Reasoning about integrity constraints for tree-structured data*

Wojciech Czerwiński[1], Claire David[2], Filip Murlak[1], and Paweł Parys[1]

1 **University of Warsaw, Poland**
{wczerwin,fmurlak,parys}@mimuw.edu.pl
2 **Université Paris-Est Marne-la-Vallée, France**
claire.david@u-pem.fr

Abstract

We study a class of integrity constraints for tree-structured data modelled as data trees, whose nodes have a label from a finite alphabet and store a data value from an infinite data domain. The constraints require each tuple of nodes selected by a conjunctive query (using navigational axes and labels) to satisfy a positive combination of equalities and a positive combination of inequalities over the stored data values. Such constraints are instances of the general framework of XML-to-relational constraints proposed recently by Niewerth and Schwentick. They cover some common classes of constraints, including W3C XML Schema key and unique constraints, as well as domain restrictions and denial constraints, but cannot express inclusion constraints, such as reference keys. Our main result is that consistency of such integrity constraints with respect to a given schema (modelled as a tree automaton) is decidable. An easy extension gives decidability for the entailment problem. Equivalently, we show that validity and containment of unions of conjunctive queries using navigational axes, labels, data equalities and inequalities is decidable, as long as none of the conjunctive queries uses both equalities and inequalities; without this restriction, both problems are known to be undecidable. In the context of XML data exchange, our result can be used to establish decidability for a consistency problem for XML schema mappings. All the decision procedures are doubly exponential, with matching lower bounds. The complexity may be lowered to singly exponential, when conjunctive queries are replaced by tree patterns, and the number of data comparisons is bounded.

1998 ACM Subject Classification H.2.1 [**Database Management**]: Logical Design

Keywords and phrases data trees, integrity constraints, unions of conjunctive queries, schema mappings, entailment, containment, consistency

Digital Object Identifier 10.4230/LIPIcs.ICDT.2016.1

1 Introduction

Static analysis is an area of database theory that focuses on deciding properties of syntactic objects, like queries, integrity constraints, or data dependencies. The unifying paradigm is that since these objects are mostly user-generated, they tend to be small; hence, higher complexities are tolerable. Typical problems include satisfiability, validity, containment, and equivalence of queries [5, 8, 23, 24], and consistency and entailment of constraints [13, 26]. More specialized tasks include query rewriting in data integration scenarios [22], and manipulating schema

* The first, third, and fourth author of this paper were supported by Poland's National Science Centre grant no. UMO-2013/11/D/ST6/03075.

19th International Conference on Database Theory (ICDT 2016).
Editors: Wim Martens and Thomas Zeume; pp. 1–22

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

mappings in data exchange and schema evolution scenarios [1, 12]. Many of these problems are equivalent to satisfiability of fragments of first order logic, possibly over a restricted class of structures, but they are rarely presented this way, because the involved fragments are tailored for specific applications, and usually do not form natural sublogics. As the satisfiability problem for first order logic is undecidable even for relatively simple fragments, in static analysis undecidability is always close [15, 16].

In this paper we present a decidability result (with tight complexity bounds) for a problem in static analysis for tree-structured data. The specific data model we consider is that of data trees: finite ordered unranked trees whose nodes have a label from a finite alphabet and store a data value from an infinite data domain. The problem has three possible formulations:

- consistency modulo schema for a class of integrity constraints;
- validity modulo schema for a class of queries; and
- consistency for a class of schema mappings.

The more general problems of entailment (or implication) of constraints and containment of queries are—as is often the case—very close to their restricted counterparts listed above, and can be solved by easy modifications of our decision procedure.

Our basic setting is that of consistency of integrity constraints; it seems best suited for proofs and—in combination with entailment—the most appealing. We consider *non-mixing constraints* of the forms

$$\alpha(\bar{x}) \Rightarrow \eta_{\sim}(\bar{x}) \qquad \text{and} \qquad \alpha(\bar{x}) \Rightarrow \eta_{\nsim}(\bar{x})$$

that require each tuple $\bar{x}$ of nodes selected by α to satisfy, respectively, a positive combination of equalities $\eta_{\sim}$ or a positive combination of inequalities $\eta_{\nsim}$ over the stored data values. As tuple selectors $\alpha(\bar{x})$ we use conjunctive queries over the signature including label tests and the usual navigational axes.

What is the expressive power of non-mixing constraints? Let us first look at what they cannot do. Being first-order constraints, they cannot compare full subtrees, unlike some other formalisms [17, 18]. They have purely universal character (can be written as universal sentences of first order logic), so they cannot express general inclusion dependencies nor foreign keys, as these need quantifier alternation. Finally, the inability to mix freely data equalities and inequalities within a single constraint makes them unable to express general functional dependencies. What can they do, then?

Non-mixing integrity constraints can be seen as a special case of the general framework of XML-to-relational constraints (X2R constraints) introduced by Niewerth and Schwentick [25]. Within this framework they cover a wide subclass of functional dependencies, dubbed XKFDs, which are particularly well suited for tree-structured data and include W3C XML Schema key and unique constraints [14], as well as absolute and relative XML keys by Arenas, Fan, and Libkin [2], and XFDs by Arenas and Libkin [3]. XKFDs can be expressed with non-mixing constraints of the form $\alpha(\bar{x}) \Rightarrow \eta_{\nsim}(\bar{x})$; that is, using only data inequalities.

Constraints of the form $\alpha(\bar{x}) \Rightarrow \eta_{\sim}(\bar{x})$—that is, using only equalities—can express all sorts of finite data domain restrictions, either to a specific set of constants or to a set of data values taken from the data tree (the latter can be seen as a limited variant of inclusion constraints), as well as cardinality restrictions over data values.

The novelty of our work is that we allow these two kinds of constraints simultaneously. Unrestricted mixing of data equalities and inequalities in constraints would immediately lead to undecidability [6], but for non-mixing constraints we can show decidability of the consistency problem, and a slight extension of the proof gives decidability for entailment (with the same complexity bounds).

Under the second interpretation our result shows decidability of validity and containment for unions of conjunctive queries where each conjunctive query can use either data equality or inequality, but never both. Seen this way, our result is a uniform extension of decidability results for UCQs using only data equality, and UCQs using only data inequality by Björklund, Martens and Schwentick [6] (see also [9]). However, it cannot be obtained via a combination of techniques used in these cases, as they are virtually contradictory: they require assuming that almost all data values in counter-examples are, respectively, different and equal. If data equalities and inequalities are mixed freely in UCQs, even validity is undecidable [6].

