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 [8], 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 offer now 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 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. As explained, this is a critical problem in XML data exchange. In addition, as we show here, this problem is also technically interesting and non-trivial from a theoretical perspective.

We define a language, TreeQL, 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

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

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

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

asked to check whether every input structure satisfying the constraints is mapped into a tree in the output type. Solving the typechecking problem boils down to checking 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, combining 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 [15]. 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, since our language is declarative and there is no notion of an execution. 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 type checking rapidly becomes undecidable if we allow the transformation language or the output types to be too expressive. In contrast, type inference is generally decidable for functional programming languages (that are Turing complete) and for powerful type systems.

Our work is motivated by the practical need to typecheck XML views from relational databases. SilkRoute [10] 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 tree transformations. In previous work [14] 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 labeled over a fixed, finite alphabet Σ. So they do not take into account the data values present in XML documents. In subsequent work [3] we considered trees with labels from an infinite alphabet, that model more closely XML trees where internal nodes have labels from a known, fixed alphabet, while leaves contain data values from an infinite domain. We showed that typechecking quickly becomes undecidable, even if one considers very restricted transformations. 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. 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. However, some of the lower bound results in the present paper can be transferred to the XML context and strengthen results from [3]. A more detailed comparison is deferred to the full version of this paper. Conversely,

Organization The paper is organized as follows. The first section develops the basic formalism, including our abstraction of XML documents, DTDs, and the variant of TreeQL used as transformation language. Section 3 presents the decidability results; Section 4 the complexity analysis; and Section 5 the undecidability results. The paper ends with brief conclusions. Due to space limitations, some proofs are only sketched or omitted entirely.

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.

Trees. Trees are our abstraction of XML documents [1]. They capture the nesting structure of XML elements and their tags. We refrain from modeling data values as they are not relevant w.r.t. typechecking. Indeed, output types only constrain the structure of the output tree not the data values at the leaves. 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 trees over Σ by $\mathcal{T}_\Sigma$. By $\text{root}(t)$, we denote the root of 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. By $\sigma(t_1, \dots, t_n)$, where $t_1, \dots, t_n$ are trees, we mean the tree where the root is labeled with σ and the i-th subtree is t_i.

Types and DTDs. DTDs and their variants provide a typing mechanism for XML documents. We use several notions of types for trees. For $\mathcal{C}$ a class of string languages over Σ, a *DTD over* Σ *w.r.t.* $\mathcal{C}$ is a mapping from Σ to languages in $\mathcal{C}$. We denote the class of all such DTDs by $\text{DTD}(\mathcal{C})$. Let $d \in \text{DTD}(\mathcal{C})$. Then, a Σ-tree t *satisfies* d, if for every node v of t with children $v_1, \dots, v_n$, $\text{lab}(v_1) \cdots \text{lab}(v_n) \in d(\text{lab}(v))$. Note that, if $n = 0$, then ε should 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. Star-free regular languages are defined by the *star-free* regular expressions, which are build from single symbols and ε, using concatenation, union, and complement. They correspond exactly to the languages defined by first-order logic (FO) over the vocabulary $\{<, (O_\sigma)_{\sigma \in \Sigma}\}$ where $<$ is a binary relation and every O_σ is a unary relation [13, 18]. A string $w = \sigma_1 \dots \sigma_n$ is then represented by the logical structure $(\{1, \dots, n\}; <, (O_\sigma)_{\sigma \in \Sigma})$ where $<$ is the natural order on $\{1, \dots, n\}$, and for each i, $i \in O_\sigma$ iff $\sigma_i = \sigma$.

We will consider an even simpler class of DTDs, which specify cardinality constraints on the tags of children of a node, but does not restrict their order. Such DTDs are useful either when order is irrelevant, or when the order of tags in the output is hard-wired by the syntax of the query and so can be factored out. We use a logic called $\mathcal{SL}$, inspired by [16]. The syntax of the language is as follows. For every $\sigma \in \Sigma$ and natural number i, $\sigma^{=i}$ and $\sigma^{\geq i}$ are *atomic* $\mathcal{SL}$ *formulas*; true is also an atomic $\mathcal{SL}$ formula. Every atomic formula is a formula and the negation, conjunction, and disjunction of formulas are also 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. [1] Satisfaction of Boolean combination 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, also present in more recent DTD proposals such as XML-Schemas [4, 5]. This is motivated by a severe limitation of basic DTDs: their definition of the type of a given tag depends only on the tag itself and not on the context in which it occurs. For example, this means that the singleton $\{t\}$ where t is the tree $a(b(c), b(d))$ cannot be described by a DTD, because the "type" of the first b differs from that of the second b. This naturally leads to an extension of DTDs with *specialization* (also called decoupled types) which, intuitively, allows defining the type of a tag by several "cases" depending on the context. Formally, we have:

