Typechecking XML Views of Relational Databases*†

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

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

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

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

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

Abstract

Motivated by the need to export relational databases as XML data in the context of the Web, we investigate the *typechecking* problem for transformations of relational data into tree data (XML). The problem consists of statically verifying that the output of every transformation belongs to a given output tree language (specified for XML by a DTD), for input databases satisfying given integrity constraints. The typechecking problem is parameterized by the class of formulas defining the transformation, the class of output tree languages, and the class of integrity constraints. While undecidable in its most general formulation, the typechecking problem has many special cases of practical interest that turn out to be decidable. The main contribution of this paper is to trace a fairly tight boundary of decidability for typechecking in this framework. In the decidable cases we examine the complexity, and show lower and upper bounds. We also exhibit a practically appealing restriction for which typechecking is in PTIME.

1 Introduction

Since Codd [11], databases have been modeled as first-order relational structures and database queries as mappings from relational structures to relational structures. This captured well relational databases, where both data and query answers are represented as tables.

Today's technology trends require us to model data that is no longer tabular. The World Wide Web Consortium has adopted a standard data exchange format for the Web, called Extended Markup Language (XML) (see [1]), in which data is represented as a labeled ordered tree, rather than as a table. XML is rapidly becoming the de facto data format on the Web, and many industries (e.g. financial, manufacturing, health care) are migrating their application-specific formats to XML. All major database vendors now offer tools for exporting relational data as XML, thus making it easier for companies to define XML views of their relational data and share it with business partners over the Web. An important aspect of XML is that it allows users to define *types*. A type is a tree language, and

*Work supported in part by the U.S.-Israel Binational Science Foundation under grant number 97-00128.
†A preliminary version of this paper appeared in the proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science, Boston, MA, 2001.
‡This author supported in part by the National Science Foundation under grant number IIS-9802288.

the current standards for XML types (DTD and XML-Schema) correspond to restricted regular tree languages. XML data exchange is always done in the context of a fixed type: a community (or industry) agrees on a certain type, and subsequently all members of the community create XML views of their relational data that are of that type.

In this paper we study the problem of mapping relational data into tree data, specifically addressing the *typechecking* problem. Given a mapping and a type for the output tree, we wish to automatically check whether every database is mapped to a tree of the desired output type.

Our work addresses a real practical need. The current consensus is that most data processing will continue to be done by existing relational database systems, which over the decades have evolved into robust, highly scalable, and highly available systems. XML data will be generated dynamically, from the relational back-end, much like HTML pages are today generated dynamically. The mapping from the relational data into XML will be expressed in a declarative language. Research systems like SilkRoute [16] and Experanto [12, 32, 31] have pioneered languages to express such mappings, and all database vendors already offer such mappings with their systems. Such systems need to provide typecheckers for these mappings, in order to enable users to verify whether a given mapping produces XML outputs of the desired type. This is precisely the problem addressed in this paper. In addition to its practical importance, we will show that the problem is also technically interesting and non-trivial from a theoretical perspective.

Example 1.1. *To illustrate the typechecking problem, consider a car dealership called* LogiCar *that maintains an SQL database about used cars, containing two tables* `CAR(Name, Brand, Price)` *and* `STOCK(Name, Quantity)`*, with the following content.*

CAR	Name	Brand	Price
	Fiesta	Ford	300
	Golf	VW	450
	Focus	Ford	800

STOCK	Name	Quantity
	Fiesta	5
	Golf	7
	Focus	2

Further assume that several dealerships in the same geographical area, including LogiCar*, agree to exchange data between their database systems in order to facilitate cooperation. They agree on a common XML format having the following DTD:*

```
<!ELEMENT dealership (name, brand*)>
<!ELEMENT brand      (name, car*)>
<!ELEMENT car        (name, price, quantity)>
```

This DTD states that the exchanged XML files start with the dealership name and then, for each brand, contain the brand name and the brand's cars, including the car name, price, and available quantity. To illustrate such an XML data instance, LogiCar*'s data is exported as follows:*

```
<dealership>
   <name> LogiCar </name>
   <brand> <name> Ford </name>
           <car> <name> Fiesta </name>
                 <price> 300 </price>
                 <quantity> 5 </quantity>
           </car>
```

```
        <car> <name> Focus </name>
              <price> 800 </price>
              <quantity> 2 </quantity>
        </car>
   </brand>
   <brand> <name> VW </name>
        <car> <name> Golf </name>
              <price> 450 </price>
              <quantity> 7 </quantity>
        </car>
   </brand>
</dealership>
```

To generate this XML data from its relational database, LogiCar *might use the following SilkRoute query*[1]*:*

```
return
  <dealership>
       <name> LogiCar </name>
       for $B IN distinct(sqlDatabase()/CAR/Brand)
       return <brand>
                  <name> $B </name>
                  for $C IN sqlDatabase()/CAR
                  where $C/Brand = $B
                  return
                     <car> <name> $C/Name </name>
                           <price> $C/Price </price>
                           for $Q IN sqlDatabase()/STOCK
                           where $Q/name = $C/name
                           return <quantity> $Q/Quantity </quantity>
                     </car>
              </brand>
   </dealership>
```

The query starts by constructing a `<dealership>` *root element with a* `<name> LogicCar </name>` *child, then iterates over all distinct values of the* `Brand` *attribute in the* `CAR` *relation: here* `sqlDatabase()` *is the XML representation of the relational database, and the variable* `$B` *is bound successively to each distinct brand name in this relation. For each such value it constructs a* `<brand>` *element with a* `<name>` *child, then iterates again over all tuples in the* `CAR` *to retrieve all cars of that brand: a join is performed here with the condition* `$C/Brand = $B`*. For each such car it constructs a* `<car>` *element. Finally, a third level of iteration is needed to retrieve the quantities for that particular car, with a new join between the CAR and the STOCK relations:* `$Q/name = $C/name`*.*

The question is: does this query typecheck, i.e. does it always *return an XML document conforming to the DTD above ? The answer here is* no*. While the query, when applied*

[1]We follow here the newer SilkRoute syntax described in [17], which is based on XQuery [13]. The original SilkRoute language [16] was based on XML-QL [14].

on the particular relational instance presented above, does yield a file with the appropriate structure, this may not be the case for arbitrary instances. For example, if the relation `STOCK` *contains two tuples with the same car name and different quantities, the XML file generated by the query will contain a* `<car>` *element with two* `<quantity>` *subelements, thus violating the DTD.*

However, the query does typecheck if one enforces some constraints *on the relational database. For example,* LogiCar *may have the following two constraints:*

- `STOCK.Name` *is a key.*
- `CAR.Name` *is a foreign key to* `STOCK.Name`*.*

The first constraint ensures that each `<car>` *will have at most one* `<quantity>`*; the second constraint ensures that each* `<car>` *has at least one* `<quantity>`*. Together they ensure that the query typechecks.*

To study formally the typechecking problem, we have defined a language, TreeQL, which is an abstraction of the real practical mapping languages mentioned above, expressing mappings from relational structures to trees. A mapping m in TreeQL is specified as a tree where each node is labeled by a logical formula, possibly with free variables, and a symbol from a finite alphabet Σ. An ordered relational structure is mapped into a Σ-tree whose nodes consists of all tuples that satisfy some formula in the tree, and whose edges are defined based on the edges in m. In the typechecking problem we are given a regular tree language, called the *output type*, and a set of integrity constraints, and are asked to check whether every input structure satisfying the constraints is mapped into a tree in the output type. When the output type is a DTD, typechecking boils down to verifying whether the strings generated by the ordered sets of tuples satisfying a sequence of logical formulas belong to some regular language. The typechecking problem is parameterized by the fragment of TreeQL, the class of output types, and the class of integrity constraints.

The typechecking problem in its various instantiations requires an understanding of the interaction between logic and tree languages. We found this interaction interesting, and had to develop distinct approaches for the different instances of the typechecking problem, bringing into play techniques from finite-model theory, language theory, and combinatorics.

It is easily seen that typechecking becomes undecidable when arbitrary first-order logic (FO) formulas are allowed in the mapping, due to a reduction from the FO finite satisfiability problem. Hence, we focus our investigation on the particular case when the formulas are *conjunctive queries*. When the output types are further restricted to *star-free* regular languages, typechecking is decidable. When the output type is an arbitrary regular expression, typechecking is still decidable for *projection-free* conjunctive formulas (the proof uses a combinatorial argument based on Ramsey's Theorem). On the other hand, we show that even small extensions to the basic decidable cases lead to undecidability of typechecking. Thus, our results provide a fairly tight boundary of decidability of typechecking. A side benefit is new insight into the subtle interplay between constraints, query languages, and output tree types.

Related work. Type inference is a well-studied topic in functional programming languages [24]. A type inference system consists of a set of inference rules that can be used to check whether a function (program) is type safe. This means that during execution the program will never get into a state where it attempts to apply an operator to operands of wrong types. The problem we consider here is different.We are checking a semantic property, namely whether every input database is mapped to an output tree of the right type, which is in contrast to the syntactic nature of applying the type inference rules. In our setting, typechecking rapidly becomes undecidable if we allow the transformation language or the output types to be too expressive. In contrast, type inference for functional programming languages (that are Turing complete) is usually decidable for powerful type systems but is only sound.

Our work is motivated by the practical need to typecheck XML views of relational databases. SilkRoute [16] is a research prototype enabling an XML view to be defined from a relational database using a declarative language. The language TreeQL used in the present paper is an abstraction of the language used by SilkRoute.

A different but related problem is that of typechecking programs defining XML transformations. In previous work [22] a subset of the authors studied the typechecking problem for transformations of unranked trees expressed by k-pebble transducers, and showed that typechecking is decidable. The unranked trees considered there are abstractions of the basic nesting structure of XML documents. They are labeled over a fixed, finite alphabet Σ and so do not take into account the data values present in XML documents. The k-pebble transducer model captures the tree manipulation core of common XML languages, but also ignores data values. In subsequent work [3] we pursued this investigation by considering an extended framework accounting for data values. More precisely, XML documents are abstracted as trees whose nodes have associated data values from an infinite domain in addition to the tags. A transformation language QL is defined, that can make use of data values. It is shown that typechecking quickly becomes undecidable when QL programs can perform joins on data values. However, typechecking becomes decidable for several restrictions on the class of transformations and/or the tree types. While some of the techniques in [3] are similar in flavor to those in the present paper, there are considerable differences in the two settings. Indeed, relational structures can be encoded as XML, but the integrity constraints do not have an analog in XML. Conversely, the DTDs that constrain XML documents cannot be expressed by the relational constraints we consider. Furthermore, negation used in formulas of TreeQL programs do not have a natural analog in XML languages, and the use of path expressions in XML languages requires a form of recursion, which is not available in TreeQL programs. Despite this basic mismatch, some connections exist between the two frameworks. There is an immediate translation from TreeQL programs using conjunctive queries to programs in QL. Hence, the typechecking problem for such TreeQL programs in the absence of constraints can be reduced to the typechecking problem for QL queries. Consequently, some of the lower bound results in the present paper can be transferred to the XML context and strengthen results from [3]. Conversely, the only result that transfers from [3] to the present paper is the decidability result provided by Theorem 3.6. A direct, simplified proof of Theorem 3.6 is 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 [3] and the present framework is limited by the significant differences between the two settings.

Organization The paper is organized as follows. The first section develops the basic formalism, including our abstraction of XML documents, DTDs, and the transformation language TreeQL. Section 3 presents the decidability results; Section 4 the complexity analysis; and Section 5 the undecidability results. The paper ends with brief conclusions.

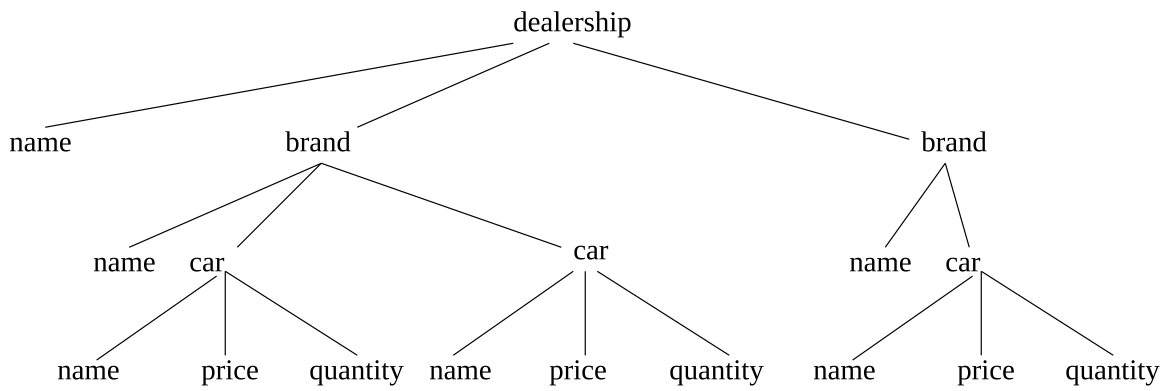


Figure 1: Dealer XML document

2 Basic Framework

We introduce here the basic formalism used throughout the paper, including our abstraction of XML documents, DTDs, and the query language TreeQL.

2.1 Trees

XML documents are abstracted as ordered labeled tree documents [1]. They capture the nesting structure of XML elements and their tags. We refrain from modeling data values since output types only constrain the structure of the output XML document, not the data values at the leaves or in attributes. For example, the tree abstraction of the XML document in Example 1.1 of the introduction (LogiCar's exported data) is shown in Figure 1, with its data values omitted.

We consider ordered trees with node labels from a finite alphabet Σ. We also refer to such trees as Σ-trees. We denote by $\text{nodes}(t)$ the set of *nodes* of a tree t; for a node v, we denote by $\text{lab}(v)$ the *label* of v. There is no a priori bound on the number of children of a node; we therefore call these trees *unranked*. We denote the empty tree by ε and the set of all finite trees over Σ by $\mathcal{T}_\Sigma$. The root of t is denoted by $\text{root}(t)$. To define the semantics of TreeQL programs, we also need the notion of a *forest* which is just a sequence of trees. We employ the following notational convenience. We use $\sigma(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are trees, to denote a tree where the root is labeled with σ and the i-th subtree is t_i.

2.2 Types and DTDs

DTDs and their variants provide a typing mechanism for XML documents. We use several notions of types for trees.

Definition 2.1. Let Σ be an alphabet and $\mathcal{C}$ a class of languages over Σ. A *DTD over* Σ *w.r.t.* $\mathcal{C}$ is a mapping that associates to each symbol σ in Σ a language $d(\sigma)$ in $\mathcal{C}$.

When Σ is understood, we denote the class of DTDs w.r.t. $\mathcal{C}$ by $\text{DTD}(\mathcal{C})$. Let $d \in \text{DTD}(\mathcal{C})$. A Σ-tree t *satisfies* d if for every node v of t with children $v_1, \ldots, v_n$, $\text{lab}(v_1) \cdots \text{lab}(v_n) \in d(\text{lab}(v))$. Note that, if $n = 0$ (so v is a leaf) then ε must belong to $d(\text{lab}(v))$. The set of trees that satisfy d is denoted by $L(d)$.

Obvious examples of classes $\mathcal{C}$ are the regular languages (REG), the star-free regular languages (SF), and the context-free languages (CFL). When $\mathcal{C}$ are the regular languages our notion of DTDs corresponds closely to the DTDs proposed for XML documents.

Example 2.2. Let d be a DTD(REG) specifying the structure of the exported LogiCars documents in Example 1.1, defined as follows:

$$\begin{array}{lcl} d(\text{dealership}) & := & \text{name} \cdot \text{brand}^* \\ d(\text{brand}) & := & \text{name} \cdot \text{car}^* \\ d(\text{car}) & := & \text{name} \cdot \text{price} \cdot \text{quantity} \\ d(\text{name}) & := & \epsilon \\ d(\text{price}) & := & \epsilon \\ d(\text{quantity}) & := & \epsilon \end{array}$$

Obviously, the tree depicted in Figure 1 satisfies d. □

Usual DTDs in XML 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, 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 $w = \sigma_1 \ldots \sigma_n$ 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 $\sigma_i = \sigma$. It is well-known that regular languages are exactly those definable by Monadic Second-Order (MSO) logic[2] on the logical vocabulary of strings [9, 15]. 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*. There is a language-theoretic characterization of star-free languages: they are precisely described by the *star-free regular expressions*, which are build from single symbols, $\emptyset$, and ϵ using concatenation, union, and complement. The correspondence between the logical and language-theoretic definitions of star-free languages was shown by McNaughton and Papert [21, 29]. Note that d in Example 2.2 maps every symbol to a star-free language (the regular expressions used in the example are not star-free, but are obviously equivalent to star-free expressions since the languages they denote are definable in FO).