In its third incarnation, our result gives decidability of the consistency problem for XML schema mappings with source integrity constraints, which asks to decide if there exists a source instance which satisfies the integrity constraints and admits a target instance satisfying the requirements imposed by the schema mapping.

In all three cases the decision procedure is doubly exponential. This bound is tight, as already validity modulo schema for UCQs over trees without data values is 2ExpTime-complete [6]. We show that restricting the CQs to tree patterns does not help. However, the complexity does drop to ExpTime-complete when we replace CQs with tree patterns and bound the number of variables used in data comparisons.

The remainder of the paper begins with a precise definition of non-mixing constraints and a short discussion of their scope (Section 2). Then we present the decision procedure for consistency of non-mixing constraints (Section 3), followed by a detailed discussion of the entailment problem, the lower-complexity fragment, the relationships with existing constraint formalisms, and the two alternative interpretations of our results (Section 4). We conclude with a brief discussion of further extensions and open questions (Section 5). An appendix containing the missing proofs is available at: `www.mimuw.edu.pl/~fmurlak/papers/concon.pdf`.

2 Non-mixing constraints

Preliminaries

Let us fix a finite labelling alphabet Γ and a countably infinite set of data values $\mathbb{D}$. A *data tree* t is a finite ordered unranked tree whose nodes are labelled with elements of Γ by function $\mathsf{lab}_t : \mathsf{dom}_t \to \Gamma$, and with elements of $\mathbb{D}$ by function $\mathsf{val}_t : \mathsf{dom}_t \to \mathbb{D}$; here, dom_t stands for the *domain* of tree t, that is, the set of its nodes. If $\mathsf{lab}_t(v) = a$ and $\mathsf{val}_t(v) = d$, we say that node v has label a and stores data value d. For a node v of t, we write t_v for the fragment of t consisting of trees rooted at v itself and at all preceding siblings of v. By slight abuse of notation we write $t - t_v$ for the remaining part of t.

We abstract schemas as tree automata in the "previous sibling, last child" variant. A *tree automaton* $\mathcal{A}$ is a tuple (Q, q_0, F, δ), where Q is a finite set of states, $q_0 \in Q$ is an initial state, $F \subseteq Q$ is a set of accepting states, and $\delta \subseteq Q \times Q \times \Gamma \times Q$ is a set of transitions. Being in a node v of the input tree t, the automaton has processed t_v. The state for node v depends on the label of v and the states from the previous sibling and the last child of v. In leftmost siblings and in leaves we resort to imaginary nodes outside of the actual tree t, which are always assigned the initial state q_0. Formally, let dom_t^{cl} be the set containing each node of t, an artificial previous sibling for each leftmost sibling in t, and an artificial (last) child for each leaf in t. A run of $\mathcal{A}$ on t is a function $\rho : \mathsf{dom}_t^{cl} \to Q$ such that $\rho(v) = q_0$ for every node $v \in \mathsf{dom}_t^{cl} - \mathsf{dom}_t$, and for every node $v \in \mathsf{dom}_t$ with previous sibling v_{ps} and last child v_{lc} there is a transition $(\rho(v_{ps}), \rho(v_{lc}), \mathsf{lab}_t(v), \rho(v)) \in \delta$. A run ρ is *accepting* if it assigns a state from F to the root of t, and a tree t is *accepted* by $\mathcal{A}$ if it admits an accepting run.

To facilitate the use of the standard first order semantics, we model data trees as relational

structures over signature

$$\mathsf{sig}_{dt} = \{\downarrow, \downarrow^+, \rightarrow, \rightarrow^+, \sim, \not\sim\} \cup \Gamma \cup \mathbb{D} \cup \widehat{\mathbb{D}}$$

with $\widehat{\mathbb{D}} = \{\hat{d} \mid d \in \mathbb{D}\}$; that is, we have

- binary relations: *child* $\downarrow$, *descendant* $\downarrow^+$, *next sibling* $\rightarrow$, and *following sibling* $\rightarrow^+$;
- data equality relation $\sim$ and data inequality relation $\not\sim$ that contain pairs of nodes storing, respectively, the same data value and different data values;
- unary relation a for each label $a \in \Gamma$;
- unary relations d and $\hat{d}$ for each data value $d \in \mathbb{D}$ that contain nodes storing, respectively, data value d and some data value different from d.

Signature sig_{dt} is infinite (because of $\mathbb{D}$ and $\widehat{\mathbb{D}}$), but queries use only finite fragments.

A *conjunctive query* $\alpha(x_1, \ldots, x_n)$ over a signature sig is a first order formula of the form

$$\exists y_1 \ldots \exists y_m \, \beta(x_1, \ldots, x_n, y_1, \ldots, y_m) \,,$$

where $\beta(x_1, \ldots, x_n, y_1, \ldots, y_m)$ is a conjunction of atoms over signature sig and variables $x_1, \ldots, x_n, y_1, \ldots, y_m$, such that each variable occurs in at least one atom.

Definition

In their most general form, *non-mixing integrity constraints* σ are formulas of the form

$$\alpha(\bar{x}) \Rightarrow \eta_\sim(\bar{x}) \wedge \eta_{\not\sim}(\bar{x})$$

where

- $\alpha(\bar{x})$ is a conjunctive query over signature $\mathsf{sig}_{nav} = \{\downarrow, \downarrow^+, \rightarrow, \rightarrow^+\} \cup \Gamma$;
- $\eta_\sim(\bar{x})$ is a positive Boolean combination of atoms over $\mathsf{sig}_\sim = \{\sim\} \cup \mathbb{D}$ and variables $\bar{x}$;
- $\eta_{\not\sim}(\bar{x})$ is a positive Boolean combination of atoms over $\mathsf{sig}_{\not\sim} = \{\not\sim\} \cup \widehat{\mathbb{D}}$ and variables $\bar{x}$.

Query α is called the *selector* of σ, and $\eta_\sim, \eta_{\not\sim}$ are its *assertions*. Non-mixing constraints have the usual semantics of first order logic formulas: a data tree t satisfies constraint σ, denoted $t \models \sigma$, if each tuple $\bar{v}$ of nodes of t selected by α satisfies both $\eta_\sim$ and $\eta_{\not\sim}$; that is,

$$t \models \alpha(\bar{v}) \quad \text{implies} \quad t \models \eta_\sim(\bar{v}) \wedge \eta_{\not\sim}(\bar{v}) \,.$$

For a set Σ of non-mixing constraints, we write $t \models \Sigma$ if $t \models \sigma$ for all $\sigma \in \Sigma$.

