pumping Lemma for regular languages, there exist positive k and i bounded by $|Q|$ such that, for each $w \in L_{pq}^m$ of length $> k$, $w(a_m^i)^* \subseteq L_{pq}^m$. Let $L_{\leq k}$ consist of the words in L_{pq}^m of length $\leq k$ and $L_{>k}$ consists of the words in L_{pq}^m of length $> k$. Clearly, $L_{pq}^m = L_{\leq k} \cup L_{>k}$. Let J be the set of equivalence classes j modulo i for which there exists some word w in $L_{>k}$ of length $j \ mod \ i$, and let w_j be the shortest such word. Clearly,

$$L_{pq}^m = L_{\leq k} \cup \bigcup_{j \in J} w_j (a_m^i)^*.$$

Thus, words in L_{pq}^m can be described using pairs of integers as follows. Each singleton word $w \in L_{\leq k}$ is described by the pair $(|w|, 0)$. Each language $w_j(a_m^i)^*$ consists of the words of length $|w_j| + \alpha$ where $\alpha \equiv 0 \ mod \ i$ so is described by the pair $(|w_j|, i)$. Thus, each language L_{pq}^m is a union of languages described by pairs of numbers. The lemma easily follows by distributing concatenation over union in each of the languages $L_{\vec{q}} = L_{q_0 q_1} \ldots L_{q_{n-1} q_n}$. Finally, note that the sizes of the integers involved in the vectors ν are linear in the number of states of M, so linear in r. □

Thus, $\hat{r_a}$ can be written as a union of languages L_ν as defined in Lemma 3.8. Furthermore, the sizes of the integers involved in the vectors ν are linear in $\neg r_a$ so at most exponential in the size of r_a. The following is immediate from the above discussion.

Proposition 3.9 *Let q be a QL query, T some input tree, and τ_2 some output DTD; $q(T)$ violates τ_2 iff there exists a node $c \in nodes(q)$ with some tag a and children $c_1, \ldots, c_n$ having tags $a_1, \ldots, a_n$ and free variables $\bar{x}_1, \ldots, \bar{x}_n$, and a vector $(i_1, j_1), ..., (i_n, j_n)$ in $Vec(\neg r_a \cap a_1^* \ldots a_n^*)$, such that*

(†) *$Binds(c, T)$ contains a tuple v where for all $l = 1 \ldots n$, $\pi_{\bar{x}_l}(\sigma_{vars(c)=v}(Binds(c_l, T)))$ is of size α for some positive integer α such that $\alpha \equiv i_l \ mod \ j_l$ when $j_l > 0$, and of size i_l when $j_l = 0$.* [1]

For brevity, we will refer in the sequel to the property (†) of $Binds(c, T)$ as the *modulo property.*

It thus remains to show that, given an input DTD τ_1, a query q, some node $c \in nodes(q)$, and a vector $(i_1, j_1), ..., (i_n, j_n)$ 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, whose size is bounded by some function of the sizes of τ_1, q, and the numbers $(i_1, j_1), ..., (i_n, j_n)$. Then, to decide if the above holds, we simply need to check all the trees up to that size. We show below that such a bound on the tree size indeed exists.

[1] Recall that we view $Binds(c, T)$ as a relation with attributes $vars(c)$.

Without loss of generality, assume that τ_1 contains no redundant symbols (i.e., every symbol occurs as a label in some tree satisfying τ_1). 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 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 due 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 as follows: whenever we say a *tree* we shall 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 at depth greater than $|q|$, to a tree that satisfies τ_1.

Let $T \in inst(\tau_1)$ be a tree of minimum size having the modulo property. Let N be the set of nodes in T consisting of the following (using the notation in (†)):

1. all the nodes in the vector v,

2. for every c_l where the relation
$$R_l = \sigma_{vars(c)=v}(Binds(c_l, T)))$$
is not empty, the nodes in some vector $r_l \in R_l$,

3. for every c_l with corresponding pair (i_l, j_l), all the nodes in some sub-relation of size i_l of
$$\pi_{\bar{x}_l}\sigma_{vars(c)=v}(Binds(c_l, T))),$$
and

4. all the nodes on the paths from the root to the nodes in items 1-3 above.

We next argue that the size of N is bounded by the input, independently of T. Indeed, (1) generates at most $|q|$ nodes; as the number of c's is bounded by $|q|$, (2) generates at most $|q|^2$ and (3) at most $i_l \times |q|^2$ nodes. Let the sum of these numbers be m. Then, taking into account (4), the size of N is bounded by $m \times |q|$. Clearly, N only depends on the input and not on T. 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 no longer belong to $inst(\tau_1)$. We will thus be interested only in deletion of nodes resulting in trees in $inst(\tau_1)$. To this end we define the notion of *deletable unit*, and *deletable set of units*.

Definition 3.10 *Given a tree $T \in inst(\tau_1)$, a sequence of full subtrees of T rooted at consecutive siblings in T is called a* unit. *A unit $\bar{n}$ is* deletable *if (i) the tree T' obtained from T by deleting all subtrees in $\bar{n}$ still belongs to $inst(\tau_1)$, and (ii) $\bar{n}$ contains no consecutive subsequence having property (i).*

Two units $\bar{n}$, $\bar{m}$ of T are *overlapping* if they have some common node. Otherwise, they are called *non-overlapping*. A set U of units of T is *deletable* if it consists of non-overlapping deletable units of T and for each subset U' of U, the tree obtained from T by deleting all nodes in the units of U' still satisfies τ_1.

We can show the following.

Proposition 3.11 *Let U be a maximal deletable set of units of T, not containing any of the nodes in N. The number of units in U is no less than $|T|/((|\tau_1| \times (|N|+1))^{|q|})$.*

Proof: It is sufficient to consider sets of deletable units $\bar{n}$ with the following property:

(†) Let $n_1 \dots n_k$ be the sequence of roots of the subtrees in $\bar{n}$, α the sequence of preceding siblings, β the sequence of siblings following it, a the label of their parent, and M_a the standard non-deterministic finite-state automaton for the regular expression associated to a by τ_1 (whose number of states is bounded by τ_1). There exists an accepting computation of M_a on input $\alpha\, n_1 \dots n_k\, \beta$ such that the state after reading α is the same as the one after reading $\alpha\, n_1 \dots n_k$, and is distinct from those reached after reading $\alpha\ n_1 \dots n_j$ for $1 \leq j < k$.

Note that any set U of deletable units satisfying (†) is a deletable set of units. Consider a maximal such set U not containing any of the nodes in N. By the Pumping Lemma for regular languages, the length of each deletable unit in U is at most $|\tau_1|$, and the number of siblings separating any two consecutive deletable units in U is also at most $|\tau_1|$. Similarly, a node in N has at most $|\tau_1|$ siblings between it and the next (previous) sibling belonging to N or to some deletable unit. Consider the maximal size of a tree not having any deletable unit. This is a tree where every node has as children nodes in N separated by at most $|\tau_1|$ nodes. So, the maximal size of such a tree is $(|\tau_1| \times (|N|+1))^h$, where h is the depth of the tree. Any node added to this tree is part of some deletable unit. In the case of T, each such additional node belongs, in the worst case, to one distinct deletable unit. For $c := (|\tau_1| \times (|N|+1))^{|q|-1}$, the number of such nodes is then at least $|T|/(c+1)$. Furthermore, since each deletable unit is of length at most $|\tau_1|$ this yields a deletable set of units of size at least $T/((c+1) \times |\tau_1|)$. Summing up, we have that T has a deletable set of units of size at least

$$T/(((|\tau_1| \times (|N|+1))^{|q|-1} + 1) \times |\tau_1|),$$

which is bounded by

$$T/((|\tau_1| \times (|N|+1))^{|q|}.$$

□

Corollary 3.12 *For each positive integer m, and each tree $T \in inst(\tau_1)$ of size larger than $m \times ((|\tau_1| \times (|N|+1))^{|q|})$, there exists a deletable set of units of T of size at least m, whose units do not contain any of the nodes in N.*

We will use Corollary 3.12 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 has a sufficiently large deletable set of units so that some subset can be removed without affecting the modulo property.