We will consider an even simpler class of DTDs, which specifies cardinality constraints on the tags of children of a node, but does not restrict their order. Such DTDs are useful when order is irrelevant for the given application. We use a logic called $\mathcal{SL}$, inspired by [25]. The syntax of the language is as follows.

Definition 2.3. For every $\sigma \in \Sigma$ and natural number[3] i, $\sigma^{=i}$ and $\sigma^{\geq i}$ are *atomic $\mathcal{SL}$-formulas*; true is also an atomic $\mathcal{SL}$-formula.

Every atomic $\mathcal{SL}$-formula is an $\mathcal{SL}$-formula and the negation, conjunction, and disjunction of $\mathcal{SL}$-formulas are also $\mathcal{SL}$-formulas.

A string w over Σ satisfies an atomic formula $\sigma^{=i}$ if it has exactly i occurrences of σ, and similarly for $\sigma^{\geq i}$. Further, true is satisfied by every string. Note that the empty string

[2] MSO is first-order logic augmented with quantification over sets.
[3] The natural numbers are $\{0, 1, \ldots\}$.

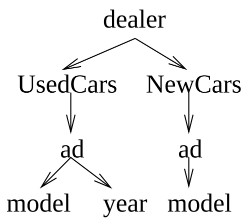


Figure 2: Dealership adds XML document

is defined by $\bigwedge_{\sigma\in\Sigma}\sigma^{=0}$ and the empty set by $\neg$true. Hence, we use ε and $\emptyset$ as shorthand in $\mathcal{SL}$ formulas. Satisfaction of Boolean combinations of atomic formulas is defined in the obvious way.

As an example, consider the $\mathcal{SL}$ 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 $\mathcal{SL}$ 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, $\mathcal{SL}$ forms a natural subclass of the star-free regular expressions.

We have so far defined DTDs and several restrictions. We next consider an orthogonal *extension* of basic DTDs, variants of which are also present in more recent DTD proposals such as XML-Schema [4, 6]. This is motivated by a severe limitation of basic DTDs: their definition of the type of a given tag depends only on the tag itself and not on the context in which it occurs. For example, consider the dealer's document tree in Figure 2.

A DTD corresponding to it might be:

$d(dealer) \;=\; UsedCars \cdot NewCars$
$d(UsedCars) \;=\; ad^*$
$d(NewCars) \;=\; ad^*$
$d(ad) \;=\; model \cdot year \mid model$

However, it may be natural for used car ads to have different structure than new car ads. There is no mechanism to do this using DTDs, since rules depend only on the name of the element, and not on its context. To overcome this limitation, extensions of DTDs provide mechanisms to decouple element names from their types and thus allow context-dependent definitions of their structure. Interestingly, this also leads to closure of the definable sets of trees under Boolean operations. We show one way to formalize the decoupling of names from types, using the notion of *specialized DTD* (studied in [27] and equivalent to formalisms proposed in [5, 10]). Formally, we have:

Definition 2.4. For a class of languages $\mathcal{C}$, a *specialized* DTD over Σ w.r.t. $\mathcal{C}$ is a tuple $\tau = (\Sigma, \Sigma', d, \mu)$ where

(*i*) Σ and Σ' are finite alphabets;

(*ii*) d is a DTD over Σ' w.r.t. $\mathcal{C}$; and

(*iii*) μ is a mapping from Σ' to Σ. A tree t over Σ satisfies a specialized DTD τ, if $t \in \mu(L(d))$. Here, we also denote by μ the homomorphism induced on strings and trees. When Σ is understood, we denote the set of all specialized DTDs w.r.t. $\mathcal{C}$ by S-DTD($\mathcal{C}$).

Intuitively, Σ' provides for some a's in Σ a set of specializations of a, namely those $a' \in \Sigma'$ for which $\mu(a') = a$. Interestingly, it turns out that the class S-DTD(REG) is precisely equivalent to the class of regular tree automata over unranked trees [8, 27]. This is more evidence that specialized DTDs are a robust and natural specification mechanism.

For example, we can now write a specialized DTD distinguishing used car ads from new car ads in the dealer example as follows. $\Sigma = \{dealer, UsedCars, NewCars, ad, model, year\}$, $\Sigma' = \Sigma \cup \{ad^{used}, ad^{new}\}$, μ is the identity on Σ and $\mu(ad^{used}) = \mu(ad^{new}) = ad$ and the mapping d is defined by:

$$
\begin{aligned}
d(dealer) &= UsedCars \cdot NewCars \\
d(UsedCars) &= (ad^{used})^* \\
d(NewCars) &= (ad^{new})^* \\
d(ad^{used}) &= model \cdot year \\
d(ad^{new}) &= model
\end{aligned}
$$

2.3 Logic

Consider some fixed relational vocabulary $\mathcal{S}$. A database over $\mathcal{S}$ is just a finite $\mathcal{S}$-structure defined in the usual way [2, 15]. We denote the domain of a database $\mathcal{A}$ by $\mathrm{dom}(\mathcal{A})$. Furthermore, let $\mathcal{L}$ be a logic over $\mathcal{S}$. We denote the free variables occurring in $\varphi \in \mathcal{L}$ by $\mathrm{Free}(\varphi)$. In the sequel, $\mathcal{L}$ will usually be the set of conjunctive queries over $\mathcal{S}$, denoted by CQ.

Formally, a conjunctive query is a positive existential first-order logic formula $\varphi(x_1, \ldots, x_n)$ having conjunctions as its only Boolean connective. Without loss of generality, we can assume a conjunctive query to be a formula of the form

$$\exists y_1 \cdots \exists y_m \psi(\bar{y}, \bar{x}),$$

where ψ is a conjunction of atomic formulas over $\mathcal{S}$. The basic conjunctive queries do not contain equality.

We relax the definition of CQs by introducing the following notation. We denote by CQ with superscripts in $\{=, \neg\}$ the conjunctive queries where ψ can contain equality and negations of atomic formulas, respectively. A conjunctive query is *projection-free* when there are no leading existential quantifiers.

Another logic frequently referred to in the sequel consists of the FO formulas of the form $\exists \bar{x} \forall \bar{y} \varphi(\bar{x}, \bar{y})$ with φ quantifier-free. We denote this class by $\mathrm{FO}(\exists^*\forall^*)$. This is known as the Bernays-Schönfinkel-Ramsey prefix class (see, e.g., [7]). We denote by $\mathrm{FO}(\forall^*)$ the universal fragment of FO, that is, $\mathrm{FO}(\exists^*\forall^*)$-formulas without existential quantifiers.

Finally, we recall the following technical notion. For a finite sequence of variables X, an *X-substitution θ for database $\mathcal{A}$* is a mapping from the variables in X to $\mathrm{dom}(\mathcal{A})$. Let $\bar{x}$ be a sequence of variables not occurring in X and let $\bar{a}$ be a sequence of as many elements of $\mathrm{dom}(\mathcal{A})$. Then $\theta \cup \{\bar{x} \mapsto \bar{a}\}$ denotes the $(X \cup \{\bar{x}\})$-substitution that maps each x_i to a_i and every $y \in X$ to $\theta(y)$.

2.4 Integrity constraints

In relational databases, one usually considers databases satisfying some integrity constraints [2]. These are sentences in a specific logic. A database $\mathcal{A}$ satisfies a set of constraints Φ, if $\mathcal{A} \models \varphi$ for every $\varphi \in \Phi$. We mainly consider constraints specified in $\mathrm{FO}(\exists^*\forall^*)$. Note

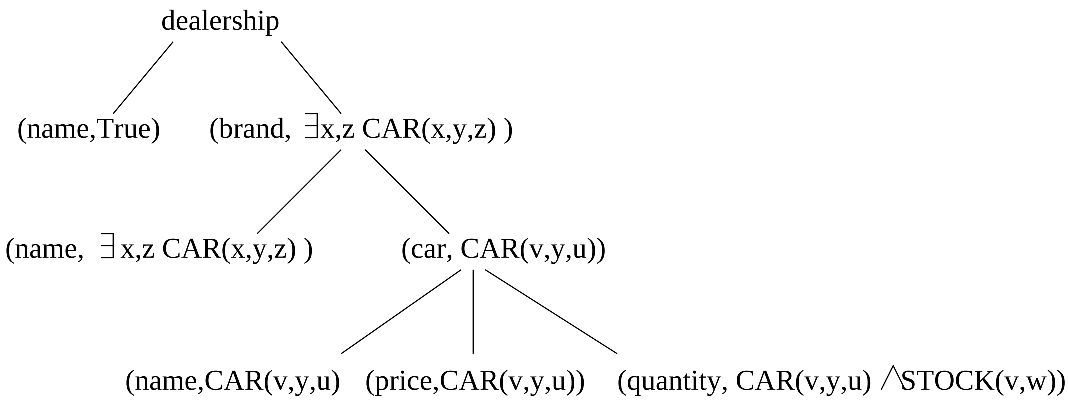


Figure 3: The TreeQL query of Example 1.1

that they encompass functional dependencies (FDs), but not, for instance, inclusion dependencies (IDs). Recall that FDs are expressions of the form $X \to Y$ where X and Y are sets of attributes of a relation, and $X \to Y$ holds in a relation if whenever two tuples agree on X they also agree on Y. IDs are of the form $R[i_1, \ldots, i_k] \subseteq S[j_1, \ldots, j_k]$ where R and S are relation symbols, and $i_1, \ldots, i_k$ and $j_1, \ldots, j_k$ are natural numbers less than or equal to the arity of R and S, respectively. A database satisfies the above inclusion dependency iff $\pi_{i_1,\ldots,i_k}(R) \subseteq \pi_{j_1,\ldots,j_k}(S)$ where π denotes projection as usual. An inclusion dependency is *unary* when $k = 1$. A set Φ of IDs is *cyclic* iff either one of the following holds

- Φ contains a dependency of the form $R[\bar{i}] \subseteq R[\bar{j}]$ with $\bar{i} \neq \bar{j}$; or
- Φ contains dependencies $R_1[\bar{i}_1] \subseteq R_2[\bar{j}_2]$, $R_2[\bar{i}_2] \subseteq R_3[\bar{j}_3]$, $\ldots$, $R_m[\bar{i}_m] \subseteq R_1[\bar{j}_1]$.

A set of IDs is *acyclic* when it is not cyclic. Acyclic IDs arise naturally in practice, for example in relational representations of class hierarchies of object-oriented databases (e.g., see [2]). We denote the class of acyclic inclusion dependencies by AcIDs.

2.5 TreeQL

The transformation language we consider defines mappings of relational databases to trees and is an abstraction of the language used by SilkRoute [16] . We refer to it as TreeQL. The queries are trees whose nodes are labeled with symbol-formula pairs. Denote by $\Sigma \times \mathcal{L}$ the set of pairs $(\sigma, \varphi(\bar{x}))$ with $\sigma \in \Sigma$, and $\varphi(\bar{x})$ a formula in $\mathcal{L}$. TreeQL programs are trees in $\mathcal{T}_{(\Sigma\times\mathcal{L})\cup\Sigma}$. In the following definition we denote by formula(v) the formula associated to a node v.

The TreeQL abstraction of the SilkRoute query in Example 1.1 of the introduction is shown in Figure 3. Note that the TreeQL version of the query does not specify the data values associated to nodes in the answer tree. This can be done in any number of ways, but is left unspecified as it is immaterial to the typechecking problem we are investigating. The formal syntax and semantics of TreeQL are provided below.

Definition 2.5. A *TreeQL*$(\mathcal{L}, \Sigma)$ *program* is a tree $P \in \mathcal{T}_{(\Sigma\times\mathcal{L})\cup\Sigma}$ such that

- the root is labeled with an element from Σ;

- every non-root node is labeled with an element from $\Sigma \times \mathcal{L}$; and,
- $\text{Free}(\text{formula}(v)) \subseteq \text{Free}(\text{formula}(v'))$, for all non-root nodes v and v' where v' is a descendant of v.

The role of the last condition is to enforce grouping of children according to bindings of the variables in their parent node.

If $\mathcal{L}$ or Σ are clear from the context or not important, we sometimes omit them.

Definition 2.6. Let $\mathcal{A}$ be a database over $\mathcal{S}$, P a TreeQL program, and $<$ a total order on $\text{dom}(\mathcal{A})$ and the variables occurring in P.

Let θ and θ' be two X-substitutions where X is the set of variables $\{x_1, \ldots, x_n\}$. Assume $x_i < x_j$ for all $i < j$. We say that $\theta < \theta'$ iff the string $\theta(x_1) \cdots \theta(x_n)$ is smaller than $\theta'(x_1) \cdots \theta'(x_n)$ in the lexicographic order w.r.t. $<$ on $\text{dom}(\mathcal{A})$.

The tree $P(\mathcal{A}, <)$ generated by P from $\mathcal{A}$ and $<$ is now defined as follows.

- The root is $(\text{root}(P), \emptyset)$.
- The non-root nodes consist of pairs of the form (v, θ) where v is a non-root node of P and θ is an X-substitution (where $X = \text{Free}(\text{formula}(v))$) such that $\mathcal{A} \models \varphi[\theta]$ for every formula φ labeling v or labeling an ancestor of v in P.
- The edges in $P(\mathcal{A}, <)$ are $((v, \theta), (v', \theta'))$ such that v' is a child of v in P and θ' is an extension of θ.
- Sibling nodes in $P(\mathcal{A}, <)$ are ordered as follows: if v and v' are siblings in P and v occurs before v', then all nodes (v, θ) occur before all nodes (v', θ') in $P(\mathcal{A}, <)$. Moreover, for given v in P, (v, θ) occurs before (v, θ') in $P(\mathcal{A}, <)$ iff $\theta < \theta'$.
- Finally, the label of a node (v, θ) is the Σ-label of v in P.

Example 2.7. *To illustrate the above definitions, let P be the TreeQL query in Figure 3. Suppose that the database $\mathcal{A}$ consists of the two relations* `CAR` *and* `STOCK` *in the introduction, and let $<$ be the alphabetical order. Then $P(\mathcal{A}, <)$ is the tree depicted in Figure 1. To understand how this tree is constructed, we show (part of) it again in Figure 4, this time showing for each node the variable substitution generating it. As a shorthand we use $\{''\}$ in the figure to denote the fact that the variable substitution of a node is the same as that of its parent. Note that, in this example, the ordering does not affect the output.*

We remark that SilkRoute's mapping language [16], of which TreeQL is an abstraction, also allows to output data values occurring in the input database as labels of leaves in XML documents. However, since our focus is on typechecking and output types do not constrain data values, we choose to omit them from the formalism. □

Example 2.8. As another example, let P' be the TreeQL-program in Figure 5.