Note that $\alpha \Rightarrow \eta_\sim \wedge \eta_{\not\sim}$ is equivalent to $\{\alpha \Rightarrow \eta_\sim \,,\ \alpha \Rightarrow \eta_{\not\sim}\}$. Consequently, each set Σ of non-mixing constraints is equivalent to $\Sigma_\sim \cup \Sigma_{\not\sim}$, where $\Sigma_\sim$ is a set of constraints of the form $\alpha \Rightarrow \eta_\sim$, $\Sigma_{\not\sim}$ is a set of constraints of the form $\alpha \Rightarrow \eta_{\not\sim}$, and the sizes of $\Sigma_\sim$ and $\Sigma_{\not\sim}$ are bounded by the size of Σ. Thus, without loss of generality, we restrict our attention to sets of constraints of the form $\Sigma_\sim \cup \Sigma_{\not\sim}$, which do not mix $\mathsf{sig}_\sim$ and $\mathsf{sig}_{\not\sim}$ (hence "non-mixing"). One can also assume that α is quantifier free: $\exists \bar{y} \, \alpha(\bar{x}, \bar{y}) \Rightarrow \eta(\bar{x})$ is equivalent to $\alpha(\bar{x}, \bar{y}) \Rightarrow \eta(\bar{x})$.

Scope

Using non-mixing constraints one can express a variety of useful constraints. Let us consider a database storing information about banks, each in a separate sub-document. We want each bank to be identified by its BIC number. This **key constraint** can be expressed as

$$q_{\text{BIC}}(x, x') \wedge q_{\text{BIC}}(y, y') \wedge x \neq y \;\Rightarrow\; x' \not\sim y'$$

where q_{BIC} selects the root of the sub-document for bank, and the node storing the BIC number. Depending on the schema, query q_{BIC} could be for instance $q_{\text{BIC}}(x, x') = \texttt{bank}(x) \wedge x \downarrow$

$x' \wedge \texttt{BIC}(x')$. Node inequality $\neq$ is not part of the signature, but can be expressed using sig_{nav}. Assuming that the roots of the sub-documents for banks are siblings, $x \neq y$ can be replaced by $x \rightarrow^+ y$. In general, we also need to consider four other possible ways in which two different nodes x and y can be positioned in a tree (up to swapping x and y):

$$x \downarrow^+ y\,, \quad x \rightarrow^+ z \wedge z \downarrow^+ y\,, \quad z \downarrow^+ x \wedge z \rightarrow^+ y\,, \quad \text{and} \quad z \downarrow^+ x \wedge z \rightarrow^+ z' \wedge z' \downarrow^+ y\,,$$

which means that we need five non-mixing constraints to express a single key constraint.

Another natural constraint is that account numbers should be different for every account within the same bank, but different banks may use the same account numbers. Such a **relative key constraint** can also be expressed as

$$\texttt{bank}(z) \wedge z \downarrow^+ x \wedge z \downarrow^+ y \wedge q_{\text{ACC}}(x, x') \wedge q_{\text{ACC}}(y, y') \wedge x \neq y \;\Rightarrow\; x' \nsim y'\,.$$

where $q_{\text{ACC}}(x, x')$ selects account x and its number x', similarly to q_{BIC}.

We can also express **multi-attribute keys** (i.e. keys using composite fields). For example

$$q_{\text{BIC}}(u, u') \wedge q_{\text{BIC}}(v, v') \wedge u \downarrow^+ x \wedge v \downarrow^+ y \wedge q_{\text{ACC}}(x, x') \wedge q_{\text{ACC}}(y, y') \wedge x \neq y \;\Rightarrow\; u' \nsim v' \vee x' \nsim y'\,.$$

asserts that BIC and account number form an absolute key, not relative to bank sub-document.

If, as a result of redundancy, BIC appears in several places within a bank sub-document, using the **singleton constraint**

$$\texttt{bank}(x) \wedge x \downarrow^+ x' \wedge \texttt{BIC}(x') \wedge x \downarrow^+ x'' \wedge \texttt{BIC}(x'') \;\Rightarrow\; x' \sim x''$$

we can guarantee that each time it gives the same value (for the same bank).

Assume now that each bank has a director and several branches, each of them having a team of employees among which one is the manager of the branch. The information about each employee is stored in a sub-document of its branch's sub-document. Each employee reports either to the manager of the branch or directly to the director of the bank. Using a conjunctive query $q_{\text{SUPER}}(x, y, z)$, we can select the director's ID node x, the branch manager's ID node y and the node z storing the supervisor's ID for an employee of the same branch. The constraint on employee's supervisor can be encoded as

$$q_{\text{SUPER}}(x, y, z) \;\Rightarrow\; x \sim z \vee y \sim z\,.$$

Following this idea we can express **inclusion constraints of a restricted form**, where the intended superset is a tuple of values that can be selected by a conjunctive query. This includes **enumerative domain restrictions**, like the constraint

$$\texttt{creditCard}(x) \wedge x \downarrow^+ x' \wedge \texttt{brand}(x') \Rightarrow \mathit{Visa}(x') \vee \mathit{MasterCard}(x') \vee \mathit{AmericanExpress}(x')\,,$$

ensuring that banks issue only Visa, Master Card, and American Express cards. Unrestricted inclusion constraints are beyond the scope of our formalism. Indeed, non-mixing constraints cannot be violated by removing nodes, which is not the case even for the simplest unary inclusion constraints, like *each value stored in an a node is also stored in a b node*.

Our formalism is also capable of expressing **cardinality constraints**. Assume, for instance, that banks support charity projects by delegating their employees to help. The projects are organized by category (culture, education, environment, etc.) and each project sub-document carries the list of involved employees. For the sake of balance, we want each category to involve at most ten different employees in total. This can be imposed by selecting eleven employee nodes below a single category node and imposing at least two of them to

carry the same data value by means of a long disjunction of data equalities. We can also ensure that no employee is involved in more than three different projects: the conjunctive query selects four different project nodes and an employee for each of them; the assertion imposes at least two of the four employees to have different ID.

Let us remark that while these constraints look clumsy expressed as non-mixing constraints, one can easily imagine a syntactic-sugar layer on top of our formalism. The point is that all these constraints can be rewritten as non-mixing constraints of linear size (except for the cardinality constraints, where the numerical bounds would be typically given in decimal).

In Section 4 we examine the expressive power of non-mixing constraints further by comparing them to other existing formalisms.

3 Consistency problem

Our main result is decidability of the consistency problem for non-mixing constraints:

PROBLEM:	**Consistency of non-mixing constraints**
INPUT:	A set Σ of non-mixing constraints, a tree automaton $\mathcal{A}$.
QUESTION:	Is there a data tree such that $t \in L(\mathcal{A})$ and $t \models \Sigma$?