Definition 2.1. *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* $t \in \mu(L(d))$. *We denote the set of all such specialized DTDs 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 $\mu(a') = a$. We also denote by μ the homomorphism induced on strings and trees. Interestingly, it turns out that the class S-DTD(REG) is precisely equivalent to the class of regular tree automata over unranked trees [7, 17]. This is more evidence that specialized DTDs are a robust and natural specification mechanism.

Logic. Consider some fixed relational vocabulary $\mathcal{S}$. A database over $\mathcal{S}$ is just an $\mathcal{S}$-structure defined in the usual way [2, 9]. We denote the domain of a database $\mathcal{A}$ by $\text{dom}(\mathcal{A})$. Further, let $\mathcal{L}$ be a logic over $\mathcal{S}$. Then we denote the free variables occurring

[1]The empty string is obtained by $\bigwedge_{\sigma \in \Sigma} \sigma^{=0}$ and the empty set by $\neg$true. We, hence, use ε and $\emptyset$ as shorthands in $\mathcal{SL}$ formulas.

in $\varphi \in \mathcal{L}$ by Free(φ). 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, that is, 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 $\mathcal{S}$ (so, no equalities). By CQ with superscripts in $\{=, \neg\}$ we mean CQ 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 FO($\exists^*\forall^*$).

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 FO($\exists^*\forall^*$). Note 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 coordinates 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 dependencies 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 $R_1[\bar{i}_1] \subseteq R_2[\bar{j}_2]$, $R_2[\bar{i}_2] \subseteq R_3[\bar{j}_3]$, …, $R_m[\bar{i}_m] \subseteq R_1[\bar{j}_1]$.

A set of dependencies is *acyclic* when it is not cyclic. We denote the class of acyclic inclusion dependencies by AcIDs.

Finally, we recall the following technical notion. For a finite set of variables X, an *X-substitution θ for $\mathcal{A}$* is a mapping from X to dom($\mathcal{A}$). Let $\bar{x}$ be variables not occurring in X and let $\bar{a}$ be as many elements of 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 $\theta(y)$.

TreeQL. The transformation language we consider, mapping databases to trees, is an abstraction of RXL [10]. We refer to it as TreeQL. The queries are tree patterns where nodes are labeled with label-formula pairs. Therefore, 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 $\mathcal{T}_{\Sigma\times\mathcal{L}}$. In the next definition, denote by formula(v) the formula associated to a node v.

Definition 2.2. *A* TreeQL($\mathcal{L}, \Sigma$) program *is a tree $P \in \mathcal{T}_{\Sigma\times\mathcal{L}}$ such that Free(formula(v)) $\subseteq$ Free(formula(v')), for all nodes v and v' where v' is a descendant of v; in addition, the formula in the label of the root is equivalent to true.*

If $\mathcal{L}$ or Σ are clear from the context or not important, we sometimes omit them. Sometimes, we abbreviate the label (σ, true) simply by σ.

Let $\mathcal{A}$ be a database over $\mathcal{S}$, $<$ a total order on dom($\mathcal{A}$), and P a TreeQL program.

Definition 2.3. *The tree $P(\mathcal{A}, <)$ generated by P from $\mathcal{A}$ and $<$ is defined as follows. Its nodes consist of pairs of the form (v, θ) where v is a node of P and θ an $\bar{x}$-substitution (where $\bar{x}$ = Free(formula(v))) such that $\mathcal{A} \models \varphi[\theta]$ for every formula φ labeling v or labeling an ancestor of v in P. The root is $(root(P), ())$ and nodes are ordered component-wise, using the node order in P for v and the lexicographic order $<$ on θ. 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. Finally the label of a node (v, θ) is the Σ label of v in P.*

Example 2.4. *Consider the TreeQL(CQ) program $P = v_0(v_1, v_2, v_3)$ (i.e. the tree has root node v_0 with children v_1, v_2, v_3) and $lab(v_0) = (a, true)$, $lab(v_1) = (b, R(x, y) \wedge R(y, x))$, $lab(v_2) = (c, R(x, y))$, $lab(v_3) = (d, R(x, y) \wedge R(u, v))$, and consider 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.*

We remark that RXL [10], the language TreeQL is an abstraction of, also allows to output data values occurring in the input database as labels of leaves in XML documents. However, as we study typechecking and output types do not constrain these data values we chose to omit them from the formalism.