Consider further a database $\mathcal{A}$ in which $R = \{(i, j) \mid 0 \leq i \leq j \leq 9\}$, and the natural order $<$ on $\{0, \ldots, 9\}$. Then $P(\mathcal{A}, <)$ is a tree whose root has 10 children labeled b followed by 55 children labeled c and followed by $55^2 = 3025$ children labeled d. □

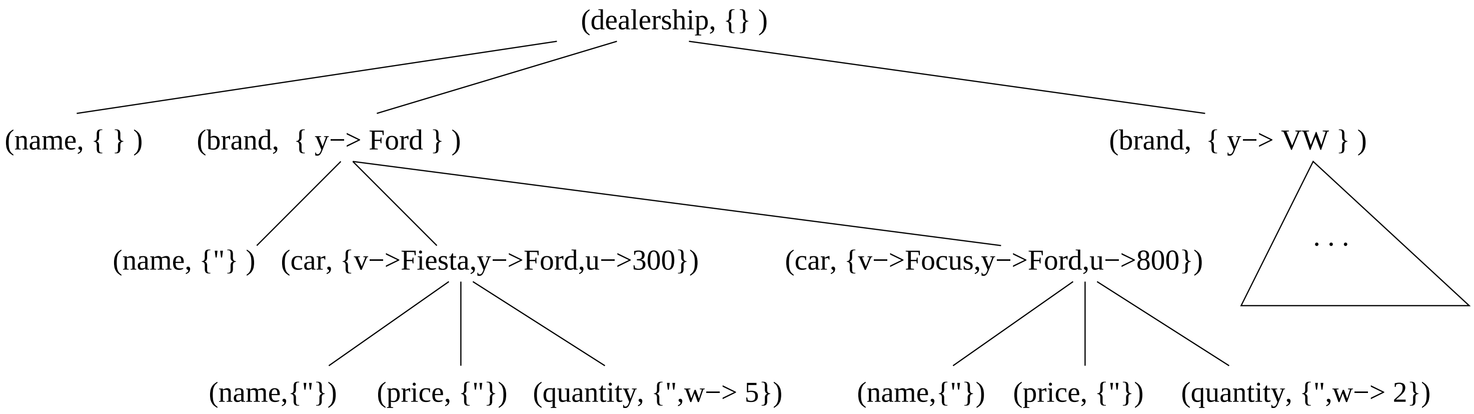


Figure 4: The variables substitutions of the TreeQL query

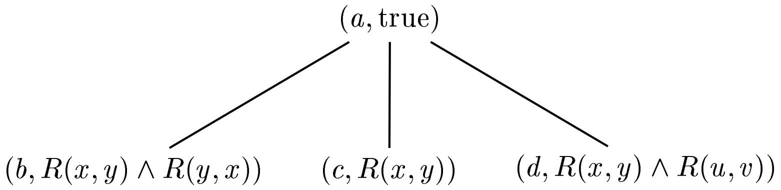


Figure 5: Another TreeQL program example

2.6 An extension: TreeQL with virtual nodes

We will use an extension of TreeQL that allows programs to define "temporary" nodes, that are eliminated in the final answer. We call these *virtual nodes*. While this feature is not supported by Silkroute, it is a useful construct that has been proposed in [5, 23]. To see why virtual nodes are useful, consider again the dealership relational database in Example 1.1 and assume we now wish that the exported XML file obey the following DTD.

$$\begin{array}{lcl} d(\text{dealership}) & := & \text{name} \cdot \text{brand}^* \\ d(\text{brand}) & := & \text{bname} \cdot \ (\text{cname} \cdot \text{price} \cdot \text{quantity})^* \\ d(\text{name}) & := & \epsilon \\ d(\text{price}) & := & \epsilon \\ d(\text{quantity}) & := & \epsilon \end{array}$$

Thus, the exported file must contain, for each brand, a list of triples consisting of car name, price, and quantity. Note the difference with the original DTD: these triples are no lnger nested within `<car>` elements. It is easy to see that this cannot be defined by any TreeQL program, because the program cannot group the car names, prices, and quantities as required. Indeed, the sequence of labels of the children of any node in a tree generated by a TreeQL program always satisfies a pattern of the form $\sigma_1^* \cdots \sigma_n^*$ where every σ_i is a Σ-symbol.

However, suppose we can use temporary nodes, identified by a special label #. Consider the query P_v in Figure 6. The query is similar to that of Figure 3, except that it produces, for each triple of car name, price, and quantity, a parent node labeled by # rather than by `car`.

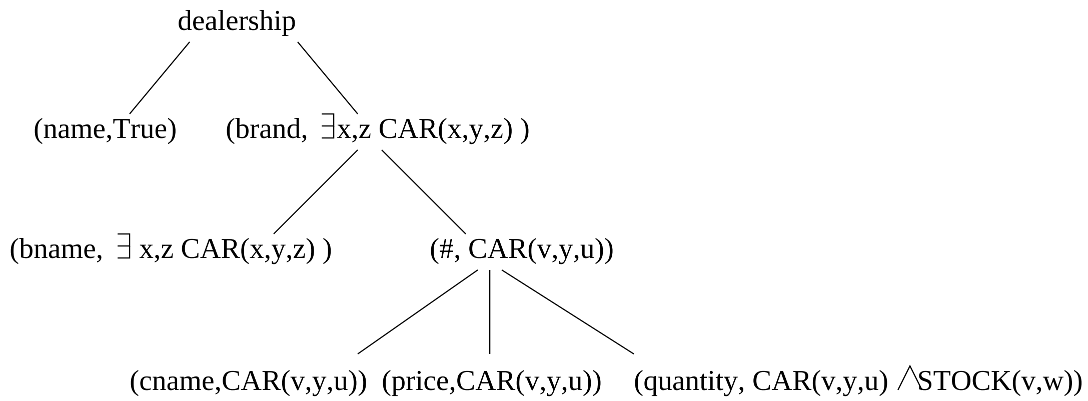


Figure 6: Query with virtual nodes

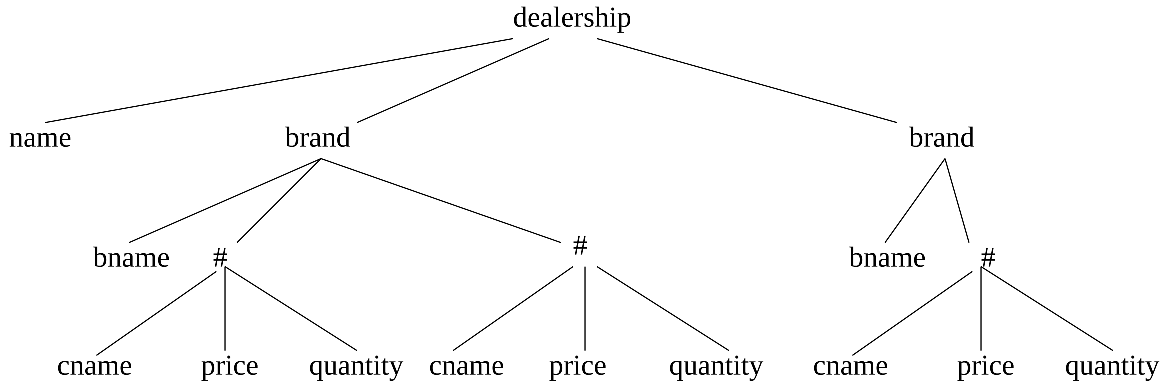


Figure 7: Query result with virtual nodes

The desired ordered sequence of car name, price, and quantity triples can now be obtained by a "flattening" operation that eliminates the # nodes and concatenates their children.

More formally, let $\#$ be a special symbol not occurring in Σ. We denote by $\Sigma_\#$ the set $\Sigma \cup \{\#\}$. The symbol $\#$ will be used to specify virtual nodes. The function $\lambda_\#$ maps trees to forests by recursively eliminating $\#$-labeled nodes as follows. Let t be the tree $\sigma(t_1, \ldots, t_n)$. Then

$$\lambda_\#(t) := \begin{cases} \sigma(\lambda_\#(t_1), \ldots, \lambda_\#(t_n)) & \text{if } \sigma \neq \#; \\ \lambda_\#(t_1), \ldots, \lambda_\#(t_n) & \text{if } \sigma = \#. \end{cases}$$

Definition 2.9. A TreeQL$(\mathcal{L}, \Sigma)$ *program P with virtual nodes* is a TreeQL$(\mathcal{L}, \Sigma_\#)$ program where $\mathrm{lab}(\mathrm{root}(P)) \neq \{\#\}$. We denote the set of all such programs by TreeQL$^{\mathrm{virt}}(\mathcal{L}, \Sigma)$. The tree generated by P from $\mathcal{A}$ and $<$ is defined as $\lambda_\#(P(\mathcal{A}, <))$, and denoted, by slight abuse of notation, also by $P(\mathcal{A}, <)$.

To continue with our running example, for the database $\mathcal{A}$ consists of the relations `CAR` and `STOCK` in the introduction, and for the alphabetical order $<$, $P_v(\mathcal{A}, <)$ is the tree depicted in Figure 7, and $\lambda_\#(P_v(\mathcal{A}, <))$ is the tree in Figure 8.

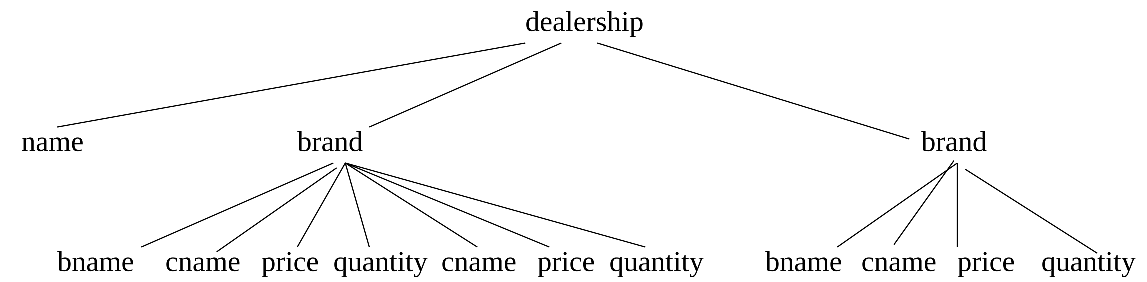


Figure 8: Query result after eliminating the virtual nodes

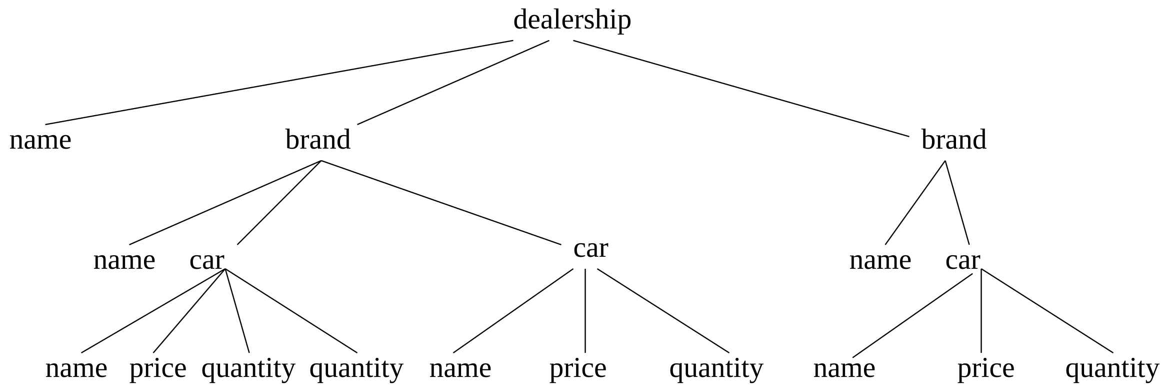


Figure 9: Query result not conforming to the DTD

2.7 Typechecking

We next formalize the central problem of this paper.

Definition 2.10. A TreeQL program P *typechecks* with respect to a set of constraints Φ and an output type d iff $P(\mathcal{A}, <) \subseteq L(d)$ for every database $\mathcal{A}$ that satisfies Φ and every total order $<$ on $\mathrm{dom}(\mathcal{A})$.

Example 2.11. Continuing with Example 2.8, consider the DTD defined by the mapping $d : \{a, b, c, d\} \to \mathrm{REG}$ given by:

$$d(a) := (b^* \cdot (cc)^* \cdot (dd)^*) \mid (b^* \cdot (cc)^* \cdot c \cdot (dd)^* \cdot d)$$

and $d(b) := d(c) := d(d) := \varepsilon$. The type says that the number of c's and the number of d'd under the root are both even or both odd. Then the TreeQL program P' in Example 2.8 typechecks w.r.t. this DTD. □

Example 2.12. Consider the DTD d defined in Example 2.2. The TreeQL-program P in Figure 3 (Example 2.7) does not typecheck w.r.t. d. The reason is that there is no constraint enforcing, in the `STOCK` relation, a single quantity for each car name. Indeed, suppose we add a tuple $(\mathrm{Fiesta}, 6)$ to `STOCK`. Then we obtain a tree of the form depicted in Figure 9, which does not conform to the DTD because the left-most `car` node has two children labeled `quantity`.

However, when we add the constraint

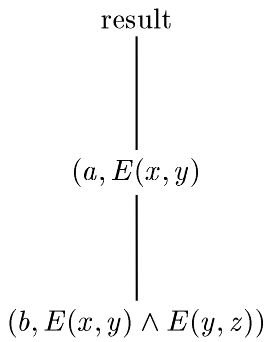
$$\Phi := \{\forall x \forall y \forall y' \, (STOCK(x, y) \wedge STOCK(x, y')) \to (y = y')\},$$

P typechecks w.r.t. d and Φ. $\square$

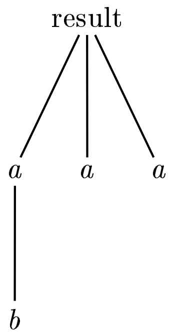
Recall that by our definition, P typechecks w.r.t. d and Φ iff $P(\mathcal{A}, <) \subseteq L(d)$ for every database $\mathcal{A}$ that satisfies Φ and *every total order* $<$ *on* $dom(\mathcal{A})$. The reason for requiring correctness of the answer for *every* ordering of the input is that in practice, some arbitrary ordering of the input is made up in order to translate an unordered relational into an ordered XML document. However, it is of interest to note that for DTDs without specialization, correctness of the answer for every input ordering is implied by correctness of the answer for *some* arbitrary input ordering, as stated next.

Proposition 2.13. *Let d be a DTD (without specialization) and P be an TreeQL program. Suppose there exists a total ordering $<$ of the input $\mathcal{A}$ such that $P(\mathcal{A}, <) \subseteq L(d)$. Then $P(\mathcal{A}, <') \subseteq L(d)$ for every ordering $<'$ of $\mathcal{A}$.*

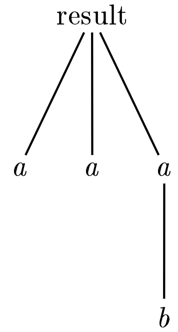
The proof follows immediately from the definition of TreeQL programs and the fact that DTDs without specialization do not enforce constraints across multiple levels in the answer tree. Proposition 2.13 no longer holds for specialized DTDs. Indeed, consider the specialized DTD d defined as $d(\text{result}) := a_{\text{first}} a^*_{\text{rest}}$, $d(a_{\text{first}}) := b$, and $d(a_{\text{rest}}) := \varepsilon$. Define $\mu(\text{result}) = \text{result}$, $\mu(b) = b$, and $\mu(a_{\text{first}}) = \mu(a_{\text{rest}}) = a$. Intuitively, this specialized DTD requires that the first a should have one b-labeled child while all subsequent a's should have no children. Let P be the program



let $\mathcal{A}$ be the structure $E := \{(1,3),(3,5),(2,4)\}$, and let $<_{\text{asc}}$ and $<_{\text{desc}}$ be the ascending and descending ordering of $\{1,2,3,4,5\}$, respectively. Hence, $P(\mathcal{A}, <_{\text{asc}})$ is



while $P(\mathcal{A}, <_{\text{desc}})$ is



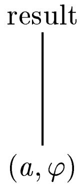
Clearly, the first tree satisfies d while the second does not.

The typechecking problem is parameterized by (1) the fragment of TreeQL; (2) the output type; and (3) the integrity constraints. Therefore, we denote by

$$\mathrm{TC}[\mathcal{R}, \mathcal{D}, \mathcal{IC}],$$

the above decision problem where $\mathcal{R}$ is a fragment of TreeQL or TreeQL$^{\mathrm{virt}}$, $\mathcal{D}$ is a class of output types, and $\mathcal{IC}$ is a class of integrity constraints. To reduce notation, we abbreviate TreeQL($\mathcal{L}$) and TreeQL$^{\mathrm{virt}}(\mathcal{L})$ by $\mathcal{L}$ and $\mathcal{L}_{\mathrm{virt}}$, respectively; and, we abbreviate DTD($\mathcal{C}$) and S-DTD($\mathcal{C}$) by $\mathcal{C}$ and $\mathcal{C}_{\mathrm{spec}}$, respectively.