More precisely, we show the following theorem, establishing tight complexity bounds.

▶ **Theorem 1.** *Consistency of non-mixing constraints is* 2EXPTIME*-complete.*

The proof of Theorem 1 is based on a simple idea with a geometric flavour, but does not require any specialist knowledge from geometry or linear algebra. The key fact is an upper bound on the number of affine subspaces that constitute an intersection of unions of affine subspaces of an Euclidean space; it has a short elementary proof. From this bound we infer a "bounded data cut" model property for non-mixing constraints, where by *data cut* of a data tree t, denoted by $datacut(t)$, we mean the maximum over nodes $v \in \mathsf{dom}_t$ of the number of data values shared by t_v and $t - t_v$.

A *subspace* of $\mathbb{D}^\ell$ is a subset of $\mathbb{D}^\ell$ defined by equating pairs of coordinates and fixing coordinates; that is, it is a set of points $(x_1, x_2, \ldots, x_\ell)$ in space $\mathbb{D}^\ell$ defined by a conjunction of equalities of the form $x_i = x_j$ or $x_i = d$ where $d \in \mathbb{D}$. Each nonempty subspace of $\mathbb{D}^\ell$ can be defined by a *canonical* set of at most ℓ equalities such that

- for each coordinate i we have either $x_i = x_j$ with $i < j$, or $x_i = d$ with $d \in \mathbb{D}$, or nothing;
- each coordinate j occurs at most once on the right side of an equality; and
- no data value d is used in more than one equality.

A subspace of $\mathbb{D}^\ell$ has *dimension* k if its canonical definition consists of $\ell - k$ equalities. In other words, each equality that does not follow from the others decreases the dimension by one. To enhance intuitions, let us remark that if we equip $\mathbb{D}^\ell$ with the structure of linear space by assuming that $\mathbb{D}$ is a field, this notion of dimension coincides with the classical notion of dimension for affine subspaces (of which the subspaces above are a special case).

An intersection $X \cap Y$ of subspaces X, Y is also a subspace, defined by the conjunction of conditions defining X and Y. If $X \not\subseteq Y$, then the canonical definition of $X \cap Y$ contains at least one more equation, consequently, the dimension of $X \cap Y$ is strictly smaller than the dimension of X. Similarly, intersecting unions of subspaces, we obtain a union of subspaces; the following lemma gives a bound on the size of such union.

▶ **Lemma 2.** *Let $Z_1, Z_2, \ldots, Z_m \subseteq \mathbb{D}^\ell$ be such that each Z_i is a union of at most n subspaces of $\mathbb{D}^\ell$. Then, $Z_1 \cap Z_2 \cap \cdots \cap Z_m$ is a union of at most n^ℓ subspaces of $\mathbb{D}^\ell$.*

Proof. Assume that $Z_1 \cap Z_2 \cap \cdots \cap Z_{i-1}$ is a union $X_1 \cup X_2 \cup \cdots \cup X_p$ of subspaces of $\mathbb{D}^\ell$. We can write Z_i as $Y_1 \cup Y_2 \cup \cdots \cup Y_n$, where some of subspaces Y_k may be empty. We have

$$Z_1 \cap Z_2 \cap \cdots \cap Z_i = (X_1 \cup X_2 \cup \cdots \cup X_p) \cap Z_i = (X_1 \cap Z_i) \cup (X_2 \cap Z_i) \cup \cdots \cup (X_p \cap Z_i)\,.$$

Let us examine a single $X_j \cap Z_i$. If $X_j \subseteq Y_k$ for some k, then $X_j \cap Z_i = X_j$. Otherwise, $X_j \cap Z_i$ is a union of n subspaces, $X_j \cap Y_1, X_j \cap Y_2, \ldots, X_j \cap Y_n$, where each $X_j \cap Y_k$ is either empty or has dimension strictly smaller than X_j. Thus, when $X_1 \cup X_2 \cup \cdots \cup X_p$ is intersected with Z_i, each X_j either does not change, or falls apart into at most n subspaces of strictly smaller dimension; if X_j is a point, in the second possibility it disappears.

Now, consider the following process: begin with $\mathbb{D}^\ell$, a single subspace of dimension ℓ, and then intersect with Z_i for i from 1 to m, one by one. It follows immediately from the observation above that we cannot obtain more than n^ℓ subspaces in this process. ◀

We remark that the bound in Lemma 2 is tight, as shown by the following example.

▶ **Example 3.** [1] Assume $0, 1 \in \mathbb{D}$ and let $Z_i = \left\{\bar{x} \subseteq \mathbb{D}^\ell \;\middle|\; x_i = 0 \vee x_i = 1\right\}$ for $i = 1, 2, \ldots, \ell$. Then $Z_1 \cap Z_2 \cap \cdots \cap Z_\ell = \{0,1\}^\ell$ is a union of 2^ℓ (disjoint) subspaces of $\mathbb{D}^\ell$ of dimension 0.

Based on this geometric fact, in Lemma 5 we bound the data cut of data trees witnessing consistency of non-mixing constraints. The proof relies on a simple compositionality property for conjunctive queries over trees, stated in Lemma 4 (see Appendix A for proof).

▶ **Lemma 4.** *Let $\alpha(\bar{x}, \bar{y})$ be a conjunction of atoms over sig_{nav}, where $\bar{x}$ and $\bar{y}$ are disjoint, and let w be a node of a data tree t. For all tuples $\bar{u}, \bar{u}'$ of nodes from t_w and tuples $\bar{v}, \bar{v}'$ of nodes from $t - t_w$, if $t \models \alpha(\bar{u}, \bar{v})$ and $t \models \alpha(\bar{u}', \bar{v}')$, then $t \models \alpha(\bar{u}, \bar{v}')$ and $t \models \alpha(\bar{u}', \bar{v})$.*

▶ **Lemma 5.** *If $\Sigma_\sim \cup \Sigma_{\not\sim}$ is satisfied in a data tree t, it is also satisfied in some data tree t' obtained from t by changing data values, such that $\mathrm{datacut}(t') \leq \ell \cdot 2^\ell \cdot (\ell + m)^{\ell^2} \cdot |\Sigma_\sim|$, where ℓ, m are the maximal numbers of variables and predicates from $\mathbb{D} \cup \hat{\mathbb{D}}$ in constraints of $\Sigma_\sim$.*

Proof. Assume that $t \models \Sigma_\sim \cup \Sigma_{\not\sim}$ and let w be a node of the data tree t. We shall replace all but $\ell \cdot 2^\ell \cdot (\ell + m)^{\ell^2} \cdot |\Sigma_\sim|$ data values used in t_w with distinct fresh data values, thus ensuring that in the resulting tree the number of data values used both in t_w and $t - t_w$ is bounded by $\ell \cdot 2^\ell \cdot (\ell + m)^{\ell^2} \cdot |\Sigma_\sim|$. As the fresh data values are be distinct, the new $\sim$ relation over nodes of t is a subset of the old one. In consequence, the operation does not increase the number of data values shared by $t_{w'}$ and $t - t_{w'}$ for other nodes w'. For the same reason, the obtained tree still satisfies $\Sigma_{\not\sim}$. We only need to ensure that $\Sigma_\sim$ is not violated.