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

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

Note that each vector associated to a deletable set of units can be viewed abstractly as a color. We can then apply Ramsey's Theorem, (stated below for convenience), and its corollary 3.14. Indeed, it follows from Corollary 3.14 and from the defnition of deletable set of units that for every $m > 0$, if T is larger than some given bound (a function of the m, the given $j_l's$, $|\tau_1|$, and $|q|$, called the *Ramsey bound*) then there exists a deletable set X of m units in T, not including any of the nodes in N, such that for every integer $r \leq |q|$, 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).

To obtain our result, we make use of the above for a conveniently chosen value of m. Specifically, let $m = \Pi_{l=1...n} j_l \times k!$ (where $k = |q|$). If T is larger than the corresponding Ramsey bound, then it contains a deletable set X of m 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_l where, in the corresponding (i_l, j_l) tuple, $j_l = 0$, $\pi_{\bar{x}_l}(\sigma_{vars(c)=v}(Binds(c_l, T')))$ contains exactly i_l tuples.
- For every c_l where $\bar{x}_l$ is the empty vector

$$\pi_{\bar{x}_l}(\sigma_{vars(c)=v}(Binds(c_l, T'))) = \pi_{\bar{x}_l}(\sigma_{vars(c)=v}(Binds(c_l, T))).$$

  More specifically, the relations are either both empty or both contain a single 0-ary tuple. 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 projection results in one tuple - the empty tuple.

To show that T' has the modulo property, it remains to prove that for every c_l where $\bar{x}_l$ is not the empty vector, the relations $\pi_{\bar{x}_l}(\sigma_{vars(c)=v}(Binds(c_l, T')))$ is of size $(\alpha \times j_l + i_l)$ for some positive integer α. Rather than computing the exact size of this relation, we will compute the number of vectors deleted from

$$\pi_{\bar{x}_l}(\sigma_{vars(c)=v}(Binds(c_l, T))).$$

If we show that this number is zero modulo j_l, then we are done.

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

$$n_l = m \times c_l^1 - \binom{m}{2} \times c_l^2 + \binom{m}{3} \times c_l^3 - ... + / - \binom{m}{|X|} \times c_l^{|X|},$$

which we explain next. The formula first counts separately for each node in a deletable unit in X how many tuples are deleted in $\pi_{\bar{x}_l}(\sigma_{vars(c)=v}(Binds(c_l, T)))$ when the node is removed. Note that according to the corollary to 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 at most $m \times c_l^1$. However, this is an overestimate: some tuples, (i.e. those containing two or more nodes in X) are counted several times. To fix this we subtract for every pair of nodes the number of tuples in which the two nodes appear together. There are $\binom{m}{2}$ such pairs of nodes, each appearing, (again according to the corollary to Ramsey's 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 $\binom{m}{2} \times c_l^2$. Note that this time we deducted too much: some tuples (i.e. those containing 3 or more nodes in X) where counted several times. To compensate, we add for every triplet of nodes the number of tuples in which the three nodes all appear. As above this is $\binom{m}{3} \times c_l^3$, 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 that size is reached.

Since we chose m to be the number $\Pi_{l=1...n} j_l \times k!$ (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, we state Ramsey's Theorem, and the corollary used above.

Theorem 3.13 *(Ramsey's Theorem) ([18], see also [32], pp.7-9) For all natural numbers k, m, w there exists a finite number $R(k, m, w)$ such that for every set Y of elements with $|Y| \geq R(k, m, w)$ 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 $|X| = m$ where all the subsets of X of size k have the same color.*

Furthermore, the number $R(k, m, w)$ is computable but of size non-elementary with respect to k, m, w. The following variant is an easy consequence of Ramsey's Theorem.

Corollary 3.14 *[18] For all natural numbers k, m, w there exists a finite number $R'(k, m, w)$ such that for every set Y of elements of size $|Y| \geq R'(k, m, w)$ 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 $|X| = m$ 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).*

Proof: The proof is by induction on k. For $k = 1$ the statement follows immediately from Ramsey theorem. Now, assume correctness for $k - 1$ (and every m, w). We prove the statement for k. Define $R'(k, m, w) == R(k, m, w)$ for $k = 1$ and $R'(k, m, w) == R(k, R'(k-1, m, w), w)$ for $k > 1$. Note that $R'(k-1, m, w)$ exists by the induction hypothesis and $R(k, R'(k-1, m, w), w)$ exists by Ramsey's theorem. Now, if $|Y'| \geq R'(k, m, w)$ then, by the definition of R', for every coloring of subsets of Y' there is some $X' \subset Y'$ of size $|X'| = R'(k-1, m, w)$ where all the subsets of size k have the same color. There is some $X \subset X'$ of size $|X| = m$ where for every $k' \leq k - 1$ all the subsets of size k' have the same color. But all subsets of size k of X also have the same color since $X \subseteq X'$ and all subsets of size k of X' have the same color. $\Box$

Observe that, in the proof above, we simply need to consider each possible vector attached to a deletable set of units as a color. The number w of available colors is then simply $j_1 \times ... \times j_n$; $k = |q|$; and, $m = \Pi_{l=1...n} j_l \times k!$. Now, recall from Corollary 3.12 that every tree T of size larger than $R'(k, m, w) \times ((|\tau_1| \times (|N| + 1))^{|q|})$ has a deletable set of units of size at least $R'(k, m, w)$, whose units do not contain any of the nodes in N. The rest follows immediately from Corollary 3.14.

To conclude, recall that in the course of the above proof we made two simplifying assumptions on the structure of the query q: (1) We assumed for every node $c \in nodes(q)$, the tags of c's children are all distinct, and (2) we assumed that all the path expressions in q are single labels or disjunctions of such labels. It is therefore left to remove these restrictions and extend the proof to the general case.

Repeated tags: Let q be a query and τ an output DTD. As in the proof of Theorem 3.3, we construct from q a query $\bar{q}$ by replacing all tags $a_1, \ldots, a_k$ of children of a node labeled a by *distinct* tags $b_1, \ldots b_k$. We also construct from τ a new DTD $\bar{\tau}$ by replacing the regular expression φ_a by $h^{-1}(r_a) \cap b_1^* \ldots b_k^*$ where h is the mapping $h(b_i) = a_i$, $1 \leq i \leq k$. It is easy to see that q typechecks with respect to τ iff $\bar{q}$ typechecks with respect to $\bar{\tau}$ (the input DTD remains unchanged).

General path expressions: A path expression in non-recursive queries represents a finite union of simple paths (rather than just a union of single labels, as we assumed so far). We next explain how the proof can be extended to accommodate those. Consider a node $c \in nodes(q)$, whose attached formula contains such a path expressions. The first step is to take these (unions of) simple paths in the query and make them, and each of the edges they contain, "explicit", by attaching a new variable name to each node along those paths. Each conjunct with such a path expression becomes a disjunction of conjunctions, describing the possible paths and the edges they contain.

Consider for instance the formula below (similar but not identical to Example 2.5).

$$\begin{array}{lll} title(X_0, X_1, X_2, X_3) & :- & X_0 \ catalog.movies \mid cinema.info \ X_1, \\ & & X_1 \ title \ X_2, \\ & & X_1 \ director \ X_3, \\ & & X_3 = \text{“}W.Allen\text{”} \end{array}$$

After transforming the path expression in the first conjunct we obtain the following.

$$\begin{array}{lll} title(X_0, X_1, X_2, X_3) & :- & (\ (\ X_0 \ catalog \ Z_1, Z_1 \ movies \ X_1 \ ) \vee (\ X_0 \ cinema \ Z_2, Z_2 \ info \ X_1 \ ) \ ), \\ & & X_1 \ title \ X_2, \\ & & X_1 \ director \ X_3, \\ & & X_3 = \text{“}W.Allen\text{”} \end{array}$$

Correspondingly, the notion of *variables binding* is extended in the natural way to contain the new variables. The only point to note is that since our XML data consists of trees, and each node is reachable from the root by a *single path*, in each of the bindings some of the variables may not have "real nodes" assigned to them (i.e. the variables corresponding to the "non used" paths). but instead some arbitrarily chosen null value.