Remark 2.14. Clearly, $\mathrm{TC}[\mathcal{L}, \mathcal{D}, \mathcal{IC}]$ is undecidable for any logic $\mathcal{L}$ for which satisfiability is undecidable. Indeed, for a sentence $\varphi \in \mathcal{L}$, consider the program



with an output type d that maps $d(\text{result})$ to $\{\varepsilon\}$ (namely the language containing the empty string as a single word) . Then φ is satisfiable iff the program does not typecheck w.r.t. d.

In the sequel we focus on conjunctive queries, which correspond to the widely used select-project-join queries in SQL. As shown in Section 5, the typechecking problem quickly becomes undecidable. Nevertheless, as shown in the next section, we obtain decidability and even tractability for a large class of transformations.

3 Decidability

We present in this section our decidability results on typechecking TreeQL queries:

(i) We show that typechecking is decidable for TreeQL($\mathrm{CQ}^{=,\neg}$) programs, integrity constraints in $\mathrm{FO}(\exists^*\forall^*)$, and star-free output DTDs. The proof yields a CONEXPTIME upper bound. In Section 4, we provide the matching lower bound.

(ii) Typechecking remains decidable for DTDs with full regular expressions when the queries are restricted to projection-free CQs and the integrity constraints to FDs. The proof is based on Ramsey theory and yields a non-elementary upper bound. It is open whether this can be improved.

In Section 5, we show that the above decidability results are essentially tight: slight increase of the power of the DTDs or the integrity constraints leads to undecidability. However, it remains open whether in (*ii*) above, the restriction to projection-free CQs is required.

We begin by introducing some notation needed in the decidability proofs (Theorems 3.2 and 3.6 below). We use in the following $\bar{v}$ to denote a vector of nodes in the query tree, $\bar{x}$ (resp. $\bar{y}$) to denote a vector of formula variables and $\bar{a}$ (resp. $\bar{b}$) to denote a vector of database values.

Definition 3.1. Let R be a TreeQL-program, d a DTD, and Φ a set of constraints such that R does *not* typecheck w.r.t. d and Φ. Consequently,

- there is a path $\bar{v} := v_1, \ldots, v_k$ in R where
  - (*i*) v_1 is a child of the root;
  - (*ii*) $\text{lab}(v_i) = (\sigma_i, \varphi_i(\bar{x}_1, \ldots, \bar{x}_i))$, for $i \in \{1, \ldots, k\}$;
  - (*iii*) let $\bar{x} = \bar{x}_1, \ldots, \bar{x}_k$. v_k has precisely n children with labels $(\delta_1, \psi_1(\bar{x}, \bar{y}_1)), \ldots, (\delta_n, \psi_n(\bar{x}, \bar{y}_n))$ in that order; and
- there is an $\mathcal{A}$ with elements $\bar{a} := \bar{a}_1, \ldots, \bar{a}_k$ such that
  - (*i*) $\mathcal{A} \models \Phi$;
  - (*ii*) $\mathcal{A} \models \varphi_i(\bar{a}_1, \ldots, \bar{a}_i)$ for each $i = 1, \ldots, k$; and
  - (*iii*) $\delta_1^{j_1} \cdots \delta_n^{j_n} \notin d(\sigma_k)$ with $|\{\bar{b} \mid \mathcal{A} \models \psi_i(\bar{a}, \bar{b})\}| = j_i$ for all $i = 1, \ldots, n$.

We say that $(\bar{v}, \mathcal{A}, \bar{a})$ is a *breakpoint* for R, d, and Φ.

We next consider the typechecking problem for TreeQL programs using conjunctive queries with equality and negation, star-free output DTDs, and integrity constraints in FO($\exists^*\forall^*$).

Theorem 3.2. TC[CQ$^{=,\neg}$, SF, FO($\exists^*\forall^*$)] *is in* CONEXPTIME.

Proof. The roadmap of the proof is as follows. The main idea is to show that for every TreeQL program R that does not typecheck with respect to d and Φ there exists some breakpoint of size exponential in $|R| + |d| + |\Phi|$. To this end, we first prove a lemma describing the fine structure of a possible breakpoint. All conditions required for a breakpoint (except the integrity constraints Φ) are existential. Therefore, if some breakpoint involving a structure $\mathcal{A}$ exists, the existential conditions have witnesses drawn from $\mathcal{A}$. We then use the witnesses to construct a smaller breakpoint involving a substructure $\mathcal{B}$ of $\mathcal{A}$, whose size is bounded as desired. Finally, a standard argument shows that Φ holds in $\mathcal{B}$, because of the form of Bernays-Schönfinkel-Ramsey formulas.

Let R be a TreeQL(CQ$^{=,\neg}$) program, let $d \in$ DTD(SF), and let Φ be a finite set of FO($\exists^*\forall^*$) sentences. Suppose that R does not typecheck with respect to d and Φ. Then there exists a breakpoint $(\bar{v}, \mathcal{A}, \bar{a})$ for R, d and Φ. We use below the notation introduced in Definition 3.1.

Let $d(\sigma_k)$ be represented by the star-free regular expression r. So, $\delta_1^{i_1} \cdots \delta_n^{i_n} \notin L(r)$. That is, $\delta_1^{i_1} \cdots \delta_n^{i_n} \in L(\neg r)$. Since $\delta_1^{i_1} \cdots \delta_n^{i_n} \in \delta_1^* \cdots \delta_n^*$, it follows that $\delta_1^{i_1} \cdots \delta_n^{i_n}$ belongs to $\neg r \cap \delta_1^* \cdots \delta_n^*$.

The next lemma characterizes the structure of star-free regular expressions of the above form. We first extend the star-free regular expressions by the constructs $\sigma^{=i}$ and $\sigma^{\geq i}$. These denote the languages $\{\sigma^i\}$ and $\{\sigma^j \mid j \geq i\}$, respectively. As the proof is straightforward but technical we defer it to the appendix. We denote the length of a regular expression s by $|s|$.

Lemma 3.3. *Let r be a star-free regular expression. Then $s := r \cap \sigma_1^* \cdots \sigma_n^*$ is equivalent to a finite disjunction ρ_r of expressions of the form $\sigma_1^{*_1 i_1} \cdots \sigma_n^{*_n i_n}$ where each $*_j \in \{=, \geq\}$ and $i_j \in \mathbb{N}$. Moreover, $i_1, \ldots, i_n \leq |s|$, the size of ρ_r is exponential in $|s|$, and ρ_r can be computed in time exponential in $|s|$.*

By Lemma 3.3, $\neg r \cap \delta_1^* \cdots \delta_n^*$ is equivalent to a disjunction, of exponential size, of expressions of the form $\delta_1^{*_1 j_1} \cdots \delta_n^{*_n j_n}$ where each $*_i \in \{=, \geq\}$ and $j_i \leq |\neg r| + |\delta_1^* \cdots \delta_n^*| \leq |d| + |R|$. Let D be a particular disjunct $\delta_1^{*_1 j_1} \cdots \delta_n^{*_n j_n}$ such that

(1) $\mathcal{A} \models \Phi$ and $\mathcal{A} \models \varphi_i(\bar{a}_1, \ldots, \bar{a}_i)$ for each i; and

(2) $|\{\bar{b} \mid \mathcal{A} \models \psi_i(\bar{a}, \bar{b})\}| *_i j_i$ for $i = 1, \ldots, n$.

We next show there is a substructure $\mathcal{B}$ of $\mathcal{A}$ of size exponential in $|R| + |d| + |\Phi|$, such that $(\bar{v}, \mathcal{B}, \bar{a})$ is a breakpoint for R, d and Φ. In particular, $\mathcal{B}$ satisfies (1) and (2). To see this, we introduce some notation. Let

$$\Phi := \{\ \exists \bar{x}_\ell^\alpha \forall \bar{y}_\ell^\alpha \alpha_\ell(\bar{x}_\ell^\alpha, \bar{y}_\ell^\alpha) \mid 1 \leq l \leq p\},$$

$$\varphi_i(x_1, \ldots, x_i) := \exists \bar{x}_i^\varphi \gamma_i(\bar{x}_1, \ldots, \bar{x}_i, \bar{x}_i^\varphi),$$

for each $i = 1, \ldots, k$, and

$$\psi_i(\bar{x}, \bar{y}_i) := \exists \bar{x}_i^\psi \beta_i(\bar{x}, \bar{y}_i, \bar{x}_i^\psi),$$

for each $i = 1, \ldots, n$.

We next identify a subset E of the domain of $\mathcal{A}$ such that $\mathcal{A}_{|E}$ satisfies (1) and (2). Recall that $\mathcal{A}_{|E}$ is obtained from $\mathcal{A}$ by restricting every relation to its tuples using only elements in E. The set E is the union of three sets E_1, E_2, and E_3, defined next.

For each $\ell \in \{1, \ldots, p\}$, let $\bar{a}_\ell^\alpha$ be a tuple such that $\mathcal{A} \models \forall \bar{y}_\ell^\alpha \alpha_\ell(\bar{a}_\ell^\alpha, \bar{y}_\ell^\alpha)$. As $\mathcal{A}$ satisfies Φ such elements can always be found. Let $E_1 := \{\bar{a}_\ell^\alpha \mid \ell = 1, \ldots, p\}$. Recall that $\bar{a}$ in the breakpoint $(\bar{v}, \mathcal{A}, \bar{a})$ is of the form $\bar{a}_1, \ldots, \bar{a}_n$. For each $i \in \{1, \ldots, k\}$, let $\bar{a}_i^\varphi$ be a tuple such that $\mathcal{A} \models \gamma_i(\bar{a}_1, \ldots, \bar{a}_i, \bar{a}_i^\varphi)$. As $(\bar{v}, \mathcal{A}, \bar{a})$ is a breakpoint, such tuples can always be found. Let $E_2 := \{\bar{a}_i^\varphi \mid i = 1, \ldots, k\}$. Furthermore, for each $i = 1, \ldots, n$, let $\bar{b}_i^1, \ldots, \bar{b}_i^{j_i}$ be j_i tuples $\bar{b}_i$ such that $\mathcal{A} \models \psi_i(\bar{a}, \bar{b}_i^\ell)$ for $\ell = 1, \ldots, j_i$. For each tuple $\bar{b}_i^\ell$, let $\bar{a}_i^{\psi,\ell}$ be a tuple such that $\mathcal{A} \models \beta_i(\bar{a}, \bar{b}_i^\ell, \bar{a}_i^{\psi,\ell})$. Let $E_3 := \{\bar{b}_i^\ell, \bar{a}_i^{\psi,\ell} \mid 1 \leq \ell \leq j_i, 1 \leq i \leq n\}$. Note that the size of $E := E_1 \cup E_2 \cup E_3$ is at most polynomial in $|R| + |d| + |\Phi|$. Hence, the size of $\mathcal{A}_{|E}$ is exponential in $|R| + |d| + |\Phi|$.

By construction, $|\{\bar{b} \mid \mathcal{A}_{|E} \models \psi_i(\bar{a}, \bar{b})\}| *_i j_i$ for $i = 1, \ldots, n$. Indeed, if $|\{\bar{b} \mid \mathcal{A} \models \psi_i(\bar{a}, \bar{b})\}| = j_i$ then also $|\{\bar{b} \mid \mathcal{A}_{|E} \models \psi_i(\bar{a}, \bar{b})\}| = j_i$, and if $|\{\bar{b} \mid \mathcal{A} \models \psi_i(\bar{a}, \bar{b})\}| \geq j_i$ then also $|\{\bar{b} \mid \mathcal{A}_{|E} \models \psi_i(\bar{a}, \bar{b})\}| \geq j_i$.

Moreover, $\mathcal{A}_{|E} \models \Phi$. The latter follows by a standard argument (see, e.g., Proposition 6.2.17 in [7]). Indeed, for each ℓ, $(\mathcal{A}, E_1) \models \forall \bar{y}_\ell^\alpha \alpha_j(\bar{a}_\ell^\alpha, \bar{y}_\ell^\alpha)$, where the elements in E_1 are taken as constants. As these resulting sentences are universal, $(\mathcal{A}_{|E}, E_1) \models \forall \bar{y}_\ell^\alpha \alpha_\ell(\bar{a}_\ell^\alpha, \bar{y}_\ell^\alpha)$ for each ℓ. Hence, $\mathcal{A}_{|E} \models \exists \bar{x}_\ell^\alpha \forall \bar{y}_\ell^\alpha \alpha_\ell(\bar{x}_\ell^\alpha, \bar{y}_\ell^\alpha)$ for each ℓ.

Finally, let $\mathcal{B}$ equal $\mathcal{A}_{|E}$. In view of the above, $(\bar{v}, \mathcal{B}, \bar{a})$ is a breakpoint for R, d, and Φ and the universe of $\mathcal{B}$ is polynomial in $|R| + |d| + |\Phi|$. In summary, whenever a breakpoint

$(\bar{v}, \mathcal{A}, \bar{a})$ exists for R, d, and Φ, there is another breakpoint $(\bar{v}, \mathcal{B}, \bar{a})$ for which the universe of $\mathcal{B}$ is polynomial in $|R| + |d| + |\Phi|$. As the number of tuples in $\mathcal{B}$ can be exponential, this leads to the following NEXPTIME algorithm for computing a counter-example: guess a structure $\mathcal{B}$ together with an ordering $<$ of its domain, and verify that $\mathcal{B}$ satisfies Φ and that $P(\mathcal{B}, <) \notin L(d)$. Note that the verification phase can be done in PSPACE with respect to $\mathcal{B}$, R, d and Φ, so in EXPTIME with respect to $|R| + |d| + |\Phi|$. □

Remark 3.4. (i) In the proof of Theorem 3.2, the structure $\mathcal{B}$ constructed for the counter-example breakpoint $(\bar{v}, \mathcal{B}, \bar{a})$ has universe polynomial in $|R| + |d| + |\Phi|$ and size exponential in the same. This assumes, in particular, that the schema of the relational database is part of the input to the typechecking problem. If, to the contrary, the input schema is considered fixed, then the size of $\mathcal{B}$ remains polynomial in $|R| + |d| + |\Phi|$ and the typechecking problem is in PSPACE with respect to the same. (ii) It is easily seen that the above proof extends to TreeQL programs with FO($\exists^*$)-formulas rather than CQs. The former are FO($\exists^*\forall^*$)-formulas without universal quantifiers. Proposition 5.5 suggests that it is unlikely that typechecking remains decidable for further extensions of the logic used by TreeQL programs.

The next result shows that typechecking remains decidable even when DTDs use full regular languages, as long as the conjunctive queries in the TreeQL program are restricted to be projection-free and the constraints are universal formulas. The proof is non-trivial and is based on Ramsey's Theorem. It is similar and follows from the proof of an analogous but harder result in [3]. We provide here the direct, simpler variant of the proof for the reader's convenience.

We make use of the following extension of Lemma 3.3.

Lemma 3.5. *Let $\delta_1, \ldots, \delta_n$ be symbols and let $\nu = (k_1, j_1), ..., (k_n, j_n)$ be a vector of n pairs of natural numbers. We denote by L_ν the language consisting of all words of the form $\delta_1^{k_1+\alpha_1 \times j_1} \ldots \delta_n^{k_n+\alpha_n \times j_n}$ where each α_i is a natural number, $1 \leq i \leq n$. For each regular language r over alphabet $\{\delta_1, \ldots, \delta_n\}$, there exists a finite set $Vec(r)$ of vectors of pairs of natural numbers as above such that $r \cap \delta_1^* \ldots \delta_n^* = \bigcup_{\nu \in Vec(r)} L_\nu$.*

The proof of the lemma is straightforward and provided in the appendix. We can now show our next decidability result.

Theorem 3.6. TC[projection-free CQ$^{=,\neg}$, REG, FO($\forall^*$)] *is decidable.*

Proof. As in the proof of Theorem 3.2, we show that whenever there exists a breakpoint for an instance of the typechecking problem, there is also one whose size is bounded by some function in the size of the instance. Let $d \in$ DTD(REG) and let R be a projection-free TreeQL(CQ) query. To simplify, we initially assume there are no constraints, that is, $\Phi = \emptyset$. The general case is considered afterwards.