Consider a constraint $\alpha \Rightarrow \eta_\sim$ in $\Sigma_\sim$. Recall that we assume that α is quantifier free. Let $\bar{x}$, $\bar{y}$ be a partition of variables used in α (one of the tuples $\bar{x}$, $\bar{y}$ may be empty). We shall indicate the partition of variables by writing the constraint as $\alpha(\bar{x}, \bar{y}) \Rightarrow \eta_\sim(\bar{x}, \bar{y})$. Directly from the definition it follows that $t \models \alpha \Rightarrow \eta_\sim$, if and only if for each partition $\bar{x}, \bar{y}$ of variables in α, for each tuple $\bar{u}$ of nodes from t_w and each tuple $\bar{v}$ of nodes from $t - t_w$, if $t \models \alpha(\bar{u}, \bar{v})$, then $t \models \eta_\sim(\bar{u}, \bar{v})$.

Fix a partition $\bar{x}$, $\bar{y}$. By Lemma 4, this is equivalent to: for all tuples $\bar{u}, \bar{u}'$ of nodes from t_w and all tuples $\bar{v}, \bar{v}'$ of nodes from $t - t_w$, if $t \models \alpha(\bar{u}, \bar{v})$ and $t \models \alpha(\bar{u}', \bar{v}')$, then $t \models \eta_\sim(\bar{u}, \bar{v}')$.

Let us turn this into a condition on stored data values. Define $\eta(\bar{x}, \bar{y})$ as the formula obtained from $\eta_\sim(\bar{x}, \bar{y})$ by replacing $\sim$ with $=$, and $d(x)$ with $x = d$ for all variables x and all $d \in \mathbb{D}$. Reformulating the condition above we obtain: for each tuple $\bar{u}$ of nodes from t_w

[1] Provided by Michał Pilipczuk, during the Warsaw Automata Group's research camp *Autobóz 2015*.

such that $t \models \alpha(\bar{u}, \bar{v})$ for some tuple $\bar{v}$ of nodes from $t - t_w$, the tuple $\mathsf{val}_t(\bar{u})$ of data values belongs to the set

$$Z_{\alpha(\bar{x},\bar{y})\Rightarrow\eta_\sim(\bar{x},\bar{y})} = \bigcap_{\bar{v}'} \left\{ \bar{c} \in \mathbb{D}^{|\bar{x}|} \;\middle|\; \eta(\bar{c}, \mathsf{val}_t(\bar{v}')) \right\},$$

where $\bar{v}'$ ranges over tuples of nodes from $t - t_w$ satisfying $t \models \alpha(\bar{u}', \bar{v}')$ for some tuple $\bar{u}'$ of nodes from t_w.

Let $\mathsf{val}' : \mathsf{dom}_t \to \mathbb{D}$ be a new data labelling of t. Since we are only planning to change data values stored in nodes of t_w, the labelling val' does not violate $\Sigma_\sim$ if and only if for each constraint $\alpha(\bar{x}, \bar{y}) \Rightarrow \eta_\sim(\bar{x}, \bar{y})$ in $\Sigma_\sim$ (with each partition of variables), for each tuple $\bar{u}$ of nodes from t_w that satisfies $t \models \alpha(\bar{u}, \bar{v})$ for some tuple $\bar{v}$ of nodes from $t - t_w$, the tuple $\mathsf{val}'(\bar{u})$ belongs to the set $Z_{\alpha(\bar{x},\bar{y})\Rightarrow\eta_\sim(\bar{x},\bar{z})}$. Writing $\eta(\bar{x}, \mathsf{val}_t(\bar{v}'))$ in the disjunctive normal form, we see that the set $\left\{ \bar{c} \in \mathbb{D}^{|\bar{x}|} \;\middle|\; \eta(\bar{c}, \mathsf{val}_t(\bar{v}')) \right\}$ is a union of subspaces of $\mathbb{D}^{|\bar{x}|}$. How many subspaces? The canonical definition of each nonempty subspace has for each coordinate i either an equality $x_i = x_j$ for some $j > i$, or an equality $x_i = d$ for some $d \in \mathbb{D}$, or nothing. In our case, d is a data value used explicitly in η or occurring in the data tuple $\mathsf{val}_t(\bar{v}')$. Consequently, the number of these subspaces can be bounded by $(N + |\bar{x}| + |\bar{y}|)^{|\bar{x}|}$, where N is the number of data values used explicitly in η. That is, $Z_{\alpha(\bar{x},\bar{y})\Rightarrow\eta_\sim(\bar{x},\bar{y})}$ is an intersection of unions of at most $(N + |\bar{x}| + |\bar{y}|)^{|\bar{x}|}$ subspaces of $\mathbb{D}^{|\bar{x}|}$. By Lemma 2, it is a union of at most $(N + |\bar{x}| + |\bar{y}|)^{|\bar{x}|^2}$ subspaces. Let us analyse the restrictions it puts on val'. We have declared that we shall only replace some data values in t_w with fresh data values. That is, equalities of the form $x_i = x_j$ in the definitions of those subspaces will not get violated. It remains to check that equalities of the form $x_i = d$ are not violated. Each subspace involves at most $|\bar{x}|$ such equalities, so for set $Z_{\alpha(\bar{x},\bar{y})\Rightarrow\eta_\sim(\bar{x},\bar{y})}$ we have at most $|\bar{x}| \cdot (N + |\bar{x}| + |\bar{y}|)^{|\bar{x}|^2}$ of them. Let $D \subseteq \mathbb{D}$ be the set of data values occurring in these equalities for all sets $Z_{\alpha(\bar{x},\bar{y})\Rightarrow\eta_\sim(\bar{x},\bar{y})}$, with $\alpha(\bar{x}, \bar{y}) \Rightarrow \eta_\sim(\bar{x}, \bar{y})$ ranging over constraints from $\Sigma_\sim$ with all possible partitions of variables. We have $|D| \leq |\Sigma_\sim| \cdot 2^\ell \cdot \left(\ell \cdot (m + \ell)^{\ell^2}\right)$, where ℓ and m are the maximal numbers of variables and predicates from $\mathbb{D} \cup \hat{\mathbb{D}}$ in constraints from $\Sigma_\sim$. Altogether, a labelling val' that replaces each data value from $\mathbb{D} - D$ used in t_w with a fresh data value does not violate $\Sigma_\sim$. ◀