Finally, observe that since each tree node uniquely determines its ancestors, the bindings to the new variables functionally depend on those of the original query variables. It follows immediately that one can replace the head $f(\bar{x})$ of each formula by by $f(\bar{x}, \bar{z})$ where $\bar{z}$ are the new variables and the query result will remain the same. For instance, in the above rule we could replace $title(X_0, X_1, X_2, X_3)$ by $title(X_0, X_1, X_2, X_3, Z_1, Z_2)$ without changing the query result.

The resulting query is projection-less and the proof proceeds from here on along the same lines as above.

This concludes the proof of Theorem 3.6.

4 More on Complexity

Theorems 3.2 and 3.3 provide an upper bound of CO-NEXPTIME 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 DTDs. For example, relational databases can naturally be represented by

DTDs of depth at most 2 (the root has depth zero). We can show the following using the proofs of Theorems 3.2 and 3.3.

Corollary 4.1 *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*.*

Proof: The size of the smallest counterexample is now polynomial in the output, but testing whether a candidate input is indeed a counterexample requires PSPACE. □

Once again, it remains open whether the above complexity is tight. To show some lower bounds, we first make the following remark.

Remark 4.2 It is not so difficult to see that the usual conjunctive queries over a relational schema can be simulated in QL. Indeed, we only need to specify a tree representation of a database instance and to translate the conjunctive queries into a QL query. Let $\overline{R} := R_1, \ldots, R_n$ be a relational schema, i.e. a set of relation names with associated arities. We specify the tree representing the database by the DTD $\tau_{\overline{R}}$ as follows: $root \to R_1^* \cdots R_n^*$, for each $i \in \{1, \ldots, n\}$, $R_1 \to A_1, \ldots, A_{n_i}$ where n_i is the arity of R_i, and $A_i \to \varepsilon$ for each A_i. Note that this corresponds to a standard encoding of the relation R as an XML tree. Let $q(\bar{x})$ be a conjunctive query. We can assume without loss of generality that the relational literals in $q(\bar{x})$ contain distinct, non-repeating variables (all equalities are stated explicitly by equality atoms). Then define $q^{QL}(\bar{X})$ as the formula obtained from $q(\bar{x})$ by replacing each occurrence of a literal of the form $R_i(x_1, \ldots, x_{n_i})$ by $X_0\, R_i\, Y$, $Y\, A_1\, X_1, \ldots, Y\, A_{n_i}\, X_{n_i}$, and occurrences of literals of the form $x_i = x_j$ by $X_i = X_j$. Recall that X_0 is mapped to the root. Further, Y here is a fresh variable; we tacitly assume that Y is a different variable for every literal $R_i(x_1, \ldots, x_{n_i})$. For a simple illustration assume R is binary, then a query $q(x_1, x_2)$ and its encoding $q^{QL}(x_1, x_2)$ are shown below, both in datalog notation:

$$\begin{array}{rcl} q(x_1, x_2) & \leftarrow & R(x_1, z_1), R(z_2, x_2), z_1 = z_2; \\ q^{QL}(X_1, X_2) & \leftarrow & X_0\ R\ Y_1,\ Y_1\ A_1\ X_1,\ Y_1\ A_1\ Z_1, \\ & & X_0\ R\ Y_2,\ Y_2\ A_1\ Z_2,\ Y_2\, A_2\ X_2, Z_1 = Z_2 \end{array}$$

We make use of the above translation in proof of the next theorem and in Section 6.

Theorem 4.3 *Typechecking non-recursive QL queries without tag variables with respect to input DTD of depth ≤ 2, and unordered output DTDs is:*[2]

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

(ii) DP*-hard for QL queries with equalities on data values; and,*

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

Proof: (i) We reduce validity of propositional formulas to the typechecking problem. Let φ be a propositional formula using variables $x_1, \ldots, x_n$. Consider the input DTD

$$\begin{array}{rcl} root & \to & A_1? \ldots A_n? \\ A_i & \to & \epsilon, \quad 1 \leq i \leq n. \end{array}$$

[2]The complexity class Π_2^p is CO-NP$^{\text{NP}}$. Recall that DP properties are of the form $\sigma_1 \wedge \sigma_2$ where $\sigma_1 \in$ NP and $\sigma_2 \in$ CO-NP. We refer the reader to [30] for more information.

$$\text{answer}()$$

$$B_1(Y) \leftarrow X\ A_1\ Y \qquad B_2(Y) \leftarrow X\ A_2\ Y \qquad \ldots \qquad B_n(Y) \leftarrow X\ A_n\ Y$$

Figure 4: The query for propositional validity

$$a()$$

$$b(\bar{X}) \leftarrow q_1^{QL}(\bar{X})$$

$$c(\bar{X}) \leftarrow q_2^{QL}(\bar{Y}) \wedge \bar{X} = \bar{Y}$$

Figure 5: QL query used in the proof of Theorem 4.3

A derivation tree of this DTD represents a truth assignment of the propositional variables $x_1, \ldots, x_n$: A_i is present iff $x_i = 1$, for $i = 1, \ldots, n$.

Consider the query q represented in Figure 4. The query's variables are $\{X, Y\}$ with X denoting the root.

The query is basically the identity function, modulo some label renaming, whose sole purpose is to make the query easier to read: i.e., the query maps an input tree into an isomorphic output tree. Label B_i is present in the output iff the label A_i is present in the input. We now define the output type as an SL formula for *answer*, by checking whether the set of B_i's in the output corresponds to a truth assignment that makes φ true. More precisely the SL formula is obtained from φ as follows: each positive literal x_i is replaced with $B_i^{=1}$ and each negative literal $\neg x_i$ is replaced with $B_i^{=0}$, $1 \leq i \leq n$, while all other connectives $\vee, \wedge, \neg$ are preserved. For a simple illustration, if the propositional formula is:

$$\varphi \equiv (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$$

then we define the output DTD:

$$answer \rightarrow (B_1^{=1} \vee B_2^{=0}) \wedge (B_1^{=0} \vee B_3^{=1}) \wedge (B_1^{=1} \vee B_2^{=1} \vee B_3^{=0})$$

Clearly, φ is valid iff q typechecks with respect to τ_1 and τ_2.

(*ii*) The proof 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 describe the subquery corresponding to conjunctive query containment. Consider two conjunctive queries $q_1(\bar{x}), q_2(\bar{x})$ over relation R of arity k, with the same set of free variables $\bar{x}$. We take τ_R as the input DTD where τ_R is as defined in Remark 4.2. Next, we construct a QL query q as shown in Figure 5. The formulas $q_1^{QL}(\bar{X})$ and $q_2^{QL}(\bar{X})$ are defined as in Remark 4.2.

Consider some relation instance R and let T be its encoding as a tree. We denote by $q_1(R)$ and $q_2(R)$ the answers of q_1, q_2 on R ($q_1(R), q_2(R)$ are sets of tuples) and denote by $q(T)$ the result of applying q to T (a tree). The tree $q(T)$ has a root labeled a and several b-labeled children, one for each $\bar{x}$ in $q_1(R)$. If $\bar{x}$ is also in $q_2(R)$ then that b node has a child labeled c; otherwise it has no children. Hence $q_1(R) \subseteq q_2(R)$ iff $q(T)$ is valid w.r.t. the following unordered output DTD: $a \rightarrow b^{\geq 0}$, $b \rightarrow c^{\geq 1}$ and, consequently, $q_1 \subseteq q_2$ iff q typechecks w.r.t. this output DTD.