Assume $(\bar{v}, \mathcal{A}, \bar{a})$ is a breakpoint for R and d (with the empty set of constraints), as defined in Definition 3.1. We use the notation introduced there. Let $d(\sigma_k)$ be represented by the regular expression r. So, $\delta_1^{i_1} \cdots \delta_n^{i_n} \notin L(r)$. Thus, $\delta_1^{i_1} \cdots \delta_n^{i_n}$ is in the regular language $\hat{r} = \neg r \cap \delta_1^* \cdots \delta_n^*$. It follows from Lemma 3.5 that $\hat{r}$ is a union of languages, each described by a vector of n pairs of natural numbers $(k_1, j_1), ..., (k_n, j_n)$, restricting the number of the δ_l's in every string to be $k_l + (\alpha_l \times j_l)$ for some natural number α_l.

As $(\bar{v}, \mathcal{A}, \bar{a})$ is a breakpoint, there exists a vector $V = (k_1, j_1), ..., (k_n, j_n)$ such that for all $l = 1, \ldots, n$,

$$|\{\bar{b} \mid \mathcal{A} \models \psi_l(\bar{a}, \bar{b})\}| = k_l + (\alpha_l \times j_l)$$

for some natural number α_l.

We will be interested in substructures of $\mathcal{A}$ that have the same property. More formally,

Definition 3.7. *For a breakpoint $(\bar{v}, \mathcal{A}, \bar{a})$ and a vector V as above, we say that substructure $\mathcal{A}'$ of $\mathcal{A}$ has the the* modulo property w.r.t. $(\bar{v}, \mathcal{A}, \bar{a})$ and V *if for all $l = 1, \ldots, n$,*

$$|\{\bar{b} \mid \mathcal{A}' \models \psi_l(\bar{a}, \bar{b})\}| = k_l + (\alpha'_l \times j_l)$$

for some natural number α'_l.

For brevity, when $(\bar{v}, \mathcal{A}, \bar{a})$ and V are clear from the context we will omit them and simple say that $\mathcal{A}'$ has the the modulo property*.*

We next show that if the size of $\mathcal{A}$ is larger than some fixed number $M(R, d)$, depending on the input, then there is always a strictly smaller substructure with the modulo property. This provides an upper bound on the size of the minimum counter-example. As discussed later, the number $M(R, d)$ is non-elementary with respect to R and d.

Let N be the set of elements in $dom(\mathcal{A})$ consisting of the following:

1. all the elements in the vector $\bar{a}$,
2. for every δ_l with corresponding pair (k_l, j_l), all the domain elements appearing in some arbitrarily chosen sub-relation of size k_l of $\{\bar{b} \mid \mathcal{A} \models \psi_l(\bar{a}, \bar{b})\}$.

It is easy to see that the size of N is bounded by the input independently of $\mathcal{A}$. Indeed, (1) generates at most $|R|$ elements, as the number of variables is bounded by $|R|$; and, (2) generates at most $k_l \times |R|$ elements for each l. We show that if $dom(\mathcal{A}) - N$ is larger than some integer $M(R, d)$ then it contains a non-empty set of elements X, disjoint from N, such that the database $\mathcal{A}'$ obtained from $\mathcal{A}$ by removing all the tuples containing elements in X still has the modulo property. As $dom(\mathcal{A}') = dom(\mathcal{A}) - X$, its size will be smaller than $M(R, d)$.

We use the following notation. To each element $u \in dom(\mathcal{A})$ we associate a vector $t_u = (t_u^1, \ldots t_u^n)$ as follows. For each l $(1 \leq l \leq n)$, let h_u^l be the number of tuples in $\{\bar{b} \mid u \text{ occurs in } \bar{b} \text{ and } \mathcal{A} \models \psi_l(\bar{a}, \bar{b})\}$. Then let t_u^l be h_u^l *mod* j_l if $j_l > 0$, and $t_u^l = 0$ if $j_l = 0$. Similarly, to every set of elements S of size $k \leq |R|$ in $dom(\mathcal{A})$ we associate a vector $t_S = (t_S^1, \ldots t_S^n)$ as follows. Let h_S^l be the number of tuples in $\{\bar{b} \mid u \text{ occurs in } \bar{b} \text{ for some } u \in S \text{ and } \mathcal{A} \models \psi_l(\bar{a}, \bar{b})\}$, and let t_S^l be h_S^l *mod* j_l if $j_l > 0$, and $t_S^l = 0$ if $j_l = 0$.

Note that each vector associated to a set of elements can be viewed abstractly as a color. We can then apply Ramsey's Theorem, (stated below for convenience), and its Corollary 3.9. Indeed, it follows from Corollary 3.9 that for every number m, if $dom(\mathcal{A})$ is larger than some integer $M(m, R, d)$, then there exists a set X of m elements in $\mathcal{A}$, disjoint from N, such that for every $p \leq |R|$, all subsets of X of size p have the same associated vector. There may be different vectors for different p's, but all subsets of the same size p have the same vector.

Now, consider the structure $\mathcal{A}'$ obtained from $\mathcal{A}$ by removing all the tuples containing elements in X. Observe that since the query is projection-free, each element in X deleted from $dom(\mathcal{A})$ affects precisely the tuples in $\{\bar{b} \mid \mathcal{A} \models \psi_l(\bar{a}, \bar{b})\}$ in which it appears. Moreover, since R is projection free and since none of the elements in N are deleted, the following holds:

- $\mathcal{A}' \models \varphi_i(\bar{a}_1, \ldots, \bar{a}_i)$ for each $i = 1, \ldots, k$; and

- for every δ_l for which the corresponding pair is (k_l, j_l) with $j_l = 0$, the set $\{\bar{b} \mid \mathcal{A}' \models \psi_l(\bar{a}, \bar{b})\}$ contains exactly k_l tuples.

To show that $\mathcal{A}'$ has the modulo property, it remains to prove that for every δ_l where $j_l > 0$, the relation $\{\bar{b} \mid \mathcal{A}' \models \psi_l(\bar{a}, \bar{b})\}$ is of size $k_l + \alpha_l \times j_l$ for some natural number α_l. To see this, we compute the number of tuples deleted from $\{\bar{b} \mid \mathcal{A} \models \psi_l(\bar{a}, \bar{b})\}$ as a consequence of removing X from $dom(\mathcal{A})$. If we show that this number is zero modulo j_l, we are done.

The total number of tuples deleted from $\{\bar{b} \mid \mathcal{A} \models \psi_l(\bar{a}, \bar{b})\}$ is described by the following *inclusion-exclusion* formula

$$N_l := \beta_l^1 - \beta_l^2 + \beta_l^3 - \ldots + / - \beta_l^{|\bar{y}_l|}, \qquad (*)$$

where for $i = 1, \ldots, |\bar{y}_l|$,

$$\beta_l^i := \begin{cases} \alpha_l^i \times j_l + \binom{m}{i} \times c_l^i & \text{if } j_l > 0; \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

Here, all α_l^i and c_l^i are positive integers. We explain this formula next. As all elements in X have the same associated vector, which we denote by $(c_1^1, \ldots, c_n^1)$, the total number of destroyed tuples in $\{\bar{b} \mid \mathcal{A} \models \psi_l(\bar{a}, \bar{b})\}$ that contain at least one element in X is of the form $\alpha_l^1 \times j_l + m \times c_l^1$ for some positive integer α_l^1. However, some tuples are counted several times, since they contain several elements of X. To fix this we subtract, for every pair of elements in X, the number of tuples in which the two elements appear together. As every subset of two elements has the same associated vector, denoted by $(c_1^2, \ldots, c_n^2)$, the total number of tuples that contain at least two elements in X is of the form $\alpha_l^2 \times j_l + \binom{m}{2} \times c_l^2$. Note that we again subtracted too much. Indeed, the tuples containing 3 or more elements in X were counted several times. To fix this we add, for each triple of elements, the number of tuples in which the three elements appear together. As above, this is $\alpha_l^3 \times j_l + \binom{m}{3} \times c_l^3$, for some positive integer α_l^3. Since the maximum number of elements in a tuple is bounded by $|\bar{y}_l|$ (which is bounded by $|R|$), the inclusion/exclusion sum stops when we reach that size.

Now, let us choose m to be $\Pi_{l=1}^n j_l \times (|R|!)$. This means that each element in $(*)$ is divisible by j_l. So, the total number N_l of tuples that we lost is divisible by j_l. It follows that $\mathcal{A}'$ still has the modulo property.

To conclude, we review Ramsey's Theorem, and the corollary used in the above proof.

Theorem 3.8. *(Ramsey's Theorem) ([18], see also [28], pp. 7-9) For every three finite natural numbers k, m, w there exists a finite number $M'(k, m, w)$ s.t. for every set Y of elements of size $|Y| \geq M'(k, m, w)$ 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 subsets of X of size k have the same color.*

Furthermore, the number $M'(k, m, w)$ 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 3.9. *For every three finite natural numbers k, m, w there exists a finite number $M(k, m, w)$ s.t. for every set Y of elements of size $|Y| \geq M(k, m, w)$ 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 subsets of X 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 $M(k,m,w) := M'(k,m,w)$ for $k = 1$ and $M(k,m,w) := M'(k, M(k-1,m,w), w)$ for $k > 1$. Note that $M(k-1,m,w)$ exists by the induction hypothesis and $M'(k, M(k-1,m,w), w)$ exists by Ramsey's Theorem. Now, if $|Y| \geq M(k,m,w)$ then, by the definition of M, for every coloring of subsets of Y there is some $X' \subset Y$ of size $|X'| = M(k-1,m,w)$ where all the subsets of size k have the same color. Also, 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. □

Now, observe that in the proof above we simply need to view each possible vector attached to a set domain elements as a color. The number w of available colors is then $j_1 \times \ldots \times j_n$ (node that these numbers depend on R and d); $k = |R|$; and, $m = \Pi_{l=1}^{n} j_l \times (|R|!)$. The rest follows immediately from Corollary 3.9.

Recall that so far we assumed no constraints. However, as $\mathcal{A}'$ is a substructure of $\mathcal{A}$ it readily follows that every universal formula satisfied by $\mathcal{A}$ is also satisfied by $\mathcal{A}'$. Hence, the proof also works for universal constraints. □

The proof yields a non-elementary upper bound. It is open whether this can be improved. It also remains open whether the projection-free restriction can be removed or whether the class of constraints can be extended.

4 Complexity

In this section we provide several lower bounds for the complexity of typechecking. We also exhibit a practically significant restriction for which typechecking is in PTIME.

Theorem 3.2 provides an upper bound of CONEXPTIME on the complexity of typechecking under certain restrictions. We next prove a matching lower bound for the case when negation and inequality are allowed in CQs. However, we show that even without these, typechecking remains intractable, more precisely DP-hard.[4] By further restricting the structure of CQs and $\mathcal{SL}$-formulas we obtain a PTIME algorithm for typechecking. To this end define $\mathcal{SL}^r$ as the fragment of $\mathcal{SL}$ where there are no occurrences of the form $\sigma^{=i}$ and all occurrences of the form $\sigma^{\geq i}$ are such that $i \in \{0, 1\}$. We abbreviate $\sigma^{\geq 1}$ simply by σ. This fragment already suffices to obtain the next lower bound.

Theorem 4.1. TC[CQ$^{\neg}$, $\mathcal{SL}^r$, $\emptyset$] *is hard for* CONEXPTIME.

Proof. The proof consists of a reduction from the satisfiability problem of FO($\exists^*\forall^*$) sentences without equality which is known to be hard for NEXPTIME (see, e.g., Theorem 6.2.21 in [7]), to the complement of the typechecking problem.

Let φ be a formula of the form $\exists x_1, \ldots, x_n \forall y_1, \ldots, y_m \psi(\bar{x}, \bar{y})$ over the relations $R_1, \ldots, R_k$ without equality. The input database for the TreeQL program consists of the relations D_1, ..., D_n, R_1, ..., R_k. Elements from the sets D_1, ..., D_n serve as possible interpretations for the variables x_1, ... ,x_n.

We have to check whether there is a database $\mathcal{A}$ with a tuple $\bar{d}$ such that $\mathcal{A} \models \forall \bar{y}\, \psi(\bar{d}, \bar{y})$. We test the converse, that is $\mathcal{A} \not\models \forall \bar{y} \psi(\bar{d}, \bar{y})$ or equivalently $\mathcal{A} \models \exists \bar{y} \neg \psi(\bar{d}, \bar{y})$. Assume that

[4]Recall that DP properties are of the form $L_1 \wedge L_2$ where $L_1 \in$ NP and $L_2 \in$ CO-NP (see, e.g., [26]).

$\neg\psi$ is of the form $\bigvee_{j=1}^{k} L_j(\bar{x},\bar{y})$ where each $L_j(\bar{x},\bar{y})$ is a conjunction $\bigwedge C$ of atomic formulas and negations thereof. Thus, each L_j is a projection-free query in $\mathrm{CQ}^{\neg}$. We define a TreeQL program as follows: the root is labeled with 'result' and has exactly one child labeled with

$$(D, \bigwedge_{i=1}^{n} D_i(x_i))$$

giving the required interpretation to the x_i's. Further, D has as children $(@_j, L_j(\bar{x},\bar{y}))$ for each $j = 1, \ldots, k$ (recall that L_j is a projection-free query in $\mathrm{CQ}^{\neg}$). The output DTD d is of the following form: $d(\text{result}) := \text{true}$ and

$$d(D) := \bigvee_{j=1}^{k} @_j.$$

Suppose the TreeQL program R does not typecheck. Hence, there is an $\mathcal{A}$ and an ordering $<$ such that $R(\mathcal{A},<)$ does not satisfy d. As a violation of d can only happen at D-nodes there is at least one D-labeled node in $R(\mathcal{A},<)$ without children, that is, for which none of the $@_j$'s appear. Let D give the interpretations $\bar{d} = d_1, \ldots, d_n$ to the variables $x_1, \ldots, x_n$. As none of the $@_j$'s appear, $\mathcal{A} \not\models \exists\bar{y}\neg\psi(\bar{d},\bar{y})$. That is, $\mathcal{A} \models \forall\bar{y}\psi(\bar{d},\bar{y})$. Hence, $\mathcal{A} \models \exists\bar{x}\forall\bar{y}\psi(\bar{x},\bar{y})$ and φ is satisfiable. Conversely, if $\mathcal{A}$ is a model of φ and we instantiate $D_1, \ldots, D_n$ with the witnesses for the existential quantifiers then $R(\mathcal{A} \cup \{D_1, \ldots, D_n\}, <) \notin L(d)$ for any $<$. $\square$

Remark 4.2. In the above proof, it is possible to eliminate the sets D_i at the expense of introducing equality. Indeed, we could replace the node $(D, \bigwedge_{i=1}^{n} D_i(x_i))$ by $(D, \bigwedge_{i=1}^{n} x_i = x_i)$. $\square$

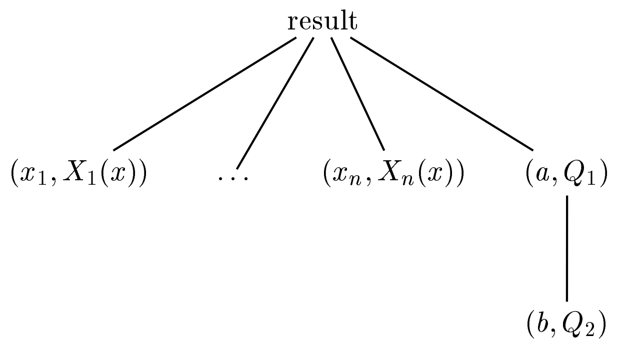
Although it is unclear whether in Theorem 4.1, negation can be dispensed with, we show that in any case the complexity of the problem, even for the standard case, remains intractable. Indeed, one can easily reduce the containment of conjunctive queries and propositional validity to typechecking. In the sequel, $\mathrm{CQ}^{\neq}$ denotes CQ with inequality. The proof of the next proposition is straightforward.