Having obtained Lemma 5, we are ready to prove Theorem 1. In the proof we use register tree automata. These are tree automata, equipped additionally with a finite number of registers. The registers can store data values (from $\mathbb{D}$) read from the data tree or just guessed, and compare them with data values seen later in the tree. Formally, a *register tree automaton* $\mathcal{A}$ is a tuple (Q, q_0, F, k, δ), where Q is a finite set of states, $q_0 \in Q$ is an initial state, $F \subseteq Q$ is a set of accepting states, k is the number of registers, and δ is a set of transitions. A transition in δ is of the form $(q_{ps}, q_{lc}, a, q_{cur}, E)$, where $q_{ps}, q_{lc}, q_{cur} \in Q$, $a \in \Gamma$, and E is an equivalence relation over $\{val\} \cup \left(\{ps, lc, cur\} \times \{1, \ldots, k\}\right)$ in which each equivalence class contains at most one element from $\{l\} \times \{1, \ldots, k\}$ for each $l \in \{ps, lc, cur\}$.

Next, we define how $\mathcal{A}$ runs on a data tree t. Similarly to a standard tree automaton, to compute the state and the register valuation in some node, automaton $\mathcal{A}$ uses the state and the register valuation from the previous sibling and the last child of that node. The equivalence relation E of the transition specifies which data values are equal: val denotes the data value stored in the tree node, (ps, i) denotes the data value assigned to the i-th register in the previous sibling, and similarly for (lc, i) and (cur, i). We do not specify initial register valuations: the register valuation in an imaginary previous sibling or last child may be arbitrary. Formally, a run $\rho = (\rho_Q, \rho_\mathbb{D})$ of $\mathcal{A}$ on t is given by functions $\rho_Q : \mathsf{dom}_t^{cl} \to Q$ and $\rho_\mathbb{D} : \mathsf{dom}_t^{cl} \times \{1, \ldots, k\} \to \mathbb{D}$ such that for every node $v \in \mathsf{dom}_t^{cl} - \mathsf{dom}_t$ it holds that $\rho_Q(v) = q_0$, and for every node $v \in \mathsf{dom}_t$ with previous sibling v_{ps} and last child v_{lc} there

is a transition $(\rho_q(v_{ps}), \rho_q(v_{lc}), \mathsf{lab}_t(v), \rho_q(v), E) \in \delta$ with $E = \{(a,b) \mid as(a) = as(b)\}$ where $as(val) = \mathsf{val}_t(v)$, $as(ps,i) = \rho_{\mathbb{D}}(v_{ps}, i)$, $as(lc,i) = \rho_{\mathbb{D}}(v_{lc}, i)$, $as(cur,i) = \rho_{\mathbb{D}}(v,i)$. Run ρ is accepting if ρ_Q assigns a state from F to the root of t.

Our automata cannot store the same value in more than one register, like in the seminal paper by Kaminski and Francez [18]; this guarantees polynomial-time emptiness test.

▶ **Theorem 6.** *It is decidable in polynomial time whether a given register tree automaton accepts at least one data tree.*

Kaminski and Francez [19] introduced register automata for words; the tree variant was proposed by Kaminski and Tan [20]. The emptiness problem was already considered in [20], but since our automaton model is slightly different, we give a sketch of the proof in Appendix B.

To prove Theorem 1, we reduce consistency of non-mixing constraints to emptiness of register tree automata of doubly exponential size. The register automaton we construct computes a representation of tuples selected from the input data tree by the selector queries of $\Sigma_\sim \cup \Sigma_{\not\sim}$. To explain how this can be done, we need some auxiliary notions.

A *partial valuation* of variables $x_1, \ldots, x_k$ is a function

$$f \colon \{x_1, x_2, \ldots, x_k\} \to \mathsf{dom}_t \cup \{\bot\}\,.$$

If $f(x_i) \neq \bot$, we say that x_i is *matched* at $f(x_i)$, and if $u_i = \bot$ we say that x_i is *not matched*. Two partial valuations of the same set of variables are *disjoint*, if no variable is matched by both of them. The *union* of disjoint partial valuations f, g of variables $x_1, \ldots, x_k$ is given as

$$(f \cup g)(x_i) = \begin{cases} f(x_i) & \text{if } f(x_i) \neq \bot \\ g(x_i) & \text{otherwise} \end{cases}$$

A *partial matching* of $\alpha(\bar{x})$ in t_w is a partial valuation f of variables $\bar{x}$ such that variables are matched only in the nodes of t_w, each atom in $\alpha(f(\bar{x}))$ that does not contain $\bot$ holds true, and each atom that contains both a node from t_w and $\bot$ is of the form

$$w \to \bot, \quad w' \to^+ \bot, \quad \bot \downarrow w', \quad \text{or} \quad \bot \downarrow^+ v\,,$$

where w' is a preceding sibling of w or w itself, and v is an arbitrary node of t_w. The last condition means that each such atom can be made true (independently of others) by replacing $\bot$ with a node from $t - t_w$, unless w has no following siblings or no ancestors in t.

If $t \models \alpha(\bar{u})$, each partial valuation matching a subset of variables x_i at nodes u_i from t_w is a partial matching of α. Conversely, if a partial matching f matches all variables $\bar{x}$, then $t \models \alpha(f(\bar{x}))$. Note, however, that not every partial matching can be extended so that it matches all variables: remaining atoms may be satisfiable on their own, but not together.

Proof of Theorem 1. Let $\Sigma_\sim \cup \Sigma_{\not\sim}$ be a set of non-mixing integrity constraints and let $\mathcal{A}$ be a tree automaton. We shall construct a register tree automaton $\mathcal{B}$ such that $\mathcal{B}$ accepts at least one data tree if and only if some tree accepted by $\mathcal{A}$ satisfies $\Sigma_\sim \cup \Sigma_{\not\sim}$. Automaton $\mathcal{B}$ will be the product of automaton $\mathcal{A}$ and a register tree automaton $\mathcal{C}$ of doubly exponential size. We would like automaton $\mathcal{C}$ to check if the input data tree satisfies $\Sigma_\sim \cup \Sigma_{\not\sim}$, but this does not seem possible. Instead, automaton $\mathcal{C}$ will accept data trees that satisfy $\Sigma_\sim \cup \Sigma_{\not\sim}$ up to a relabelling of data values. This is sufficient to guarantee correctness of the reduction.