The proof of (*iii*) is by reduction of containment of conjunctive queries with inequalities, which is known to be Π_2^p-complete [34]. 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:

Proposition 4.4 *Typechecking non-recursive QL queries without conditions on data values and without tag variables with respect to input DTD of depth ≤ 2 and star-free output DTDs (using FO sentences) is* PSPACE*-hard.*

Proof: We use a reduction from the Quantified 3SAT problem, known to be PSPACE-complete [16]. 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:

$$\begin{array}{rcl} root & \rightarrow & X_{10}X_{11}\ldots X_{n0}X_{n1} \\ X_{ij} & \rightarrow & \epsilon \end{array}$$

$1 \leq i \leq n, 0 \leq j \leq 1$. Valid input trees correspond to strings of length $2n$. Let q be the identity query, which returns the input tree unchanged (see the proof of Theorem 4.3 for an example of such a query). Hence the output tree is the same string of length $2n$: $X_{10}X_{11}X_{20}X_{21}\ldots X_{n0}X_{n1}$. We construct the output star-free DTD $\bar{\psi}$ such that the output typechecks iff formula ψ is true. Recall that a star-free regular expression is a first-order logic formula over the vocabulary $(<, O_{X_{ij}})$, $i = 1, \ldots, n$, $j = 0, 1$. We obtain $\bar{\psi}$ from ψ by replacing recursively every sub-formula of the form $\exists x_i \alpha$ with $\exists y_i (O_{X_{i0}}(y_i) \vee O_{X_{i1}}(y_i) \wedge \alpha)$ and each $\forall x_i \alpha$ with $\forall y_i ((O_{X_{i0}}(y_i) \vee O_{X_{i1}}(y_i)) \rightarrow \alpha)$. Further, we replace each occurrence of x_i ($\neg x_i$) in φ by $O_{X_{i1}}(y_i)$ ($O_{X_{i0}}(y_i)$). Clearly, the size of $\bar{\psi}$ is polynomial in that of 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. □

5 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); and

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

We also consider a variation of (1), where QL queries are further restricted to use only a limited form of nesting provided by Skolem functions, but allow disjunction of single labels in path expressions (i.e., expressions of the form a or $a + b$, for $a, b \in \Sigma$). Together with the previous decidability results, these results 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 is a single symbol in Σ.

Theorem 5.1 *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 functional dependencies (FDs) and inclusion dependencies (IDs) [24, 6]. Assume the relational schema only consists of one relation R of some arity k. A functional dependency is a rule of the form $X \rightarrow Y$ where X and Y are sets of coordinates, that is, subsets of $\{1, \ldots, k\}$. A relation satisfies the FD $X \rightarrow Y$ if whenever two tuples agree on X they should also agree on Y. Clearly, every set F id FDs is equivalent to one where all FDs have singleton right-hand sides. An inclusion dependency is of the form $[i_1, \ldots, i_n] \subseteq [j_1, \ldots, j_n]$ where each $i_\ell, j_\ell \in \{1, \ldots, k\}$. A relation satisfies the ID $[i_1, \ldots, i_n] \subseteq [j_1, \ldots, j_n]$ if $\pi_{i_1,\ldots,i_n}(R) \subseteq \pi_{j_1,\ldots,j_n}(R)$. Here, π denotes the usual projection of the relational algebra.

The implication problem for FDs and IDs is a particular instance of the following problem. Given a set $D = \{f_1, f_2, \ldots, f_p\}$ of FDs and inclusion dependencies over some k-ary relation R, and f_0 an FD over R, decide whether every relation that satisfies all dependencies in D also satisfies f_0. It is known that the latter problem is undecidable [24, 6]. First, we translate every dependency into containment of conjunctive queries as follows. Let $f = X \rightarrow Y$ be an FD with $X = \{i_1, \ldots, i_n\}$ and $Y = \{j_1, \ldots, j_m\}$. Then define the following conjunctive queries:

$$\begin{aligned} q_{1,f}(x_{i_1}, \ldots, x_{i_n}, x_{j_1}, \ldots, x_{j_m}, y_{j_1}, \ldots, y_{j_m}) &\leftarrow R(x_1, \ldots, x_k), R(y_1, \ldots, y_k), x_{i_1} = y_{i_1}, \ldots, x_{i_n} = y_{i_n}; \\ q_{2,f}(x_{i_1}, \ldots, x_{i_n}, x_{j_1}, \ldots, x_{j_m}, y_{j_1}, \ldots, y_{j_m}) &\leftarrow R(x_1, \ldots, x_k), x_{j_1} = y_{j_1}, \ldots, x_{j_m} = y_{j_m}. \end{aligned}$$

Note that $q_{1,f}(R) \subseteq q_{2,f}(R)$ iff R satisfies f. Here, $q(R)$ denotes the set of tuples obtained by evaluating q on R.

Let $f = [i_1, \ldots, i_n] \subseteq [j_1, \ldots, j_n]$ be an ID. Then define the following conjunctive queries:

$$\begin{aligned} q_{1,f}(x_{i_1}, \ldots, x_{i_n}) &\leftarrow R(x_1, \ldots, x_k); \\ q_{2,f}(x_{j_1}, \ldots, x_{j_n}) &\leftarrow R(x_1, \ldots, x_k). \end{aligned}$$

Again, note that $q_{1,f}(R) \subseteq q_{2,f}(R)$ iff R satisfies f. Thus, the implication problem for FDs and inclusion dependencies is a particular instance of the following problem: given $p+1$ pairs of queries (q_i, r_i), $i = 0, 1, \ldots, p$, decide whether for every relational instance R:

$$\Big(\bigwedge_{i=1,p} q_i(R) \subseteq r_i(R)\Big) \Longrightarrow q_0(R) \subseteq r_0(R) \qquad (3)$$

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 Eq.(3) holds for every instance R. As input DTD we take $\tau_1 = \tau_R$ where τ_R is as defined in Remark 4.2. The QL query is given in Figure 6. Here, each q_i^{QL} and r_i^{QL} is constructed from q_i and r_i as specified in Remark 4.2. The output DTD τ_2 is designed to correspond to Eq.(3) and says: "either for every $i = 0, \ldots, p$, every b_i has a child labeled c_i, or there exists some $i = 1, \ldots, p$ such that at least one node labeled b_i has no children". This can be expressed as a DTD with specializations in a straightforward way. Indeed, define the specialized DTD $\tau_2 = (\Sigma, \Sigma', \tau', \mu)$ as follows:

$$\Sigma = \{a, b_0, \ldots, b_p, c_0, \ldots, c_p\},$$

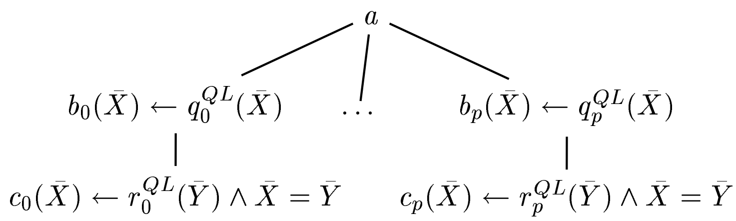


Figure 6: The QL program q of the proof of Theorem 5.1.

and

$$\Sigma' = \{a, b_0^0, \dots, b_p^0, b_0^1, \dots, b_p^1, c_0, \dots, c_p\}.$$

Here, b_i^0 and b_i^1 indicate that b_i has no child and one child, respectively. The DTD τ' consists of the rules:

- $a \to (\bigwedge_{i=0}^{p} (b_i^1)^{\geq 0}) \vee (\bigvee_{i=1}^{p} (b_i^0)^{\geq 1})$,
- $b_i^0 \to \varepsilon$, $b_i^1 \to c_i^{=1}$, $c_i \to \varepsilon$, $0 \leq i \leq p$.

The mapping μ is defined by $\mu(a) = a$, for each $i \in \{0, \dots, p\}$, $\mu(b_i^0) = \mu(b_i^1) = b_i$ and $\mu(c_i) = c_i$. $\square$

Similar undecidability results can be shown for slightly different combinations of features, which highlight rather subtle trade-offs. For example, consider the use of nested queries in QL. Consider a node n in a QL query, labeled by the rule $P(\bar{X}) \leftarrow \varphi$. Suppose the node n has a child n' labeled with $P'(\bar{X}, \bar{Y}) \leftarrow \varphi'$. The node n' can be viewed as a nested query, parameterized by bindings of $\bar{X}$ inherited from its parent n. The language QL adopts a flexible nesting semantics whereby a node generated by n may appear in the answer for a given binding of $\bar{X}$ even if n' does not generate any nodes for that binding. This is disallowed in some XML query languages, where all nodes specified in the query tree must be present in every non-empty answer. We refer informally to this nesting semantics as “nesting without optional nodes”, and to the QL nesting semantics as “nesting with optional nodes”. Note how the proof of Theorem 5.1 crucially relies on nesting with optional nodes. One might wonder if this is in fact essential to undecidability. We next show that it is not: undecidability continues to hold even if optional nodes are disallowed, so long as tag variables are allowed, and the path expressions in formulas may use disjunctions of single labels.