Proposition 4.3. *1.* $\mathrm{TC}[\mathrm{CQ}, \mathcal{SL}^r, \emptyset]$ *is* DP*-hard.*

2. $\mathrm{TC}[\mathrm{CQ}^{\neq}, \mathcal{SL}^r, \emptyset]$ *is* Π_2^p*-hard.*

Proof. (1) Let φ be a formula in propositional logic over the variables $x_1, \ldots, x_n$. Let Q_1 and Q_2 be two conjunctive queries. It is well-known that testing whether φ is valid, and testing whether $Q_1 \subseteq Q_2$, are hard for CONP and NP, respectively (see, e.g., [26, 2]).

Let R be the TreeQL program defined as follows



Define the DTD d by $d(\text{result}) := \varphi$ (recall that x_i and $\neg x_i$ are abbreviations of $x_i^{\geq 1}$ and $x_i^{=0}$, respectively) and $d(a) = b^{\geq 1}$. Note that the set X_i assigns a truth value to the variable x_i. More precisely, x_i is true iff X_i is non-empty. Clearly, R typechecks w.r.t. d iff φ is valid *and* $Q_1 \subseteq Q_2$. Indeed, suppose R typechecks w.r.t. d. Then for every input database $\mathcal{A}$ over $X_1, \ldots, X_n$ and the relations occurring in Q_1 and Q_2, and for every ordering $<$, $R(\mathcal{A}, <)$ satisfies d. By construction of d, this implies that every truth assignment to the variables x_i satisfies φ. Moreover, every a appearing as a child of result also has a b child. Hence, Q_1 is contained in Q_2. Conversely, it is easy to see that R typechecks w.r.t. d if φ is valid *and* $Q_1 \subseteq Q_2$.

(2) The proof is by reduction of containment of conjunctive queries with inequalities, which is known to be Π_2^p-complete [30]. The reduction is similar to that of (1). Indeed, let Q_1 and Q_2 be two conjunctive queries with inequalities. Define R as the program

$$\begin{array}{c} \text{result} \\ | \\ (a, Q_1) \\ | \\ (b, Q_2) \end{array}$$

and the DTD d by $d(\text{result}) := \text{true}$ and $d(a) = b^{\geq 1}$. Again, R typechecks w.r.t. d iff $Q_1 \subseteq Q_2$.

□

The proof of Proposition 4.3 implies that, in order to have a PTIME algorithm for typechecking, we must at least restrict the queries so that testing containment is in PTIME and that validity of the $\mathcal{SL}^r$ formulas used must be in PTIME. We present one set of restrictions that leads to a PTIME typechecking test. Let CQ^k denote the conjunctive queries in FO^k, i.e. the set of conjunctive queries using at most k variables. Such queries can be evaluated in combined complexity PTIME [19, 33]. We restrict TreeQL programs as follows: there exists some k such that, for each node v in the program, the conjunction of all queries of nodes along the path from root to v is in CQ^k. Furthermore, no distinct siblings v, v' in the query tree have labels (a, φ) and (a, φ') for the same $a \in \Sigma$. We call such a program *k-bounded* and denote the set of k-bounded TreeQL programs by TreeQL^k. Finally, we also need a restriction on the $\mathcal{SL}^r$ formulas used in the DTD: they are in conjunctive normal form. We call such $\mathcal{SL}^r$ formulas *conjunctive*.

The fragment CQ^k appears to be a practically useful one. Note that the only semantic restriction in the definition is that siblings cannot have the same label. However, this appears to be frequently satisfied in natural queries. For instance, the query of Figure 5, as well as the query of Example 1.1, shown in Figure 3, satisfy this condition. The number k is a parameter that can be computed for each query. For example, the query of Figure 5 belongs to CQ^4. The query of Figure 3 belongs to CQ^8 (the bound variables occurring in several formulas have to be renamed to satisfy the definition, which explains that k is larger than 6, the number of variables used in Figure 3).

Theorem 4.4. TC[CQ^k, conjunctive $\mathcal{SL}^r$, $\emptyset$] *is in* PTIME *for TreeQL*^{k} *programs.*

Proof. Let R be a TreeQL^k program and let d be a DTD using conjunctive $\mathcal{SL}^r$ formulas. We assume w.l.o.g. that no bound variable occurs in two distinct formulas of R. For every non-leaf node v of R we do the following. Let $d(\text{lab}(v)) = \varphi_v$, where $\varphi_v = \wedge_i C_i$ and each C_i is a disjunction of positive or negated a_i's. Further, let γ be the conjunction of the formulas occurring in labels along the path from $root$ to v. The program typechecks w.r.t. v if for every input, the sequence of children of v in the output satisfies each of the C_i's. So it is enough to typecheck separately with respect to each of the C_i's. Each C_i is of the form $a_1 \vee \ldots \vee a_n \vee \neg b_1 \vee \ldots \neg b_m$. For each $a \in \Sigma$, let ψ_a denote the formula associated to the unique child of v labeled with a. There are three cases to consider:

1. $n > 0$ and $m > 0$. Then C_i is $(b_1 \wedge \ldots \wedge b_m) \;\rightarrow\; (a_1 \vee \ldots \vee a_n)$. We must check that

$$\exists(\psi_{b_1} \wedge \ldots \wedge \psi_{b_m} \wedge \gamma) \rightarrow \; \exists((\psi_{a_1} \wedge \gamma) \vee \ldots \vee (\psi_{a_n} \wedge \gamma))$$

   where the $\exists$ quantify all variables on the left and right-hand sides, respectively, except the free variables occurring in γ. From standard conjunctive query techniques (e.g., see [2]) it follows that the above holds iff there exists j such that

$$\exists(\psi_{b_1} \wedge \ldots \psi_{b_m} \wedge \gamma) \rightarrow \exists(\psi_{a_j} \wedge \gamma).$$

   This in turn holds iff the result of evaluating the conjunctive query $\exists(\psi_{a_j} \wedge \gamma)$ on the canonical structure associated to the matrix[5] of $\exists(\psi_{b_1} \wedge \ldots \wedge \psi_{b_m} \wedge \gamma)$ includes the tuple of free variables of γ. Since $\exists(\psi_{a_j} \wedge \gamma)$ is in CQ^k, this can be checked in PTIME.

2. $m = 0$. This amounts to testing that $\exists((\psi_{a_1} \wedge \gamma) \vee \ldots \vee (\psi_{a_n} \wedge \gamma))$ is true on every input. This is false on the empty input, so the program does not typecheck.

3. $n = 0$. Since $\exists(\psi_{b_1} \wedge \ldots \wedge \psi_{b_m} \wedge \gamma)$ is always satisfiable, this never typechecks.

Clearly, each of the steps outlined above is polynomial with respect to R and d. The fine-grained complexity analysis depends of course on the specific data structures used and the assumptions on the set of operations that can be performed in constant time. Roughly speaking, the straightforward algorithm implementing the above takes time $O(|d|^2|R|^{k+3})$. This breaks down as follows:

- the iteration through the non-leaf nodes v of R takes time $O(|R|)$;
- the iteration through the conjuncts C_i of φ_v is $O(|d|)$;
- for each disjunct, steps (2) and (3) can be assumed to be $O(1)$, as they can be merged with the above;
- for step (1), the iteration through all disjuncts a_i takes time $O(|d|)$;
- testing each implication

$$\exists(\psi_{b_1} \wedge \ldots \psi_{b_m} \wedge \gamma) \rightarrow \exists(\psi_{a_j} \wedge \gamma)$$

[5]The canonical structure is the Herbrandt interpretation consisting of all atoms occurring in the matrix.

by evaluating the query $\exists(\psi_{a_j} \wedge \gamma)$ on the canonical structure associated to the matrix of $\exists(\psi_{b_1} \wedge \ldots \wedge \psi_{b_m} \wedge \gamma)$ takes time $O(|R|^k|R|^2)$. The $|R|^k$ factor is due to iterating through the possible assignments to the k variables of $\exists(\psi_{a_j} \wedge \gamma)$, and the $|R|^2$ factor is due to checking that every ground atom in the matrix of $\exists(\psi_{a_j} \wedge \gamma)$ for a given variable assignment is also a ground atom in the matrix of $\exists(\psi_{b_1} \wedge \ldots \wedge \psi_{b_m} \wedge \gamma)$ for the same variable assignment.

□

Remark 4.5. The typechecking algorithm for TreeQL^k programs outlined in the proof of Theorem 4.4 is polynomial for fixed k, with k in the exponent. This worst-case bound suggests that typechecking could become intractable for all but small values of k. However, this analysis is quite conservative. In practice, typechecking is likely to remain feasible even for fairly large values of k. Indeed, item (3) in the above proof, which is responsible for the presence of k in the exponent, can be implemented simply by evaluating a straightforward SQL query corresponding to $\exists(\psi_{a_j} \wedge \gamma)$ on the database corresponding to the matrix of $\exists(\psi_{b_1} \wedge \ldots \wedge \psi_{b_m} \wedge \gamma)$. The number of variables used in the SQL query (and thus the number of joins in the resulting relational algebra expression implementing the query), is bounded by k. SQL queries with large number of variables are routinely run in practice, and are feasible due to relational query optimization techniques. Lastly, note that the database corresponding to the matrix of $\exists(\psi_{b_1} \wedge \ldots \wedge \psi_{b_m} \wedge \gamma)$ is likely to be very small compared to typical databases on which SQL queries are run.

5 Undecidability Results

We have seen in the previous section that TC[CQ$^{\neg,=}$, SF, FO($\exists^*\forall^*$)] is decidable. This is a fairly tight bound. Indeed, we next show that even minor extensions lead to undecidability. We consider several extensions of the output DTDs, TreeQL queries, and integrity constraints. Specifically, we consider (*i*) specialization, (*ii*) virtual nodes, (*iii*) FO($\exists^*\forall^*$) formulas, and (*iv*) acyclic inclusion dependencies (AcID), and show that typechecking becomes undecidable with each of these extensions. Another parameter in the formalism is the class of string languages used by DTDs. Recall that decidability still holds if we replace SF by REG when restricting to projection-free CQs and omit integrity constraints. We show that this most likely cannot be extended beyond REG: allowing *deterministic* CFLs (DCFL) in DTDs leads to undecidability.

We first consider the impact of augmenting DTDs with specialization. We illustrate the reduction in Example 5.2.

Theorem 5.1. TC[projection-free CQ, $\mathcal{SL}^r_{\text{spec}}$, $\emptyset$] *is undecidable.*

Proof. We use a reduction from finite validity of first-order logic formulas without equality over directed graphs, which is well known to be undecidable (see, e.g., [7]). The finite validity problem is to check, given an FO formula φ, whether $\mathcal{A} \models \varphi$ for every finite graph $\mathcal{A}$ with non-empty universe. Assume $\varphi = \Theta_1 x_1 \ldots \Theta_n x_n \delta(x_1, \ldots, x_n)$, where each Θ_i is $\exists$ or $\forall$ and δ is quantifier-free, and the vocabulary of φ is a binary relation E providing the edges of the graph. To avoid having equalities in our CQ's we introduce, in addition to the graph relation E, a unary relation D that will contain the universe of the graph. Next, we modify

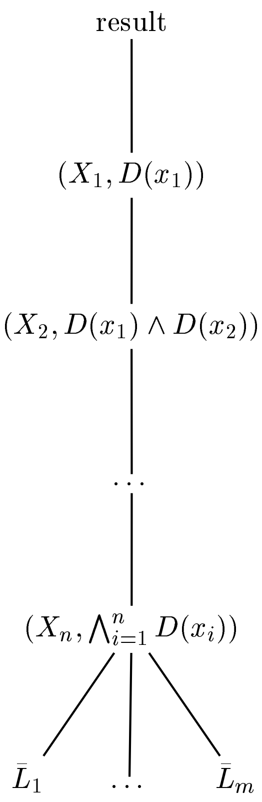


Figure 10: The TreeQL program R of Theorem 5.1.

φ to quantify explicitly over D. Therefore, denote by φ' the formula obtained from φ by recursively replacing each occurrence of a subformula of the form $\exists x_i\, \psi$ by $\exists x_i\, (D(x_i) \wedge \psi)$, and each occurrence of a subformula of the form $\forall x_i\, \psi$ by $\forall x_i\, (D(x_i) \rightarrow \psi)$. Hence, φ' is an FO formula over the vocabulary $\{E, D\}$ where D is a unary relation and E a binary edge relation. Clearly, φ is valid iff φ' is valid. This modification allows us to avoid using the equality $x = x$ in our TreeQL program to define the domain, and instead to simply state $D(x)$. We can assume that δ is in disjunctive normal form, that is, of the form $\bigvee_{i=1}^{m} L_i$, where each L_i is of the form $P^i \wedge \bigwedge_{j=1}^{m_i} N_j^i$ where P^i is a conjunction of atomic formulas and each N_j^i is the negation of a single atomic formula. For a negated atomic formula N we denote the unnegated formula by $\tilde{N}$. Recall that atomic formulas can only be of the form $E(x_i, x_j)$.

Consider the TreeQL(CQ) program R depicted in Figure 10. We denote by $\bar{L}_i$ the sequence of $m_i + 1$ nodes labeled as follows

$$(P^i, P^i \wedge \bigwedge_{i=1}^{n} D(x_i))(N_1^i, \tilde{N}_1^i \wedge \bigwedge_{i=1}^{n} D(x_i)) \dots (N_{m_i}^i, \tilde{N}_{m_i}^i \wedge \bigwedge_{i=1}^{n} D(x_i)).$$

Recall that the first component of each pair is a label while the second one is a formula. Intuitively, every occurrence of an X_i in the output tree represents a value assignment for the variable x_i. More specifically, a node (v, θ) in $R(\mathcal{A}, <)$ labeled X_i specifies in θ a value assignment for the variables $x_1, \dots, x_i$, $1 \leq i \leq n$.[6] The specialized DTD

[6] Recall the definition of nodes in the output tree $R(\mathcal{A}, <)$.

then checks the quantification pattern of φ. For instance, if the prenex of φ looks like $\exists x_1 \forall x_2 \exists x_3$, then the specialized DTD should verify that there is an X_1-node such that for all its X_2-children there is an X_3-node that satisfies δ. We make use of the alphabet $\Sigma' = \{Y_i, Z_i \mid i \in \{1, \ldots, n\}\} \cup \{\text{result}\}$. If the specialized DTD assigns the label Y_n to a node (v, θ) in $R(\mathcal{A}, <)$ with $\text{lab}(v) = X_n$ then $\mathcal{A} \models \delta(\theta)$. Further, for each $i = 2, \ldots, n-1$, if the specialized DTD assigns the label Y_i to a node (v, θ) in $R(\mathcal{A}, <)$ with $\text{lab}(v) = X_i$ then $\mathcal{A} \models \Theta_{i+1} x_{i+1} \cdots \Theta_n x_n\, \delta(\theta)$. Consequently, if $\Theta_1 = \exists$ then at least one child of the root (labeled result) should be a Y_1; otherwise, if $\Theta_1 = \forall$ then all children of the root should be Y_1's. So Y's stand for marked values assignments. Similarly, Z's will stand for unmarked ones.

We have the following specialized DTD. Define

$$d(\text{result}) := \begin{cases} Y_1 \vee \varepsilon & \text{if } \Theta_1 = \exists; \text{ and} \\ (Y_1 \wedge \neg Z_1) \vee \varepsilon & \text{if } \Theta_1 = \forall. \end{cases}$$

For $i := 1, \ldots, n-1$,

$$d(Y_i) := \begin{cases} Y_{i+1} & \text{if } \Theta_i = \exists; \\ Y_{i+1} \wedge \neg Z_{i+1} & \text{if } \Theta_i = \forall; \end{cases}$$

$$d(Z_i) := \text{true};$$

$$d(Y_n) := \bigvee_{i=1}^{m} (P^i \wedge \bigwedge_{j=1}^{m_i} \neg N_j^i); \text{ and}$$

$$d(Z_n) := \text{true}.$$

Here, ε makes sure the empty graph typechecks. Note that the expression $Y_i \wedge \neg Z_i$ means that there is at least one child and all children should be labeled Y_i. Finally, set for each i, $\mu(Z_i) := X_i$ and $\mu(Y_i) := X_i$.