An extension: TreeQL with virtual nodes. We will use an extension of TreeQL that allows programs to define "temporary" nodes, called *virtual*, that are eliminated in the final answer. To see why this is useful, consider an input binary relation R providing titles and speakers of talks (ordered alphabetically by title). Suppose we wish to output a tree listing under the root the ordered title/speaker pairs. This cannot be defined by a TreeQL program, because it cannot group the titles and speakers as required. However, suppose we can use temporary nodes, identified by a special label #. Consider the query $root((\#, R(t,s))((title, R(s,t)), (speaker, R(s,t))))$. This produces one node labeled # for each tuple in R, whose children are the corresponding title and speaker. The ordered sequence of title/speaker pairs 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. Define the function $\lambda_\#$ which maps trees to forests by eliminating #-labeled nodes, recursively 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 2.5. A TreeQL($\mathcal{L}, \Sigma$) *program P with virtual nodes* is a TreeQL($\mathcal{L}, \Sigma_\#$) program where $\text{lab}(\text{root}(P)) \notin \{\#\} \times \mathcal{L}$. We denote the set of all such programs by $\text{TreeQL}^{\text{virt}}(\mathcal{L}, \Sigma)$. 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 $P(\mathcal{A}, <)$.

Typechecking. We next formalize the central problem of this paper.

Definition 2.6. *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 dom($\mathcal{A}$).*

Example 2.7. *Continuing with Example 2.4, consider the DTD defined by the mapping $d : \{a, b, c, d\} \to REG$ given by:*

$$d(a) = (b^*.(c.c)^*.(d.d)^*) \mid (b^*.(c.c)^*.c.(d.d)^*.d)$$

and $d(b) = d(c) = d(d) = \varepsilon$. The type says that there are an even number of c's and d's or an odd number of both under nodes labeled a. Then the TreeQL program P in Example 2.4 typechecks w.r.t. this DTD.

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

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

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

Clearly, 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 $\text{result}((a, \varphi))$ with an output type d that maps $d(\text{result})$ to $\{\varepsilon\}$. 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) When restricting output DTDs to star-free languages we show that typechecking is decidable for TreeQL($\text{CQ}^{=,\neg}$) programs and integrity constraints in FO($\exists^*\forall^*$). The proof gives a CONEXPTIME upper bound. In Section 4, we provide the matching lower bound.

(ii) By restricting the queries to projection-free CQs and the integrity constraints to FDs, we show that typechecking w.r.t. DTDs with full regular expressions is decidable. 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 results are essentially optimal: slight increase of the power of the DTDs or the integrity constraints lead to undecidability. However, it remains open whether in (ii) above, the restriction to projection-free CQs is required. We first consider star-free output types and integrity constraints in FO($\exists^*\forall^*$).

Theorem 3.1. TC[$\text{CQ}^{=,\neg}$, SF, FO($\exists^*\forall^*$)] *is in* CONEXPTIME.

Proof. The decidability is shown by bounding the size of inputs that need to be checked to detect a violation of the output DTD. Let R be a TreeQL($\text{CQ}^{=,\neg}$) program, let $d \in$ DTD(SF), and let Φ be a finite set of FO($\exists^*\forall^*$) sentences.

We start by stating a technical lemma. 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.

Lemma 3.2. *Let r be a star-free regular expression. Then $r \cap \sigma_1^* \cdots \sigma_n^*$ is equivalent to a disjunction ρ_r of expression 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, $i_1, \ldots, i_n \leq |r|$, the size of ρ_r is exponential in $|r| + n$, and ρ_r can be computed in time exponential in $|r| + n$.*

Note that R does *not* typecheck w.r.t. d iff

- there is a path $v_1, \ldots, v_k$ 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) v_k has precisely n children with labels $(\delta_1, \psi_1(\bar{x}, \bar{y}_1)), \ldots, (\delta_n, \psi_n(\bar{x}, \bar{y}_n))$ and 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) $\delta_1^{i_1} \cdots \delta_n^{i_n} \notin d(\sigma_k)$ with $|\{\bar{b} \mid \mathcal{A} \models \psi_j(\bar{a}, \bar{b})\}| = i_j$ for all $j = 1, \ldots, n$.