We formalize next the variant of QL without optional nodes. For each QL query Q, we define the query $Q^{\wedge}$ obtained from Q as follows. First, we rename variables so that formulas occurring in distinct subtrees under the root of Q have disjoint sets of variables. (Recall that the sets of existential variables of formulas along the same path in the query tree are already disjoint.) Next, we replace each formula φ in a label of a node of Q other than the root by the conjunction of all formulas occurring in Q. The semantics of Q without optional nodes is then defined as the semantics of $Q^{\wedge}$ under usual QL semantics. Clearly, the answer to $Q^{\wedge}$ on any given input consists either of the root alone, or all nodes specified in the query tree are present in the answer. As a simple example consider the input DTD:

$$\begin{array}{lcl} \text{root} & \to & a \\ a & \to & b^* \end{array}$$

Consider the QL query Q:

$$\begin{array}{c} root \ \leftarrow \text{true} \\ | \\ a(X) \ \leftarrow \ Z\ a\ X \\ | \\ b(X,Y) \ \leftarrow \ X\ b\ Y \end{array}$$

With the usual QL semantics allowing nesting with optional nodes, Q simply copies its input. Note that a node a is output even if it does not have a b child. The semantics without optional nodes would only output a node a if it has at least one b child. The query $Q^\wedge$ enforcing this is:

$$\begin{array}{c} root \ \leftarrow \text{true} \\ | \\ a(X) \ \leftarrow \ Z\ a\ X,\ X\ b\ Y \\ | \\ b(X,Y) \ \leftarrow \ Z\ a\ X,\ X\ b\ Y \end{array}$$

Clearly, $Q^\wedge$ evaluated under usual QL semantics is equivalent to Q evaluated under the no-optional-nodes semantics.

Note that, by definition, all nodes in $Q^\wedge$ other than the root are labeled with the same formula, say ψ. To avoid repeating ψ at every node, one could adopt an alternative syntax that separates the formula from the query tree, such as

$$\texttt{where}\ \psi\ \texttt{construct}\ T$$

where T is $Q^\wedge$ with ψ dropped from node labels. Thus, node labels specify only the heads of rules, providing the tag and free variables. Note the similarity with XML-QL syntax using Skolem functions.

We call a QL program *disjunctive* if the path expressions in its formulas are of the form a or $a + b$ where a, b are single symbols. We can show the following.

Theorem 5.2 *Typechecking is undecidable for disjunctive QL queries without optional nodes, with tag variables, without inequality, unordered input DTDs of depth ≤ 2, and unordered output DTDs with specialization.*

Proof: We use again a reduction from the implication problem for functional and inclusion dependencies [24, 6]. However, the straightforward construction used in Theorem 5.1 no longer works, due to the more restricted semantics disallowing optional nodes. Let R be a k-ary relation, Δ a set of FDs and IDs over R, and f an additional FD over R. We assume without loss of generality that all FDs in $\Delta \cup \{f\}$ have singleton right-hand sides. We construct from (Δ, f) an unordered input DTD τ_1, a disjunctive QL query Q with tag variables and without inequality, and an unordered specialized output DTD τ_2 such that $\Delta \models f$ iff $Q^\wedge$ typechecks with respect to τ_1 and τ_2. In fact, the construction of Q will guarantee that Q and $Q^\wedge$ are equivalent under QL semantics. In other words, the usual QL semantics of Q coincides with the more restrictive semantics disallowing optional nodes.

As before, the input DTD will encode the relation R. However, for technical reasons, the input also represents explicitly the projections of R on certain subsets of its attributes involved in the FDs or IDs. The query Q together with the output DTD will allow verifying that these representations are correct. Let $\mathcal{F}$ denote the FDs in $\Delta \cup \{f\}$ and $\mathcal{I}$ the IDs in Δ. The input DTD uses the following alphabet:

- $root$, R, $1, \ldots, k$,
- $c_1 \ldots c_k$ where the c_i are new symbols,
- F_σ for each $\sigma \in \mathcal{F}$
- I_σ, J_σ for each $\sigma \in \mathcal{I}$.

and has the following rules:

$$\begin{array}{lll}
root & \rightarrow & R^{\geq 1} \wedge \bigwedge_{\sigma \in \mathcal{F}} (F_\sigma)^{\geq 1} \wedge \bigwedge_{\sigma \in \mathcal{I}} ((I_\sigma)^{\geq 1} \wedge (J_\sigma)^{\geq 1}) \\
R & \rightarrow & 1^{=1} \wedge \ldots \wedge k^{=1} \\
F_\sigma & \rightarrow & (i_0)^{=1} \wedge \ldots \wedge (i_n)^{=1} \text{ if } \sigma \text{ is } i_1 \ldots i_n \rightarrow i_0 \\
I_\sigma & \rightarrow & (i_1)^{=1} \wedge \ldots \wedge (i_n)^{=1} \wedge (c_1)^{=1} \wedge \ldots \wedge (c_n)^{=1} \text{ and} \\
J_\sigma & \rightarrow & (j_1)^{=1} \wedge \ldots \wedge (j_n)^{=1} \wedge (c_1)^{=1} \wedge \ldots \wedge (c_n)^{=1} \text{ if } \sigma \text{ is the ID } i_1 \ldots i_n \subseteq j_1 \ldots j_n \\
i & \rightarrow & \epsilon, \quad 1 \leq i \leq k \\
c_i & \rightarrow & \epsilon, \quad 1 \leq i \leq k
\end{array}$$

The additional labels $c_1, \ldots, c_n$ used in relations associated with IDs are needed for technical reasons that will become clear later on. The query Q has several subtrees under the root. Their roots are labeled as follows:

- $enc \leftarrow true$
- $\sigma \leftarrow true$ for each $\sigma \in \Delta \cup \{f\}$.

In conjunction with Q, the constraints placed by the output DTD τ_2 on the subtrees rooted at *enc* and σ ($\sigma \in \Delta \cup \{f\}$), respectively, will ensure that at least one of the following holds:

- the input does *not* represent a correct encoding of some non-empty relation R and its projections;
- some dependency $\sigma \in \Delta$ is violated;
- f is satisfied.

This and the equivalence of Q and $Q^\wedge$ imply that $Q^\wedge$ typechecks with respect to τ_1 and τ_2 iff $\Delta \models f$.

We next describe separately the subtrees of Q rooted at *enc* and each σ. The role of the subtree rooted at *enc* is to verify whether the input represents a correct encoding of some non-empty relation R and its projections. We denote by $tup(R)$ the set of tuples of data values of the children of each node labeled R in the input tree, and similarly for $tup(F_\sigma)$ where $\sigma \in \mathcal{F}$. If σ is an ID $[i_1 \ldots i_n] \subseteq [j_1 \ldots j_n]$, $tup(I_\sigma)$ denotes the set of tuples of data values of the children labeled $i_1, \ldots, i_n$ of each node labeled I_σ, and $tup(J_\sigma)$ denotes the set of tuples of data values of the children labeled $j_1, \ldots j_n$ of each node labeled J_σ. The following must hold in a correct encoding:

1. For nodes labeled R, F_σ, I_σ or J_σ, the tuples of data values of their children with labels among $1, \ldots, k$ are distinct for different nodes.
2. If σ is an FD $i_1 \ldots i_n \rightarrow i_0$, $tup(F_\sigma) = \pi_{i_0 \ldots i_n}(tup(R))$; if σ is an ID $[i_1 \ldots i_n] \subseteq [j_1 \ldots j_n]$, then $tup(I_\sigma) = \pi_{i_1 \ldots i_n}(tup(R))$ and $tup(J_\sigma) = \pi_{j_1 \ldots j_n}(tup(R))$.
3. If σ is an ID $[i_1 \ldots i_n] \subseteq [j_1 \ldots j_n]$, for each node labeled by I_σ, the tuple of data values of its children labeled by $i_1 \ldots i_n$ equals the tuple of data values of the children labeled $c_1 \ldots c_n$, and similarly for nodes labeled J_σ.

$$R\text{-}no\text{-}duplicates(\bar{X}) \leftarrow q_R(\bar{X})$$

$$|$$

$$R(\bar{X}, \bar{Y}) \leftarrow q_R(\bar{Y}), \bar{X} = \bar{Y}$$

Figure 7: Subquery detecting duplicate tuples.

$$F_\sigma\text{-}proj(X_0, \ldots, X_n) \leftarrow W\ F_\sigma\ Y, Y\ i_0\ X_0, \ldots, Y\ i_n\ X_n$$

$$|$$

$$Z(X_0, \ldots, X_n, Z) \leftarrow V\ (R + F_\sigma)\ Z, Z\ i_0\ Y_0, \ldots, Z\ i_n\ Y_n,\ Y_0 = X_0, \ldots, Y_n = X_n$$

Figure 8: First subquery checking correct projection encoding.