Clearly, $R(\mathcal{A}, <) \in L(d)$ iff $\mathcal{A} \models \varphi'$ or $\mathcal{A}$ is the empty structure. Hence, φ' (and φ) is valid iff R typechecks w.r.t. d. □

Example 5.2. We illustrate the reduction in the proof of Theorem 5.1 by an example. Let $\varphi := \forall x_1 \exists x_2 E(x_1, x_2)$. We construct φ' by restricting quantification over D. So, φ' is of the form $\forall x_1 (D(x_1) \to \exists x_2 (D(x_2) \wedge E(x_1, x_2)))$. The latter formula is equivalent to $\forall x_1 \exists x_2 (\neg D(x_1) \vee (D(x_2) \wedge E(x_1, x_2)))$ which is in disjunctive normal form. The TreeQL program R is depicted in Figure 11. For clarity, we enclosed formulas occurring as labels with brackets. The DTD d is defined as follows: $d(\text{result}) := (Y_1 \wedge \neg Z_1) \vee \varepsilon$; $d(Y_1) := Y_2$; $d(Y_2) := \neg[\neg D(x_1)] \vee [D(x_2) \wedge E(x_1, x_2)]$; $d(Z_1) := \text{true}$; and $d(Z_2) := \text{true}$. □

The next result shows that typechecking becomes undecidable when queries can use virtual nodes.

Theorem 5.3. TC[projection-free CQ_{virt}, SF, $\emptyset$] *is undecidable.*

Proof. As in the proof of Theorem 5.1, we reduce the problem of validity of FO sentences to the typechecking problem. Thus, let φ be an FO formula over graphs. We construct a TreeQL program R with virtual nodes and using only projection-free CQ formulas, and a star-free DTD d such that $\mathcal{A} \models \varphi$ iff $R(\mathcal{A}, <)$ satisfies d, for every non-empty graph $\mathcal{A}$.

We use the same notation as in the proof of Theorem 5.1. Thus, let φ' be constructed from φ in the same manner, and let L_i and $\bar{L}_i$ be defined in the same way. The idea

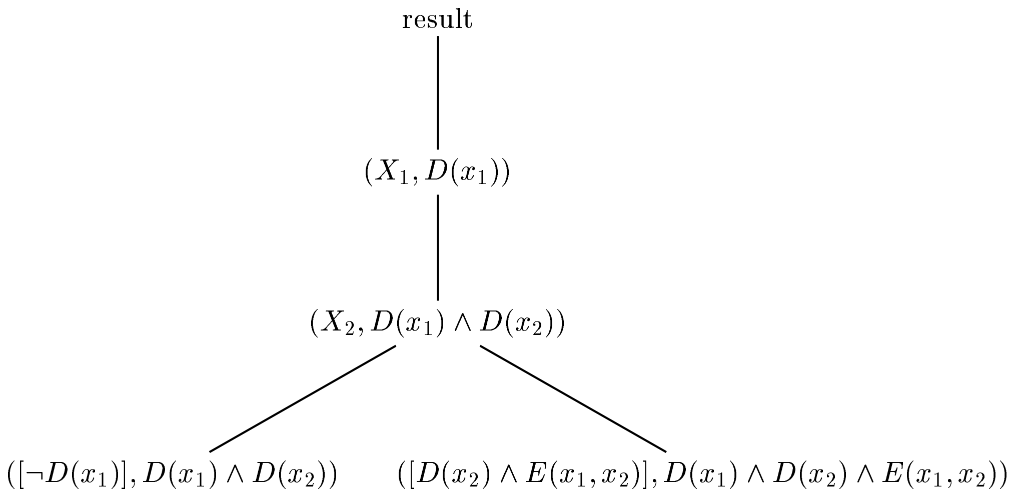


Figure 11: The TreeQL program R of Example 5.2.

of the reduction is similar. In the proof of Theorem 5.1, the answer to R consists of a tree whose internal nodes result from instantiating the variables $x_1, \ldots, x_n$ of the formula φ', and whose leaves provide the ground matrix of φ' for each complete instantiation of the variables. Then specialization is used to check the formula, following its quantification pattern. This time, however, we cannot use specialization. To overcome this, we use virtual nodes to obtain as answer to the query a string that essentially encodes the tree constructed in the previous proof. In particular, we use markers X_i and $\bar{X}_i$ to make explicit the scope of the quantification of x_i in the formula, and the scope of each instantiation. Then the quantification pattern of the formula can be checked using a star-free regular expression applied to the string.

More precisely, let R be the TreeQL$^{\text{virt}}$ program depicted in Figure 12. Intuitively, R works as follows. Due to the virtual nodes, the answer to R on input $(\mathcal{A}, <)$ is a flat tree consisting of a string w under the root. Suppose the universe of $\mathcal{A}$ (ordered by $<$) is $a_1, \ldots, a_k$. The string w is a concatenation of strings $w_1 \ldots w_k$ where each w_j is of the form $X_1 \alpha \bar{X}_1$ and corresponds to all valuations θ of for which $\theta(x_1) = a_j$. Each α is in turn a concatenation of words of the form $X_2 \beta \bar{X}_2$, one for each valuation of x_2, and this is repeated recursively. The innermost strings that are generated correspond to complete valuations θ of $x_1, \ldots x_n$, and consist of $\theta(\bar{L}_1) \ldots \theta(\bar{L}_k)$ where each $\theta(\bar{L}_i)$ is the concatenation of the labels in $\bar{L}_i$ for which the corresponding formula is true for θ.

To verify satisfaction of φ', it is enough to verify satisfaction of the matrix for its quantification pattern. For example, if x_1 is existential in φ', we need to find at least one minimal substring $X_1 \alpha \bar{X}_1$ of w (α does not contain X_1 or $\bar{X}_1$), corresponding to an instantiation of x_1 for which the remainder of the formula holds. If x_1 is universal, this has to be verified for every such minimal substring $X_1 \alpha \bar{X}_1$. It turns out that the above can be specified using a star-free regular expression. However, instead of devising directly the star-free expression, it is more convenient to construct an FO formula over strings that defines the desired property. By a Theorem of McNaughton and Papert [21, 29, 15], each FO formula on strings is equivalent to a star-free regular expression.

We next describe the construction of the FO formula over strings. The formula mimics closely φ', and in fact can be constructed directly from φ' by appropriately modifying it.

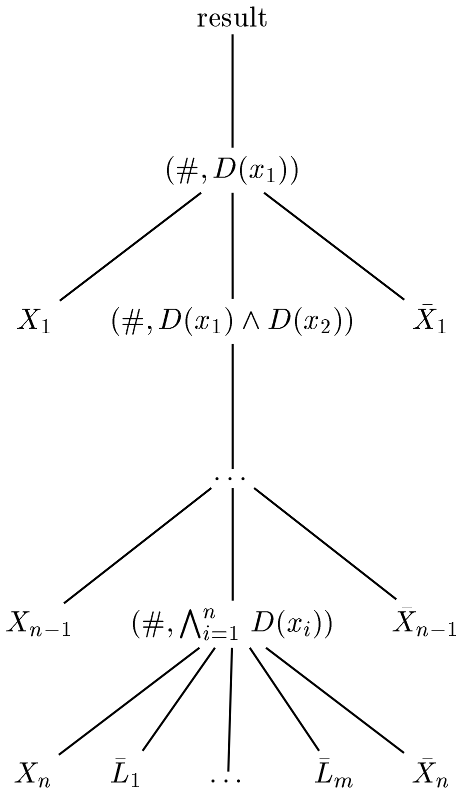


Figure 12: The TreeQL$^{\mathrm{virt}}$ program R of Theorem 5.3.

As explained in Section 2, formulas over strings use the vocabulary $\{<, (O_\sigma)_{\sigma\in\Sigma}\}$. We start by introducing some notation. We denote by $x_1, x_2 \in]x^\ell, x^r[$, the formula

$$x^\ell < x_1 \wedge x_1 < x^r \wedge x^\ell < x_2 \wedge x_2 < x^r$$

expressing that x_1, x_2 belong to the interval $]x^\ell, x^r[$. In the sequel we use variables x_i^ℓ and x_i^r, which will be interpreted by matching markers X_i and $\bar{X}_i$, respectively.

Let φ'' be the formula obtained from φ' by recursively replacing every subformula of the form $\exists x_i \alpha$ (for $i > 1$) by

$$\begin{aligned}&\exists x_i^\ell \exists x_i^r (x_i^\ell, x_i^r \in]x_{i-1}^\ell, x_{i-1}^r[\wedge O_{X_i}(x_i^\ell) \wedge O_{\bar{X}_i}(x_i^r)\\ &\qquad\qquad \wedge \neg\exists y((O_{X_i}(y) \vee O_{\bar{X}_i}(y)) \wedge y \in]x_i^\ell, x_i^r[),\\ &\qquad\qquad\qquad\qquad \wedge \alpha)\end{aligned}$$

and $\forall x_i \alpha$ (for $i > 1$) by

$$\begin{aligned}&\forall x_i^\ell \forall x_i^r (x_i^\ell, x_i^r \in]x_{i-1}^\ell, x_{i-1}^r[\wedge O_{X_i}(x_i^\ell) \wedge O_{\bar{X}_i}(x_i^r)\\ &\qquad\qquad \wedge \neg\exists y((O_{X_i}(y) \vee O_{\bar{X}_i}(y)) \wedge y \in]x_i^\ell, x_i^r[)\\ &\qquad\qquad\qquad\qquad \to \alpha).\end{aligned}$$

Then, replace $\exists x_1 \alpha$ and $\forall x_1 \alpha$ by

$$\exists x_1^\ell \exists x_1^r (O_{X_1}(x_1^\ell) \wedge O_{\bar{X}_1}(x_1^r) \wedge \neg\exists y((O_{X_1}(y) \vee O_{\bar{X}_1}(y)) \wedge x_1^\ell < y \wedge y < x_1^r) \wedge \alpha)$$

and

$$\forall x_1^\ell \forall x_1^r (O_{X_1}(x_1^\ell) \wedge O_{\bar{X}_1}(x_1^r) \wedge \neg\exists y((O_{X_1}(y) \vee O_{\bar{X}_1}(y)) \wedge x_1^\ell < y \wedge y < x_1^r) \to \alpha),$$

respectively. So far, φ'' has translated all quantifications of φ', but has left unchanged its quantifier-free part $\delta(x_1, \ldots x_n)$. Finally, let φ''' be the sentence obtained from φ'' by replacing the quantifier-free part $\delta(x_1, \ldots x_n)$, by

$$\exists y, y_1 \ldots y_{m_i} \in]x_n^\ell, x_n^r[\quad \left(\bigvee_{i=1}^{m} \left(O_{P^i}(y) \wedge \bigwedge_{j=1}^{m_i} \neg O_{N_j^i}(y_j) \right) \right).$$

Define the DTD as mapping result to the language defined by φ''', which is star-free as mentioned above. Clearly, φ is valid iff for every $\mathcal{A}$ with ordering $<$, $w \models \varphi'''$ for $R(\mathcal{A}, <) = \text{result}(w)$. Hence, R typechecks iff φ is valid.

□

Theorems 5.1 and 5.3 highlight an interesting trade-off between specialization in output DTDs and virtual nodes in queries.

The undecidability result in Theorem 5.3 requires DTDs using SF formulas. The next proposition shows that restricting the DTD language to $\mathcal{SL}$ renders typechecking decidable, even when virtual nodes are allowed, the queries in the program can use equality and negation, and the input is constrained by FO($\exists^*\forall^*$ formulas.

Proposition 5.4. TC[$\text{CQ}_{virt}^{\neg,=}$, $\mathcal{SL}$, FO($\exists^*\forall^*$)] *is decidable.*

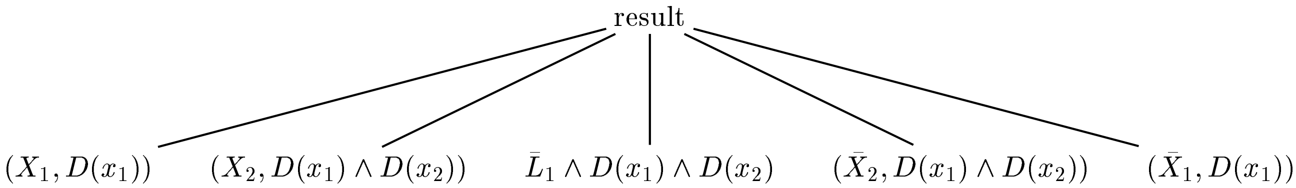


Figure 13: The flattened TreeQL program R of Figure 12.

Proof. We reduce TC[$\mathrm{CQ}^{=,\neg}_{virt}$, $\mathcal{SL}$, FO($\exists^*\forall^*$)] to TC[$\mathrm{CQ}^{=,\neg}$, $\mathcal{SL}$, FO($\exists^*\forall^*$)], which is decidable by Theorem 3.2.

Define the function ρ which maps $(\Sigma_\# \times \mathrm{CQ}^{=,\neg})$-trees to forests by eliminating #-labeled nodes and expanding the logical formulas recursively as follows. Let R be the tree $(\sigma, \varphi(\bar{x}))(R_1, \ldots, R_n)$. Then

$$\rho(R) := \begin{cases} (\sigma, \varphi(\bar{x}))(\rho(R_1), \ldots, \rho(R_n)) & \text{if } \sigma \neq \#; \\ \rho(R'_1), \ldots, \rho(R'_n) & \text{if } \sigma = \#. \end{cases}$$

Here, R'_i is obtained from R_i by replacing the label $(\delta, \psi(\bar{x}, \bar{y}))$ of the root of R_i by $(\delta, \psi(\bar{x}, \bar{y}) \wedge \varphi(\bar{x}))$.

Intuitively, the transformation ρ eliminates virtual nodes from the program, in a manner similar to the "flattening" transformation $\lambda_\#$ used to eliminate virtual nodes from answers to queries (see Section 2.6). Note that the two operators do not commute, i.e. applying flattening to the program before it is evaluated is not equivalent to applying flattening to the answer after the program is evaluated. For example, flattening the program with virtual nodes in Figure 12 (with $n = 2$ and $m = 1$) yields the program in Figure 13 (where $\bar{L}_1 \wedge D(x_1) \wedge D(x_2)$ denotes the sequence of nodes $\bar{L}_1$ where the conjunct $D(x_1) \wedge D(x_2)$ is added to each formula). Clearly, the latter program is not equivalent to the first. Indeed, the results of the two programs differ in the ordering of the nodes. However, it is easily seen that the number of nodes with a given label occurring under the root is the same. Thus, the outputs are not distinguishable by $\mathcal{SL}$ formulas, which ignore node ordering.

More precisely, we say that two trees t_1, t_2 are similar, denoted $t_1 \approx t_2$, if there is function p from nodes(t_1) to nodes(t_1) which is a permutation of the children of each node such that $p(t_1) = t_2$. A straightforward proof by induction shows that for any $\mathrm{TreeQL}^{\mathrm{virt}}$ program R, input $\mathcal{A}$ and ordering $<$, $R(\mathcal{A}, <) \approx \rho(R)(\mathcal{A}, <)$. As $\mathcal{SL}$ formulas do not take order into account, R typechecks w.r.t. an output type d iff $\rho(R)$ typechecks w.r.t. d. $\square$

Another way to strengthen the TreeQL formalism is to allow programs with more expressive formulas. We can show the following.

Proposition 5.5. TC[FO($\exists^*\forall^*$), $\mathcal{SL}^r$, $\emptyset$] *is undecidable.*

Proof. It is well-known that satisfiability of formulas of the form $\forall\bar{x}\exists\bar{y}\alpha(\bar{x}, \bar{y})$ where α is quantifier-free, is undecidable (see, e.g., [7]). Note that such a formula is not satisfiable iff its negation $\exists\bar{x}\forall\bar{y}\neg\alpha(\bar{x}, \bar{y})$ is valid. This holds iff the valid formula $\forall x(x = x)$ is contained in $\exists\bar{x}\forall\bar{y}\neg\alpha(\bar{x}, \bar{y})$. That is, for all databases $\mathcal{A}$, $\mathcal{A} \models \forall x(x = x) \rightarrow \exists\bar{x}\forall\bar{y}\neg\alpha(\bar{x}, \bar{y})$. Given $\psi := \forall\bar{x}\exists\bar{y}\alpha(\bar{x}, \bar{y})$, we define the TreeQL program R as

$$
\begin{array}{c}
\text{result} \\
| \\
(a, \forall x(x = x)) \\
| \\
(b, \exists \bar{x} \forall \bar{y} \neg \alpha(\bar{x}, \bar{y}))
\end{array}
$$

and define the DTD d by $d(\text{result}) := \text{true}$ and $d(a) = b$. Clearly, R typechecks w.r.t. d iff $\forall x(x = x)$ is contained in $\exists \bar{x} \forall \bar{y} \neg \alpha(\bar{x}, \bar{y})$. So, R typechecks w.r.t. d iff ψ is not satisfiable. $\square$

Next, we consider the effect of the constraints on decidability. We show that even the usually well-behaved unary AcIDs (which are not definable in FO($\exists^*\forall^*$)) render typechecking undecidable.