Let $d(\sigma_k)$ be represented by the star-free regular expression r. So, $\delta_1^{i_1} \cdots \delta_n^{i_n} \notin L(r)$. Since for each $\mathcal{A}$, this string will be of the form $\delta_1^* \cdots \delta_n^*$, it suffices to restrict attention to $\neg r \cap \delta_1^* \cdots \delta_n^*$. By Lemma 3.2, $\neg r \cap \delta_1^* \cdots \delta_n^*$ 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 $j_i \leq |r|$. Let D be a particular disjunct $\delta_1^{*_1 j_1} \cdots \delta_n^{*_n j_n}$ such that there is a structure $\mathcal{A}$ with elements $\bar{a} := \bar{a}_1, \ldots, \bar{a}_k$ with

(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 structure $\mathcal{B}$ of size polynomial in $|R| + |d| + |\Phi|$ satisfying (1) and (2). To see this, we introduce some notation. Suppose $\Phi = \bigcup_\ell \exists \bar{x}_\ell^\alpha \forall \bar{y}_\ell^\alpha \alpha_\ell(\bar{x}_\ell^\alpha, \bar{y}_\ell^\alpha)$, $\varphi_i(x_1, \ldots, x_i) = \exists \bar{x}_i^\varphi \gamma_i(x_1, \ldots, x_i, \bar{x}_i^\varphi)$, 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 $i = 1, \ldots, n$.

For each ℓ, pick a tuple $\bar{a}_\ell^\alpha$ such that $\mathcal{A} \models \forall \bar{y}_\ell^\alpha \alpha_\ell(\bar{a}_\ell^\alpha, \bar{y}_\ell^\alpha)$. Let E_1 be the set of these elements. Next, pick $a_1, \ldots, a_n$ and for each $i \in \{1, \ldots, k\}$ pick a tuple $\bar{a}_i^\varphi$ such that $\mathcal{A} \models \gamma_i(a_1, \ldots, a_i, \bar{a}_i^\varphi)$. Let E_2 be the set of these elements. Further, for $i = 1, \ldots, n$, pick j_i tuples $\bar{b}_i$ and for each such tuple pick a tuple $\bar{a}_i^\psi$ such that $\mathcal{A} \models \beta_i(a_1, \ldots, a_i, \bar{b}_i, \bar{a}_i^\psi)$. Let E_3 be the set of these elements. Note that the size of $E := E_1 \cup E_2 \cup E_3$ is at most polynomial in $|R| + |d| + |\Phi|$. Clearly, $|\{\bar{b} \mid \mathcal{A}_{|E} \models \psi_j(\bar{a}, \bar{b})\}| *_i j_i$ for $i = 1, \ldots, n$. Moreover, $\mathcal{A}_{|E} \models \Phi$. The latter follows by a standard argument (see, e.g., [6]). Indeed, for each ℓ, $(\mathcal{A}, E_1) \models \forall \bar{y}_\ell^\alpha \alpha_j(\bar{x}_\ell^\alpha, \bar{y}_\ell^\alpha)$, where the elements in E_1 are taken as constants. As these resulting sentences are universal, $(\mathcal{A}_{|E}, E_1) \models \forall \bar{y}_\ell^\alpha \alpha_\ell(\bar{x}_\ell^\alpha, \bar{y}_\ell^\alpha)$ 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. Then take $\mathcal{B}$ as $\mathcal{A}_{|E}$.

Hence, to look for a database that satisfies the disjunct D it suffices to guess one of exponential size. Recall that if we find such an $\mathcal{A}$, R does *not* typecheck w.r.t. d. The overall algorithm consists of two stages: (i) For every node v labeled with σ and with children $(\delta_1, \psi_1(\bar{x}, \bar{y}_1)), \ldots, (\delta_n, \psi_n(\bar{x}, \bar{y}_n))$, compute the normal form for $\neg d(\sigma) \cap \delta_1^* \cdots \delta_n^*$ as specified in Lemma 3.2. There is a linear number of nodes, so altogether we need exponential time. (ii) Subsequently, guess a path $v_1, \ldots, v_k$, a disjunct D, and a structure $\mathcal{A}$ such that the above holds. As described above this can all be done in NEXPTIME. □

The following result shows that decidability of typechecking holds 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 FDs. The proof is non-trivial and is based on Ramsey's theorem. It is similar to a proof in [3].