For all tuples $\bar{u}$ such that $t \models \alpha(\bar{u})$, automaton $\mathcal{C}$ should check that the corresponding assertion $\eta(\bar{u})$ holds. For that it suffices to know the equalities between data values stored in nodes $\bar{u}$ and the ones used explicitly in η. The information about these equalities is collected

node by node: when automaton $\mathcal{C}$ is in a node w of input data tree t, it has processed t_w, and $t - t_w$ remains to be processed. To compute these equalities we need to keep track of data values used in t_w that will also occur in $t - t_w$. We use registers to store them. More precisely, automaton $\mathcal{C}$ stores in registers M data values used explicitly in $\Sigma_\sim \cup \Sigma_{\not\sim}$ (this is the only way it can perform comparisons with them), and some data values shared by t_w and $t - t_w$. Number M is bounded by the size of the input. The number of values shared by t_w and $t - t_w$ is in general unbounded, but for a tree of optimal data cut we can store all of them, as, by Lemma 5, their number is bounded by some N, singly exponential in the size of the input. These two sets of values may overlap, but $M + N$ registers are enough to store them. Automaton $\mathcal{C}$ cannot determine which data values are used in $\Sigma_\sim \cup \Sigma_{\not\sim}$, nor which data values seen in t_w will be used again in $t - t_w$, so it uses nondeterminism to fill in the registers. The first M registers are guessed in each initial configuration and never changed; the remaining ones are updated (nondeterministically) in each processed node w.

In states, automaton $\mathcal{C}$ remembers for each selector $\alpha(\bar{x})$ a subset $\Delta_{\alpha(\bar{x})}$ of

$$\{\mathsf{r}_1, \mathsf{r}_2, \ldots, \mathsf{r}_{M+N}, \top_1, \top_2, \ldots, \top_\ell, \bot\}^{|\bar{x}|},$$

where ℓ is the maximal number of variables used in constraints in $\Sigma_\sim \cup \Sigma_{\not\sim}$. Each such tuple represents a partial matching of $\alpha(\bar{x})$ in t_w, and the whole $\Delta_{\alpha(\bar{x})}$ represents a set of such partial matchings. The intended meaning of the symbolic values is as follows: r_i in coordinate j of the tuple means that variable x_j is matched and its data value is stored in the i-th register, $\top_i$ means that variable x_j is matched but we do not store the corresponding data value and consider it different from all others, $\bot$ means that variable x_j has not been matched. In the initial state, each $\Delta_{\alpha(\bar{x})}$ is $\{(\bot, \bot, \ldots, \bot)\}$. When some $\Delta_{\alpha(\bar{x})}$ contains a tuple that does not use $\bot$ and does not satisfy the corresponding assertion $\eta(\bar{x})$, automaton $\mathcal{C}$ rejects the input tree t immediately.

Let us describe the transition relation. Assume that automaton $\mathcal{C}$ is about to determine the state in a node w. Let w' and w'' be, respectively, the previous sibling and the last child of w. The set of partial matchings of $\alpha(\bar{x})$ in t_w depends only on the sets of partial matchings in $t_{w'}$ and $t_{w''}$, and the label of w. Indeed, a partial valuation of $\bar{x}$ is a partial matching of $\alpha(\bar{x})$ in t_w if it is the union of disjoint partial matchings of $\alpha(\bar{x})$ in $t_{w'}$ and $t_{w''}$ extended by matching some (yet unmatched) variables at node w, respecting two conditions. For all atoms $x_i \to x_j$, $x_i \to^+ x_j$ in $\alpha(\bar{x})$, either x_i, x_j are both matched in $t_{w''}$ or none is; and the new matching of variables at w does not violate the definition of partial matching. The latter can be expressed as follows:

- if $\alpha(\bar{x})$ contains $x_i \downarrow x_j$ or $x_i \downarrow^+ x_j$, we may match x_i at w only if x_j is matched in $t_{w''}$; for $x_i \downarrow x_j$, if x_j is matched, we must match x_i, unless it is matched already;
- if $\alpha(\bar{x})$ contains $x_i \to x_j$ or $x_i \to^+ x_j$, we may match x_j at w only if x_i is matched in $t_{w'}$; for $x_i \to x_j$, if x_i is matched, we must match x_j, unless it is matched already;
- if $\alpha(\bar{x})$ contains $a(x_i)$, we may match x_i at w only if $\mathsf{lab}_t(w) = a$.

Checking the conditions above requires only information about which variables are matched in $t_{w'}$ and $t_{w''}$; the used tree nodes are not relevant. Consequently, one can determine the set of tuples representing partial matchings in t_w based on the sets of tuples representing partial matchings in $t_{w'}$ and $t_{w''}$, and the label of the current node w. In the resulting representation we use r'_i and r''_i for the copies of the registers from the previous sibling w' and the last child w'', and curr as a symbolic representation of the data value in the current node w. Next, we update the content of the registers. Registers $1, 2, \ldots, M$ must store the same values in both branches of the run, and they are never modified. Registers $M + 1, M + 2, \ldots, M + N$ are filled with distinct values chosen nondeterministically. Some of these values may be copied

from registers $M+1, M+2, \ldots, M+N$ in one of the two branches of the run, or from the current node w. After updating the content of registers, we must reflect the changes in sets $\Delta_{\alpha(\bar{x})}$. If the new value in the i-th register is equal to the one in the j-th register in node w' for some i, j, all occurrences of r'_j are replaced by r_i; analogously for node w''. Similarly, if the value in the i-th register is equal to $\mathsf{val}_t(w)$, we replace curr with r_i. At this point, all remaining symbols r'_j, r''_k, and curr represent data values that are not expected to be seen again. For each such symbol, in each tuple separately, we replace all its occurrences with a distinct unused element of set $\{\top_1, \top_2, \ldots, \top_\ell\}$; there are always enough elements available.

Let us see that automaton $\mathcal{C}$ is correct. By construction, it accepts each data tree satisfying $\Sigma_\sim \cup \Sigma_{\not\sim}$, whose data cut is at most N: initially it guesses correctly the M data values used in $\Sigma_\sim \cup \Sigma_{\not\sim}$, and then in each processed node w it guess a content for the remaining N registers such that all data value shared by t_w and $t - t_w$ are stored. Lemma 5 guarantees that if $\Sigma_\sim \cup \Sigma_{\not\sim}$ is satisfiable, it is satisfiable in a data tree of data cut at most N.