Each of (1)-(3) above is tested by a separate subquery sitting under *enc*. Consider (1). The subquery ensuring uniqueness of the tuples of data values represented by R is shown in Figure 7, where $q_R(X_1, \ldots, X_k)$ is the formula

$$Z_1\ R\ Z_2, Z_2\ 1\ X_1, \ldots, Z_2\ k\ X_k.$$

Note that *R-no-duplicates* always has at least one child R for every binding of $\bar{X}$. This is compatible with the semantics without optional nodes. The representation of R contains duplicate tuples iff some node labeled *R-no-duplicates* has more than one child R. Similar queries are used to detect duplicate tuples in F_σ, I_σ, and J_σ.

Next, consider (2). Satisfaction of (2) for each F_σ where σ is the FD $i_1 \ldots i_n \rightarrow i_0$ is checked by two queries: the first, represented in Figure 8, checks whether $tup(F_\sigma) \subseteq \pi_{i_0 \ldots i_n}(tup(R))$ and the second, represented in Figure 9, checks the reverse inclusion. Note that in Figure 8, each node labeled F_σ-*proj* always has at least one child. The inclusion holds if each node labeled F_σ-*proj* has at least one child labeled R. In Figure 9 the symmetric observation holds. Checking (2) is done using similar queries for relations I_σ and J_σ when σ is an ID.

Consider (3). Let σ be an ID $[i_1 \ldots i_n] \subseteq [j_1 \ldots j_n]$. The query verifying (3) for I_σ is shown in Figure 10. Note that (3) is satisfied iff each node labeled I_σ-*copy* has precisely 2^n children labeled R_{2^n}. A similar subquery verifies (3) for J_σ.

The portion of the output DTD used to check whether the input represents a correct encoding of R and its projections uses the following specialized alphabet:

- R, F_σ $(\sigma \in \mathcal{F})$, I_σ, J_σ $(\sigma \in \mathcal{I})$, R_{2^n};
- $(enc)_1$, and $(enc)_0$, signifying that the input is (resp. is not), a correct encoding satisfying (1)-(3) above;
- $(\chi\text{-}no\text{-}duplicates)_1$ and $(\chi\text{-}no\text{-}duplicates)_0$, meaning that χ satisfies (1) (resp. does not satisfy (1)), where $\chi \in \{R\} \cup \{F_\sigma \mid \sigma \in \mathcal{F}\} \cup \{I_\sigma, J_\sigma \mid \sigma \in \mathcal{I}\}$;

$$R\text{-}proj(X_0, \ldots, X_n) \leftarrow W\ R\ Y, Y\ i_0\ X_0, \ldots, Y\ i_n\ X_n$$

$$|$$

$$Z(X_0, \ldots, X_n, Z) \leftarrow V\ (R + F_\sigma)\ Z, Z\ i_0\ Y_0, \ldots, Z\ i_n\ Y_n, Y_0 = X_0, \ldots, Y_n = X_n$$

Figure 9: Second subquery checking correct projection encoding.

$$
\begin{array}{c}
I_\sigma\text{-}copy(Z, X_1, \ldots, X_n) \leftarrow W\ I_\sigma\ Z, Z\ i_1\ X_1, \ldots, Z\ i_n\ X_n \\
| \\
R_{2^n}(Z, X_1, \ldots, X_n, Y_1, \ldots, Y_n) \leftarrow Z\ (i_1 + c_1)\ Y_1, \ldots, Z\ (i_n + c_n)\ Y_n, Y_1 = X_1, \ldots, Y_n = X_n
\end{array}
$$

Figure 10: Subquery checking (3) for I_σ.

- $(F_\sigma\text{-}proj)_1$ for FDs σ, meaning that the inclusion $tup(F_\sigma) \subseteq \pi_{i_0 \ldots i_n}(tup(R))$ required by (2) is satisfied, $(F_\sigma\text{-}proj)_0$ meaning that the inclusion is violated, and similarly for $(\chi\text{-}proj)_\epsilon$ for IDs σ, $\chi \in \{I_\sigma, J_\sigma\}, \epsilon \in \{0, 1\}$;
- $(R\text{-}proj)_1$, and $(R\text{-}proj)_0$, meaning that a converse inclusion to the above is satisfied (resp. violated);
- for each ID $\sigma \in \mathcal{I}$, $(\chi\text{-}copy)_1$ and $(\chi\text{-}copy)_0$, meaning that (3) is satisfied (resp. violated) for $\chi \in \{I_\sigma, J_\sigma\}$.

The specialization mapping is the identity on R, F_σ, I_σ, J_σ, R_{2^n}, and maps every subscripted symbol $(\alpha)_\epsilon$ to α, for $\epsilon \in \{0, 1\}$. The rules of this portion of the output DTD are as follows:

- $(enc)_1 \rightarrow \psi_{no-duplicates} \ \wedge \ \psi_{projections} \ \wedge \ \psi_{copies}$ where:

$$
\begin{array}{lcl}
\psi_{no-duplicates} & = & (R\text{-}no\text{-}duplicates)_0^{=0} \wedge \bigwedge_{\sigma \in \mathcal{F}} (F_\sigma\text{-}no\text{-}duplicates)_0^{=0} \wedge \\
 & & \bigwedge_{\sigma \in \mathcal{I}} [(I_\sigma\text{-}no\text{-}duplicates)_0^{=0} \wedge (J_\sigma\text{-}no\text{-}duplicates)_0^{=0}] \\
\psi_{projections} & = & (R\text{-}proj)_0^{=0} \wedge \bigwedge_{\sigma \in \mathcal{F}} (F_\sigma\text{-}proj)_0^{=0} \wedge \bigwedge_{\sigma \in \mathcal{I}} [(I_\sigma\text{-}proj)_0^{=0} \wedge (J_\sigma\text{-}proj)_0^{=0}] \\
\psi_{copies} & = & \bigwedge_{\sigma \in \mathcal{I}} [(I_\sigma\text{-}copy)_0^{=0} \wedge (J_\sigma\text{-}copy)_0^{=0}]
\end{array}
$$