Theorem 3.3. TC[projection-free CQ$^{=,\neg}$, REG, FD] *is decidable.*

It remains open whether the projection-free restriction can be removed or whether the class of constraints can be extended.

4 Complexity

Theorem 3.1 provides an upper bound of CONEXPTIME on the complexity of type-checking. We show in this section that this is tight. Our proof requires negation and inequality in CQs. However, we show that even without these, typechecking remains intractable, more precisely DP-hard.[2] Nevertheless, 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

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

(see, e.g., [6]), to the complement of the typechecking problem.

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

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 $\neg\psi$ 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 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_is. Further, D has the following children

1. for each $i = 1, \dots, n$, $(\text{two}_i, \exists z_i \exists z_i'(D_i(z_i) \wedge D_i(z_i') \wedge z_i \neq z_i'))$, indicating that D_i has at least two elements; and
2. for each $j = 1, \dots, k$, $(@_j, L_j(\bar{x}, \bar{y}))$.

The output DTD d is of the following form $d(\text{result}) := \text{true}$ and

$$d(D) := \bigvee_{i=1}^{n} \text{two}_i \vee \bigvee_{j=1}^{k} @_j.$$

Suppose the TreeQL program R does not typecheck. Then at least one D and none of the two$_i$s appear. That is, all D_i are singleton sets. Let $D_i = \{d_i\}$ for each i. Further, none of the $@_j$s appear. Hence, $\mathcal{A} \not\models \exists \bar{y} \neg \psi(\bar{d}, \bar{y})$. Hence, $\mathcal{A} \models \exists \bar{x} \forall \bar{y} \psi$ and φ is satisfiable. Conversely, if $\mathcal{A}$ is a model of φ and we instantiate $D_1, \dots, D_n$ with the witnesses for the existential quantifiers then R does not typecheck for $\mathcal{A} \cup \{D_1, \dots, D_n\}$. □

Although it is unclear whether in Theorem 4.1, negation or inequality 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. CQ$^{\neq}$ denotes CQ with inequality.

Proposition 4.2. *1.* TC[CQ, $\mathcal{SL}^r$,$\emptyset$] *is* DP*-hard.*

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

The proof of Proposition 4.2 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 [11, 20]. 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.*

Theorem 4.3. 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 every bound variable occurs only once and is different from any free variable. For every non-leaf node v of R with children $v_1, \dots, v_n$, 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 $a_1 \vee \dots \vee a_k \vee \neg b_1 \vee \dots \neg b_m$. 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. $k > 0$ and $m > 0$. Then C_i is $(b_1 \wedge \dots \wedge b_m) \rightarrow (a_1 \vee \dots \vee a_k)$. We must check that

$$\exists(\psi_{b_1} \wedge \dots \wedge \psi_{b_m} \wedge \gamma) \\ \rightarrow \exists((\psi_{a_1} \wedge \gamma) \vee \dots \vee (\psi_{a_k} \wedge \gamma))$$

where the $\exists$ quantify all variables on the left, resp. righthand sides. From standard conjunctive query techniques it follows that the above holds iff there exists j such that

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

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 of $\exists(\psi_{b_1} \wedge \ldots \wedge \psi_{b_m} \wedge \gamma)$ is true. 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 $\exists((\psi_{a_1} \wedge \gamma) \vee \ldots \vee (\psi_{a_k} \wedge \gamma))$ is true on every input. This is false on the empty input, so the program does not typecheck.

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

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, and (*iii*) 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.

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

Proof. We use a reduction from satisfiability of first-order logic formulas over graphs without equality, which is well known to be undecidable (see, e.g., [6]). The satisfiability problem is to check, given an FO formula ψ, whether there is a non-empty graph $\mathcal{A}$ such that $\mathcal{A} \models \psi$. Let φ be the negation of ψ. We give the reduction by example. Assume $\varphi = \exists x_1 \forall x_2 \exists x_3 \delta(x_1, x_2, x_3)$, where δ is quantifier-free and 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 1. By L_i we denote the sequence

$$(P^i, P^i)(N_1^i, \tilde{N}_1^i) \ldots (N_{m_i}^i, \tilde{N}_{m_i}^i).$$

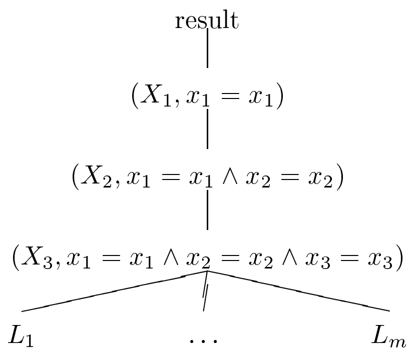
result
$(X_1, x_1 = x_1)$
$(X_2, x_1 = x_1 \wedge x_2 = x_2)$
$(X_3, x_1 = x_1 \wedge x_2 = x_2 \wedge x_3 = x_3)$
L_1
$\ldots$
L_m