Theorem 5.6. TC[$\mathrm{CQ}^{\neg,=}$, $\mathcal{SL}^r$, unary AcIDs] *is undecidable.*

Proof. We consider the fragment of FO consisting of formulas of the form $\forall x \varphi(x)$ where φ is a quantifier-free formula over the vocabulary of two unary functions f and g. It is well-known that it is undecidable whether there is a non-empty structure $\mathcal{A}$ such that $\mathcal{A} \models \forall x \varphi(x)$ (see, e.g., [7]). The schema of the input database consists of the two binary relations F and G (representing the functions f and g), and a unary relation D representing the active domain of the structure. Using D will allow to eliminate circular dependencies.

First, we have to make sure that F and G are indeed functions, that their domain is D, and their range is included in D. These are specified by the *cyclic* unary inclusion dependencies

$$
\begin{array}{llll}
(a) & F[1] \subseteq D[1] & (e) & D[1] \subseteq F[1] \\
(b) & G[1] \subseteq D[1] & (f) & D[1] \subseteq G[1] \\
(c) & F[2] \subseteq D[1] & & \\
(d) & G[2] \subseteq D[1]. & &
\end{array}
$$

However, we will only keep the dependencies (e) and (f): we show that (a)–(d) can be expressed by the TreeQL program itself. We next describe this TreeQL program in detail. We first check whether the inclusion dependency (a) holds. If not we generate the flag (a)`_does_not_hold`.

$$
\begin{array}{c}
\text{result} \\
| \\
((a)\texttt{\_does\_not\_hold}, \exists x \exists y (F(x, y) \wedge \neg D(x)))
\end{array}
$$

The same is done for the dependencies (b)–(d). Next we have to check whether F is indeed a function and not a relation. For instance, both (a, b) and (a, c), with $b \neq c$, could belong to F. This can be detected as follows result

(wrong_F,$\exists x \exists y \exists z (F(x,y), F(x,z) \wedge y \neq z)$)

The same is done for G. In particular, if G is a relation and not a function then the flag wrong_G is raised.

We test whether $\mathcal{A} \not\models \forall x \varphi(x)$, that is, $\mathcal{A} \models \exists x \neg \varphi(x)$. We can rewrite $\exists x \neg \varphi(x)$ to

$$\bigvee_{i=1}^{n} (\exists x) L_i,$$

where each L_i is of the form $\bigwedge_{j=1}^{m_i} C_j^i$ where each C_j^i is an equality or an inequality between terms. For instance, $C_1 \equiv fgx = ffx$ (parenthesis omitted for clarity) or $C_2 \equiv fgx \neq ffx$. Obviously, there is a canonical way to associate a $\mathrm{CQ}^{=,\neg}$ with each C. For instance,

$$\varphi_{C_1}(x) = \exists y_2, y_3, z_2, z_3 (G(x,y_2) \wedge F(y_2,y_3) \wedge F(x,z_2) \wedge F(z_2,z_3) \wedge y_3 = z_3),$$

and

$$\varphi_{C_2}(x) = \exists y_2, y_3, z_2, z_3 (G(x,y_2) \wedge F(y_2,y_3) \wedge F(x,z_2) \wedge F(z_2,z_3) \wedge y_3 \neq z_3).$$

Further, we define φ_{L_i} as $\varphi_{C_1^i}(x) \wedge \ldots \wedge \varphi_{C_{m_i}^i}(x)$. The just described part of the TreeQL query is then of the form:

result

(L_1,$\exists x \varphi_{L_1}(x)$) ... ($L_n$,$\exists x \varphi_{L_n}(x)$)

Hence, $\mathcal{A} \not\models \forall x \varphi(x)$ whenever one of the error flags L_i is raised.

Finally, we have to make sure that D is non-empty. Therefore we have

result

(D-not_empty, $\exists z D(z)$)

The final TreeQL program is the concatenation of the previous programs (that is, the concatenation of all children under one result node). Note that a non-empty input structure for which $\mathcal{A} \models \forall x \varphi(x)$ simply generates the tree

result

(D-not_empty)

The output DTD d then maps result to

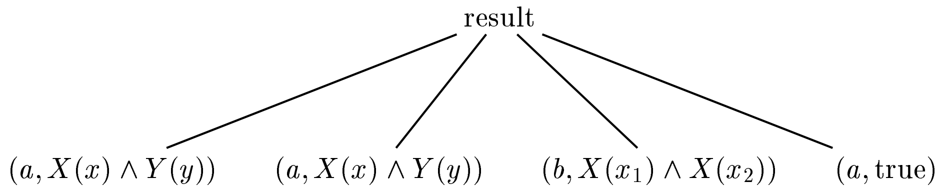
$$D\text{-not\_empty} \rightarrow \text{error}$$

where error is the disjunction over all error flags. If R does not typecheck w.r.t. d, then there is an $\mathcal{A}$ and an ordering $<$ such that $R(\mathcal{A},<) \notin L(d)$. By construction, $\mathcal{A}$ is non-empty and no error flag is raised. Therefore, $\mathcal{A}_{|D} \models \forall x \varphi(x)$. Conversely, if there is an $\mathcal{A}$ such that $\mathcal{A} \models \forall x \varphi(x)$ then for every ordering $<$, $R(\mathcal{A} \cup D,<) \notin L(d)$, where D is interpreted by the active domain of $\mathcal{A}$. $\square$

Theorem 3.6 showed that typechecking remains decidable even for DTDs using full regular languages, as long as the queries are restricted to be projection-free. As shown next, going beyond regular languages quickly leads to undecidability.

Theorem 5.7. TC[projection-free CQ, DCFL, $\emptyset$] *is undecidable.*

Proof. The proof is a reduction from Hilbert's tenth problem, diophantine equations, well-known to be undecidable [20]. We consider the following variant. For a polynomial $P(x_1,\ldots,x_n)$ with integer coefficients, are there positive integers $i_1,\ldots,i_n$ such that $P(i_1,\ldots,i_n)=0$? We only give the reduction by example. The general case is a straightforward generalization. Consider, for instance, the polynomial $2xy - x^2 + 1$. The input database consists of two sets X and Y where the cardinalities of X and Y stand for the numbers x and y, respectively. We describe a TreeQL program that generates from X and Y sequences of a's and b's. A positive term in P generates a's while a negative one generates b's. Hence, an a stands for $+1$, and a b stands for -1. The output DTD states that the number of a's differs from the number of b's. This holds iff $|X|$ and $|Y|$ do not form a solution to P, and the language specified by the DTD can easily be recognized by a deterministic PDA. The TreeQL program is a tree of depth one. For the example polynomial, the program is



Here, the first two symbols correspond to the term $2xy$ and generate a's as the term is positive; similarly, the third and the fourth symbol correspond to $-x^2$ and $+1$, respectively. The output generates sequences of a's and b's. The deterministic PDA accepts when the number of a's is different from the number of b's. Hence, the TreeQL program typechecks iff the diophantine equation has no positive solution. $\square$

6 Conclusions

We investigated the problem of typechecking XML views of relational databases satisfying given integrity constraints. This is a practically important problem in the context of the Web, where relational databases must be exported in XML form that satisfies target DTDs. The formal query language TreeQL maps first-order relational structures to tree data, and is a faithful abstraction of the view definition language used in the SilkRoute prototype. The results of the paper trace a fairly tight border of decidability for the typechecking problem. The parameters considered include features of the query language, of the DTDs, and the class of integrity constraints satisfied by the relational database. The proofs bring into play a variety of techniques at the confluence of finite-model theory, language theory, and combinatorics.

As it turns out, the results of the paper are largely a bearer of bad news with regard to the feasibility of typechecking. Indeed, they show that typechecking when data values are present quickly becomes undecidable. Some of the decidable cases have prohibitively high complexity (all the way to non-elementary) while others are more in line with the typical complexity of static analysis of conjunctive queries (PSPACE to CONEXPTIME) . On the positive side, we also exhibit a restriction of possible practical interest for which typechecking is in PTIME.

Altogether, the complexity results have to be taken with a grain of salt, as they are often overly pessimistic. As discussed in Remark 4.5, practical implementations often manage to perform in realistic cases better than the worst-case analysis would suggest.

Acknowledgments

We thank the anonymous referees whose comments improved the presentation of the paper.

References

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

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[3] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: typechecking revisited. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pp. 138–149, 2001.

[4] 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/`.

[5] C. Beeri and T. Milo. Schemas for integration and translation of structured and semistructured data. In *Int'l. Conf. on Database Theory*, pages 296–313, 1999.

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

[7] E. Börger, E. Grädel, and Y. Gurevich. *The classical decision problem.* Springer, 1997.

[8] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets: version 1, April 3, 2001. `http://www.cs.ust.hk/tcsc/RR/2001-05.ps.gz`

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

[10] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *Proc. ACM SIGMOD Conf.*, pages 177–188, 1998.

[11] E. F. Codd. A Relational Model for Large Shared Databanks. *Communications of the ACM*, 13 (6), pp. 377-387, 1970.

[12] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: publishing object-relational data as XML. In *WebDB 2000 (Informal Proceedings)*, Dallas, TX, pp. 105-110.

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

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

[15] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Second edition. Springer, 1999.

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

[17] M. Fernandez, Y. Kadiyska, A. Morishima, D. Suciu, and W. Tan. SilkRoute : a framework for publishing relational data in XML, 2002. manuscript available from `www.cs.washington.edu/homes/suciu`.

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

[19] N. Immerman. Upper and lower bounds for first-order expressibility. *Journal of Computer and System Sciences*, 25(1), pp. 76–98, 1982.

[20] Y. V. Matiyasevich. *Hilbert's tenth problem.* Foundations of Computing Series. MIT Press, 1993.

[21] R. McNaughton and S. Papert. *Counter-Free Automata.* MIT Press, 1971.

[22] T.Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pp. 11-22, 2000.

[23] T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In *Proc. Int'l. Conf. on Very Large Databases (VLDB)*, pp. 122-133, 1998.

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

[25] F. Neven and T. Schwentick. Unordered DTDs. Unpublished manuscript, 1999.

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

[27] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pp. 35-46, 2000.

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

[29] W. Thomas. Languages, automata, and logic. In Rozenberg and Salomaa, *Handbook of Formal Languages*, volume III, chapter 7. Springer, 1997.

[30] R. van der Meyden. The complexity of querying infinite data about linearly ordered domains *Journal of Coumputer and System Sciences*, 54(1):113-135, 1997.

[31] J. Shanmugasundaram, , J. Kiernana, E. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *Proceedings of VLDB*, pages 261–270, Rome, Italy, September 2001.

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

[33] M. Vardi. On the complexity of bounded-variable queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pp. 266–276, 1995.

Appendix

Proof of Lemma 3.3. We prove the lemma by induction on the structure of star-free regular expressions. We distinguish the following cases:

1. If $r = \emptyset$ or $r = \varepsilon$ then ρ_r is the empty disjunction and consists of the only disjunct $\sigma_1^{=0} \cdots \sigma_n^{=0}$, respectively.
2. If $r = \sigma_i$, then $\rho_r := \sigma_1^{=j_1} \cdots \sigma_n^{=j_n}$ with $j_i = 1$ and $j_\ell = 0$ for $\ell \neq i$.
3. If $r = r_1 + r_2$ then ρ_r consists of the disjuncts in ρ_{r_1} and ρ_{r_2}.
4. If $r = r_1 \cdot r_2$, then ρ_r consists of the disjuncts obtained from ρ_{r_1} and ρ_{r_2} as follows. Add
$$\sigma_1^{*_1 i_1} \cdots \sigma_n^{*_n i_n}$$
each time there is a disjunct
$$\sigma_1^{*_1^1 i_1^1} \cdots \sigma_n^{*_n^1 i_n^1}$$
in ρ_{r_1} and a disjunct
$$\sigma_1^{*_1^2 i_1^2} \cdots \sigma_n^{*_n^2 i_n^2}$$
in ρ_{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 ρ_r is the intersection of the negation of the disjuncts of ρ_s. The negation of a single disjunct $\sigma_1^{*_1 i_1} \cdots \sigma_n^{*_n i_n}$ is equivalent to the disjunction obtained by adding for every $j = 1, \ldots, n$, the disjuncts

$$\sigma_1^{*_1 i_1} \cdots \sigma_j^{=0} \cdots \sigma_n^{*_n i_n},\ \sigma_1^{*_1 i_1} \cdots \sigma_j^{=1} \cdots \sigma_n^{*_n i_n},\ \ldots,\ \sigma_1^{*_1 i_1} \cdots \sigma_j^{=i_j - 1} \cdots \sigma_n^{*_n i_n}.$$

If $*_j =$ '$=$', then we also add

$$\sigma_1^{*_1 i_1} \cdots \sigma_j^{\geq i_j + 1} \cdots \sigma_n^{*_n i_n}.$$

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 $\bigwedge_k \sigma_1^{*_1^k i_1^k} \cdots \sigma_n^{*_n^k i_n^k}$. 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 $\sigma_1^{*_1 i_1} \cdots \sigma_n^{*_n i_n}$, 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 $\leq |s|$. Further, as there are at most $(2|s|)^n$ possible disjuncts, the size of ρ_r is exponential in $|s|$. However, the above algorithm is double exponential in the size of $|s|$. 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 ρ_r in exponential time.

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

$$\bigvee\{\sigma_1^{*_1 i_1} \cdots \sigma_n^{*_n i_n} \mid (*_j = \text{'='} \text{ and } i_j \leq |s|) \text{ or } (*_j = \text{'}\geq\text{'} \text{ and } i_j = |s| + 1), \text{ and } \sigma_1^{i_1} \cdots \sigma_n^{i_n} \in L(r)\}.$$

As there are only exponentially many strings $\sigma_1^{*_1 i_1} \cdots \sigma_n^{*_n i_n}$ and $\sigma_1^{i_1} \cdots \sigma_n^{i_n} \in L(r)$ 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 $\sigma_1^{*_1 i_1} \cdots \sigma_n^{*_n i_n}$ satisfies this formula. This is well-known to be in PSPACE. □

Proof of Lemma 3.5 Let $M = (Q, \Sigma, \delta, q_0, F)$ be a non-deterministic finite-state automaton accepting r, where Q is the set of states, $\Sigma = \{\delta_1, \ldots, \delta_n\}$, $\delta : Q \times \Sigma \to 2^Q$ 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 $r \cap \delta_1^* \ldots \delta_n^*$, where $u_1 \in \delta_1^*, \ldots, u_n \in \delta_n^*$, 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 δ_m let $L^m_{pq} = \{\delta^h_m \mid q \in \delta(p, \delta^h_m)\}$, and for each vector $\vec{q} = q_0 \ldots q_n \in \mathbf{Q}$ let $L_{\vec{q}} = L^1_{q_0 q_1} \ldots L^n_{q_{n-1} q_n}$. Clearly,

$$r \cap \delta^*_1 \ldots \delta^*_n = \bigcup_{\vec{q} \in \mathbf{Q}} L_{\vec{q}}.$$

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

$$L^m_{pq} = L_{\leq k} \cup \bigcup_{i \in I} w_i (a^j)^*.$$

Thus, words in L^m_{pq} 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 $w_i(a^j)^*$ consists of the words of length $|w_i| + \alpha$ where $\alpha \equiv 0 \ mod \ j$ so is described by the pair $(|w_i|, j)$. Thus, each language L^m_{pq} 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. $\Box$