- $(enc)_0 \rightarrow \neg\psi_{no-duplicates} \ \vee \ \neg\psi_{projections} \ \vee \ \neg\psi_{copies}$,
- $(\chi\text{-}no\text{-}duplicates)_1 \ \rightarrow \ \chi^{=1}$ and
- $(\chi\text{-}no\text{-}duplicates)_0 \ \rightarrow \ \chi^{\geq 2}$ for $\chi \in \{R\} \cup \{F_\sigma \mid \sigma \in \mathcal{F}\} \cup \{I_\sigma, J_\sigma \mid \sigma \in \mathcal{I}\}$,
- $(\chi\text{-}proj)_1 \ \rightarrow \ R^{\geq 1}$ and
- $(\chi\text{-}proj)_0 \ \rightarrow \ R^{=0}$ for $\chi \in \{F_\sigma \mid \sigma \in \mathcal{F}\} \cup \{I_\sigma, J_\sigma \mid \sigma \in \mathcal{I}\}$
- $(R\text{-}proj)_1 \ \rightarrow \ \bigvee_{\sigma \in \mathcal{F}} F_\sigma^{\geq 1} \vee \bigvee_{\sigma \in \mathcal{I}} (I_\sigma^{\geq 1} \vee J_\sigma^{\geq 1})$
- $(R\text{-}proj)_0 \ \rightarrow \ \bigwedge_{\sigma \in \mathcal{F}} F_\sigma^{=0} \wedge \bigwedge_{\sigma \in \mathcal{I}} (I_\sigma^{=0} \wedge J_\sigma^{=0})$
- $(\chi\text{-}copy)_1 \ \rightarrow \ (R_{2^n})^{=2^n}$ for $\sigma \in \mathcal{I}$, $\chi \in \{I_\sigma, J_\sigma\}$
- $(\chi\text{-}copy)_0 \ \rightarrow \ \bigvee_{k < 2^n} (R_{2^n})^{=k}$ for $\sigma \in \mathcal{I}$, $\chi \in \{I_\sigma, J_\sigma\}$
- $\chi \ \rightarrow \ \epsilon$ for $\chi \in \{R, R_{2^n}\} \cup \{F_\sigma \mid \sigma \in \mathcal{F}\} \cup \{I_\sigma, J_\sigma \mid \sigma \in \mathcal{I}\}$

$$\sigma \leftarrow \text{true}$$

$$\text{LHS}_\sigma\text{-key}(X_1, \ldots, X_n) \leftarrow W\ F_\sigma\ Z,\ Z\ i_1\ X_1,\ \ldots\ Z\ i_n\ X_n$$

$$F_\sigma(X_1, \ldots, X_n, Y_1, \ldots, Y_n) \leftarrow V\ F_\sigma\ Y,\ Y\ i_1\ Y_1,\ \ldots,\ Y\ i_n\ Y_n,\ X_1 = Y_1,\ \ldots,\ X_n = Y_n$$

Figure 11: Subquery verifying an FD σ.

We next describe the subqueries and portion of the output DTD checking whether each FD or ID σ is satisfied by R, assuming that the input is a correct encoding of R and its projections. Let σ be an FD $i_1 \ldots i_n \to i_0$. The query used to verify σ is shown in Figure 11. Clearly, σ holds in R iff $i_1 \ldots i_n$ is a key in $\pi_{i_0 \ldots i_n}(R)$. Assuming that the input represents a correct encoding of R and its projections, σ holds iff the data values of the nodes labeled $i_1 \ldots i_n$ uniquely determine each node labeled F_σ in the input. This in turn holds iff each node labeled LHS$_\sigma$-key has exactly one child labeled F_σ in the answer to the above query.

Let σ be an ID $[i_1 \ldots i_n] \subseteq [j_1 \ldots j_n]$. Recall that every input representing a correct encoding satisfies (3) above for each ID σ. Using this property, σ can be checked by the query in Figure 12. Clearly, R satisfies σ iff each node labeled $I_\sigma^{c_1 \ldots c_n}$ has a node labeled J_σ.

We next define the portion of the output DTD corresponding to the subtrees of Q rooted at σ for some FD or ID σ. The additional symbols in the specialized alphabet are:

- $(\sigma)_1$ and $(\sigma)_0$, meaning that σ is satisfied (resp. violated);
- $(\text{LHS}_\sigma\text{-key})_1$, $(\text{LHS}_\sigma\text{-key})_0$, meaning that the left-hand side of an FD σ is a key in F_σ (resp. is not a key);
- $(I_\sigma^{c_1 \ldots c_n})_1$ and $(I_\sigma^{c_1 \ldots c_n})_0$, meaning that the projection of I_σ on $c_1 \ldots c_n$ is included in the projection of J_σ on $c_1 \ldots c_n$ (resp. is not).

As before, the specialization maps each symbol $(\alpha)_\epsilon$ to α, $\epsilon \in \{0, 1\}$. The rules for this portion of the DTD are:

- for each FD σ, $(\sigma)_1 \to (\text{LHS}_\sigma\text{-key})_0^{=0}$ and $(\sigma)_0 \to (\text{LHS}_\sigma\text{-key})_0^{\geq 1}$;
- $(\text{LHS}_\sigma\text{-key})_1 \;\to\; (F_\sigma)^{=1}$;
- $(\text{LHS}_\sigma\text{-key})_0 \;\to\; (F_\sigma)^{>1}$;
- for each ID σ, $(\sigma)_1 \to (I_\sigma^{c_1 \ldots c_n})_0^{=0}$ and $(\sigma)_0 \to (I_\sigma^{c_1 \ldots c_n})_0^{\geq 1}$
- $(I_\sigma^{c_1 \ldots c_n})_1 \;\to\; (J_\sigma)^{\geq 1}$;
- $(I_\sigma^{c_1 \ldots c_n})_0 \;\to\; (J_\sigma)^{=0}$;

Finally, the DTD rule for the root is:

$$\text{root} \;\to\; \epsilon \vee (enc)_0^{=1} \vee \bigvee_{\sigma \in \Delta} (\sigma)_0^{=1} \;\vee\; (f)_1^{=1}.$$

The above states that either the input is empty, or it is not a correct encoding of R and its projections, or at least one dependency in Δ is not satisfied, or f is satisfied. This ensures that Q typechecks with respect to τ_1 and τ_2 iff $\Delta \models f$. Note that by construction, Q and $Q^\wedge$ are equivalent. $\square$

$$\begin{array}{rl} \sigma & \leftarrow \text{true} \\ & \quad | \\ I_\sigma^{c_1\ldots c_n}(X_1,\ldots,X_n) & \leftarrow\ W\ I_\sigma\ Z,\ Z\ c_1\ X_1,\ \ldots\ Z\ c_n\ X_n \\ & \quad | \\ Y(X_1\ldots,X_n,Y) & \leftarrow V\ (I_\sigma + J_\sigma)\ Y,\ Y\ c_1\ Y_1,\ \ldots,\ Y\ c_n\ Y_n,\ X_1\ =\ Y_1,\ \ldots,\ X_n\ =\ Y_n \end{array}$$

Figure 12: Subquery verifying an ID σ.

We conclude the section by considering QL queries with recursion. All the decidability results of Section 3 assume non-recursive QL queries. We next show that removing this restriction immediately causes undecidability of typechecking, even with very simple output DTDs.

Theorem 5.3 *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, see. e.g., [17]) to typechecking a QL query with respect to an output DTD requiring a nonempty sequence of children under the root. Recall that an instance of PCP is a sequence of pairs $(u_1, v_1), \ldots (u_k, v_k)$ where $u_i, v_i \in \{a, b\}^+$. A solution for the instance is a sequence of indices $i_1 \cdots i_m$, $m \geq 1$, such that $u_{i_1} \cdots u_{i_m} = v_{i_1} \cdots v_{i_m}$. It is well-known that it is undecidable whether a PCP instance has a solution [17]. 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 $a_1 \ldots a_n$ is parsed as $u_{i_1} \ldots u_{i_m}$, and $v_{i_1} \ldots v_{i_m}$, respectively. For each $i, 1 \leq i \leq n$, the string x contains four consecutive positions $w(i)s(j)i_j a_i$, where a_i occurs within u_{i_j} and w and s are distinct tags. Note that there is an unbounded number of i and j values, so these cannot be represented as tags and are instead represented as data values of the nodes labeled w and s, respectively. On the other hand, the number of values of i_j and a_i is bounded, so these can be represented as tags. The string y is analogous for $v_{i_1} \ldots v_{i_m}$. 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 $u_1u_3u_2u_1 = v_1v_3v_2v_1 = ababbaababa$. The corresponding encoding is the string (read top-down and left-to-right):

$$\begin{array}{ll}
w(1)s(1)1a & w(1)s(1)1a \\
w(2)s(1)1b & w(2)s(2)3b \\
w(3)s(1)1a & w(3)s(2)3a \\
w(4)s(2)3b & w(4)s(2)3b \\
w(5)s(2)3b & w(5)s(2)3b \\
w(6)s(3)2a & w(6)s(2)3a \\
w(7)s(3)2a & w(7)s(3)2a \\
w(8)s(3)2b & w(8)s(3)2b \\
w(9)s(4)1a & w(9)s(3)2a \\
w(10)s(4)1b & w(10)s(3)2b \\
w(11)s(4)1a & w(11)s(4)1a \\
\$ & \#
\end{array}$$