Figure 1: The TreeQL program R.

Recall that the first component of the 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. The specialized DTD then takes care of the quantification pattern of φ. Indeed, it 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. To this end let $\Sigma' = \{Y_i, X_i \mid i \in \{1, \ldots, n\}\} \cup \{\text{result}\}$. Intuitively, whenever a node is labeled Y_i, this indicates that the path from the root to this node can be extended to a satisfiable path. Define $d(\text{result}) := Y_1 \vee \varepsilon$, $d(Y_1) := Y_2 \wedge \neg X_2$, $d(Y_2) := Y_3$, and $d(X_1) := d(X_2) := d(X_3)$. Here, ε makes sure the empty graph typechecks. Finally, set for each i, $\mu(X_i) := X_i$ and $\mu(Y_i) := X_i$. Clearly, R typechecks w.r.t. d iff $\mathcal{A} \models \varphi$ for every non-empty structure $\mathcal{A}$.

One can get rid of equality in the CQ's by introducing a relation containing all elements in the active domain. Details omitted. □

The next result shows that typechecking becomes undecidable when queries can use virtual nodes. The proof is similar to the proof of Theorem 5.1 and is omitted.

Theorem 5.2. TC[projection-free CQ$_{virt}$, SF, $\emptyset$] *is undecidable.*

Remark 5.3. The undecidability result in Theorem 5.5 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.

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

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 5.5. TC[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. [6]). 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 get rid of 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`.

$$\begin{tikzcd}\text{result} \arrow[d, no head] \\ ((a)\_\texttt{does\_not\_hold}, \exists x\exists y(F(x,y) \wedge \neg D(x))).\end{tikzcd}$$

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

$$\begin{tikzcd}\text{result} \arrow[d, no head] \\ (\texttt{wrong\_}F, \exists x\exists y\exists z(F(x,y), F(x,z) \wedge y \neq z)).\end{tikzcd}$$

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 CQ$^{=,\neg}$ with each C. For instance,

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

and

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

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:

$$\begin{tikzcd}[column sep=small] & \text{result} \arrow[dl, no head] \arrow[d, no head] \arrow[dr, no head] & \\ (L_1, \exists x\varphi_{L_1}(x)) & \ldots & (L_n, \exists x\varphi_{L_n}(x)).\end{tikzcd}$$

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

$$\begin{tikzcd}\text{result} \arrow[d, no head] \\ (D\text{-not\_empty}, \exists z D(z)).\end{tikzcd}$$

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-not_empty $\rightarrow$ 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 $\mathcal{A}$. □

Theorem 3.3 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 5.6. 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 [12]. We consider the following variant. For a polynomial $P(x_1, \dots, x_n)$ with integer coefficients, are there positive integers $i_1, \dots, i_n$ such that $P(i_1, \dots, i_n) = 0$? 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 nodes under the root are:

$$\begin{aligned}&(a, X(x) \wedge Y(y)) \cdot (a, X(x) \wedge Y(y))\\&\cdot (b, X(x_1) \wedge X(x_2))\\&\cdot (a, \text{true}).\end{aligned}$$

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. □

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.

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. To apper in *PODS* 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] P. Biron and A. Malhotra. XML schema part 2: Datatypes, May 1999. `http://www.w3.org/TR/xmlschema-2/`.

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

[7] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree languages over non-ranked alphabets, 1998.

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

[9] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer, 1995.

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

[11] N. Immerman. Upper and lower bounds for first-order expressibility. *J. of Computer and System Sciences*, vol.25, pp. 76–98, 1982.

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

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

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

[15] John C. Mitchell. *Foundations for Programmng Languages*, MIT Press, 1996.

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

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

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

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

[20] M. Vardi. On the complexity of bounded-variable queries. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pp. 266–276, 1995.