The input DTD τ_1 we use is the following:

$$\begin{array}{lll}
root \rightarrow w & w \rightarrow s & s \rightarrow 1 + \ldots + k \\
i \rightarrow a + b & (1 \leq i \leq k) & a \rightarrow w + \$ + \# \\
b \rightarrow w + \$ + \# & \$ \rightarrow w & \# \rightarrow \epsilon
\end{array}$$

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

$$\begin{array}{l}
w(\alpha_1)s(\beta_1)i_{\beta_1}a_1 \ldots w(\alpha_n)s(\beta_n)i_{\beta_n}a_n\$ \\
w(\alpha'_1)s(\beta'_1)i_{\beta'_1}a_1 \ldots w(\alpha'_m)s(\beta'_m)i_{\beta'_m}a'_m\#
\end{array}$$

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

1. $\alpha_i \neq \alpha_j$ and $\alpha'_i \neq \alpha'_j$ for all $i \neq j$.

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 $i (1 \leq i \leq n)$, $a_i = a'_i$.

   Note that (1)-(5) imply that $n = m$,
   $\alpha_i = \alpha'_i$, $1 \leq i \leq n$, *and* $a_1 \ldots a_n = a'_1 \ldots a'_n$

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 $\{\alpha_1 \ldots \alpha_n\}$, and $a_{j_1} \ldots a_{j_2} = u_{i_{\beta_j}}$.

7. the property analogous to (6) 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 $\beta_{i_1} \neq \beta_{i_2}$ then there exists some $j_2 > j_1$ such that $\beta'_{j_1} \neq \beta'_{j_2}$.

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 $i_2 = min\{i \mid i > i_1, \beta_i \neq \beta_{i_1}\}$ and $j_2 = min\{j \mid j > j_1, \beta'_j \neq \beta'_{j_1}\}$.

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. For conciseness, 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 $i, 1 \leq i < n$, $a_{\nu(X_i)+1} \dots a_{\nu(X_{i+1})} \in r_{i+1}$, and for *some* $v \leq m$, $a_{\nu(X_n)+1} \dots a_v \in r_{n+1}$. Additionally, the specified (in)equalities among the data values must hold. Clearly, such queries can be expressed by QL queries of the form:

$$\begin{array}{c} \text{answer}() \\ | \\ \text{witness}(X_1, \dots, X_n) \leftarrow X_0\, r_1\, X_1,\; X_1\, r_2\, X_2,\; \dots,\; X_n\, r_{n+1}\, Z \end{array}$$

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 $w.X.*.\$.w.Y$, $X \neq Y$. 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.

□

6 Typechecking XML views of relational data

In another paper [1], we considered typechecking of transformations mapping relational data into XML. We briefly discuss the relation between that work and the typechecking of XML-to-XML transformations investigated in the present paper.

In [1] we investigate the typechecking problem for a family of query languages TreeQL($\mathcal{L}$) defining mappings from relations to labeled trees. The parameter $\mathcal{L}$ is a relational query language, such as the conjunctive queries, possibly extended with inequalities or negation. The syntax and semantics of TreeQL($\mathcal{L}$) queries are similar to those of QL, where the queries used in the query tree belong to $\mathcal{L}$. The typechecking problem is to verify whether, for given query $q \in$ TreeQL($\mathcal{L}$), relational constraints Σ, and output DTD τ, $q(I) \in sat(\tau)$ for each relational instance I satisfying Σ.

By Remark 4.2, there is an immediate translation from TreeQL($\mathcal{L}$) queries to QL queries when the language $\mathcal{L}$ is the set of conjunctive queries (CQ). Hence, the typechecking problem for TreeQL(CQ) queries in the absence of constraints can be reduced to the typechecking problem for QL queries. We can conclude that, subject to these restrictions, upper bounds from the present paper carry over to [1] and lower bounds from [1] carry over to the present setting. Because of this correspondence, Theorem 4.3 and Theorem 5.1 carry over from [1], and Theorem 3.6 carries over from the present paper to [1]. The proofs of Theorem 4.3 and Theorem 5.1 are adapted to the QL formalism and provided here for the convenience of the reader and to make the paper self contained.

Beyond the restricted connection just described, the ability to transfer results between [1] and the present framework is limited by the significant differences between the two settings. In [1] we show that typechecking still remains in CO-NEXPTIME when negation is added to CQ, with integrity constraints specified by FO($\exists^*\forall^*$) formulas, that is, first-order logic formulas of the form $\exists x_1 \cdots \exists x_n \forall y_1 \cdots \forall y_m \varphi$ where φ is quantifier-free. We also obtain a matching lower bound. However, negation seems to be required so the lower bound does not transfer to the present formalism. We did not consider negation in this paper as it is not present in XML transformation languages while it is a well-accepted extension of conjunctive queries in the relational setting. In [1], we also consider virtual nodes and more expressive DTDs.

Conversely, the full typechecking problem considered here is not reducible to the setting in [1]. For example, input DTDs cannot be expressed by the relational constraints considered in [1]. Furthermore, the language QL cannot be simulated in the languages TreeQL($\mathcal{L}$) studied in [1], due to the presence of recursion in the form of regular expressions used in QL queries.

7 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 infeasible. 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] N. Alon, T. Milo, F. Neven, D. Suciu, V. Vianu. Typechecking XML Views of Relational Databases. In *Proceedings of the 16th IEEE Symposium on Logic in Computer Science*, 421–430, 2001.

[2] 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/`.

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

[4] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over non-ranked alphabets. Hong Kong Univ. of Science and Technology Computer Science Center Research Report HKUST-TCSC-2001-05, 2001. Available at http://www.cs.ust.hk/tcsc/RR/2001-05.ps.gz.

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

[6] A. K. Chandra and M. Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM Journal on Computing*, 14(3):671–677, 1985.

[7] D. Chamberlin, J. Robie, and D. Florescu Quilt: An XML Query Language for Heterogeneous Data Sources, In *WebDB 2000* (selected papers), Lecture Notes in Computer Science, Springer-Verlag, 1-25, 2000).

[8] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: an XML query language, 2001. available from the W3C, `http://www.w3.org/TR/query`.

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

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

[12] 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.

[13] 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)*, Computer Networks 31(11-16): 1155-1169 (1999).

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

[15] 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`.

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

[17] J. Hopcroft and J.Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

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

[19] H. Hosoya and B. C. Pierce. XDuce: An XML Processing Language (Preliminary Report). In *WebDB 2000* (selected papers), Lecture Notes in Computer Science, Springer-Verlag, 226-244, 2000.

[20] H. Hosoya and B. C. Pierce. Regular Expression Pattern Matching for XML. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*,67-80, 2001.

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

[22] T. Milo and D. Suciu. Type Inference for Queries on Semistructured Data. In *Proc. ACM Symposium on Principles of Database Systems*, pp. 215–226, 1999.

[23] 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.

[24] J. C. Mitchell. The implication problem for functional and inclusion dependencies. *Information and Control*, 56(3):154–173, 1983.

[25] J. C. Mitchell. Type systems for programming languages. In *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 8, pages 367–458. Elsevier, Amsterdam, 1990.

[26] J. C. Mitchell. *Foundations for Programmng Languages.* MIT Press, 1996.

[27] M. Murata. Transformation of Documents and Schemas by Patterns and Contextual Conditions. In *Proceedings of Third International Workshop on Principles of Document Processing (PODP)*, Palo Alto, CA, pages 153-169, 1996.

[28] M. Murata. Extended path expressions for XML. In *Proceedings of 20th ACM Symposium on Principles of Database Systems*, pages 126–137, Santa Barbara, CA, May 2001.

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

[30] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[31] 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.

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

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

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

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