It remains to see that if $\mathcal{C}$ accepts a data tree t, then by modifying data values in t we can obtain a data tree satisfying $\Sigma_\sim \cup \Sigma_{\not\sim}$. Observe first that each data tree accepted by $\mathcal{C}$ satisfies $\Sigma_\sim$, modulo a relabelling of data values (there is no guarantee that automaton $\mathcal{C}$ correctly guesses the values used in $\Sigma_\sim \cup \Sigma_{\not\sim}$, but since registers always store different values, each permutation of $\mathbb{D}$ that sends the content of the i-th register to the i-th data value used in $\Sigma_\sim \cup \Sigma_{\not\sim}$ fixes it). Indeed, in each partial matching represented by a given tuple, actual data values represented in the tuple by the same symbol are always equal. So, when an assertion $\eta_\sim$ holds for the representing tuple, it also holds for each represented matching. The same is not ensured for assertions $\eta_{\not\sim}$. Indeed, when automaton $\mathcal{C}$ sees a data value that is not in the registers, it assumes that it is different from all data values seen before and now represented by $\top_1, \top_2, \ldots, \top_\ell$. But in fact, this value might have been seen, and later forgotten and replaced with a value $\top_i$ in the tuples representing partial matchings. Consequently, in some represented matchings, the same data value may be represented with two (or more) different symbols. We shall show how to modify the data values in t to obtain a data tree, also accepted by automaton $\mathcal{C}$, for which data values represented with different symbols are always different.

Let ρ be an accepting run of automaton $\mathcal{C}$ on data tree t. For each node w and each data value d that is not stored in the registers after processing node w, replace all occurrences of d in t_w and in the corresponding part ρ_w of run ρ with a fresh data value. The resulting run ρ' is an accepting run of automaton $\mathcal{C}$ on the resulting tree t'. In data tree t', two nodes store the same data value if and only if they stored the same value in t and on the shortest path connecting these nodes this data value was always kept in a register in run ρ. Applying a permutation of $\mathbb{D}$ if necessary, we may assume that the values stored in registers $1, 2, \ldots, M$ are indeed the data values used in $\Sigma_\sim \cup \Sigma_{\not\sim}$. Hence, as we have argued, $t' \models \Sigma_\sim$ because automaton $\mathcal{C}$ accepts t'. Let us check that $t' \models \Sigma_{\not\sim}$. If a data value $\mathsf{val}_t(w)$, represented symbolically by curr, is already stored in some register r_i during run ρ, then by definition of register tree automaton it is identified with r_i. Symbol curr is not identified with any r_i only if $\mathsf{val}_t(w)$ is stored in none of the registers. By construction of t', $\mathsf{val}_{t'}(w)$ does not occur in t'_w, and it is correct to represent it with a distinct symbolic value (which is also done in run ρ'). Thus, in each partial matching represented by a given tuple, data values represented by different symbols are indeed different, and if assertion $\eta_{\not\sim}$ holds for the representing tuple, it also holds for each represented matching. This concludes the proof that $t' \models \Sigma_\sim \cup \Sigma_{\not\sim}$.

Since number $M+N$ is singly exponential, so is the number of tuples representing partial matchings. Each state of automaton $\mathcal{C}$ stores a set of such tuples for each constraint, so $\mathcal{C}$ is doubly exponential; it can be also constructed in doubly exponential time. By Theorem 6,

we test emptiness of the product of tree automaton $\mathcal{A}$ and register tree automaton $\mathcal{C}$ in time polynomial in the size of the product. This amounts to a 2ExpTime algorithm.

The matching lower bound can be shown via a reduction from the acceptance problem for alternating Turing machines using exponential space, already for constraints that use tree patterns as selectors and assertions over $\mathsf{sig}_\sim$ only; see Appendix C for details. ◀

4 Extensions, connections, and applications

Entailment of non-mixing constraints

A static analysis problem more general than consistency is entailment. Recall that a set of constraints Σ' is *entailed* by a set of constraints Σ modulo a tree automaton $\mathcal{A}$, written as $\Sigma \models_{\mathcal{A}} \Sigma'$, if for each data tree t accepted by automaton $\mathcal{A}$,

$$t \models \Sigma \quad \text{implies} \quad t \models \Sigma' .$$

The entailment problem is then defined as follows:

Problem:	**Entailment problem for non-mixing constraints**
Input:	sets Σ, Σ' of non-mixing constraints, tree automaton $\mathcal{A}$
Question:	$\Sigma \models_{\mathcal{A}} \Sigma'$?

Entailment is a more general problem than consistency, but for non-mixing constraints the results on consistency generalize to entailment almost effortlessly.

▶ **Theorem 7.** *Entailment of non-mixing constraints is* 2ExpTime*-complete.*

Proof. Inconsistency is a special case of entailment: Σ is inconsistent with respect to an automaton $\mathcal{A}$ if and only if $\Sigma \models_{\mathcal{A}} \bot$, where $\bot$ is an inconsistent set of constraints, say $\{a(x) \Rightarrow 0(x) \wedge 1(x) \mid a \in \Gamma\}$. Thus, the lower bound follows.

Lemma 5 shows that witnesses for consistency can have bounded data cut. The same is true for counter-examples to entailment. Suppose $t \models \Sigma$ and $t \not\models \Sigma'$. Then, $t \models \alpha'(\bar{u}) \wedge \neg\eta'(\bar{u})$ for some constraint $\alpha'(\bar{x}) \Rightarrow \eta'(\bar{x})$ from Σ' and some tuple $\bar{u}$ of nodes of t. Let D_0 be the set of data values used in the nodes $\bar{u}$. We can repeat the construction of t' word for word, except that we replace the set D of values not to be touched by $D \cup D_0$. This increases $\mathit{datacut}(t')$ by the maximal number of variables in the constraints of Σ'.

The automata construction in the proof of Theorem 1 is modified similarly. It is enough to consider Σ' consisting of a single constraint $\alpha'(\bar{x}) \Rightarrow \eta'(\bar{x})$. Let M' be the number of data values used explicitly in Σ'. The number of registers is increased by $M' + |\bar{x}|$, to accommodate these data values, and the ones stored in the nodes $\bar{u}$ that witness satisfaction of $\alpha'(\bar{x}) \wedge \neg\eta'(\bar{x})$ in t. All these values are guessed by the automaton in the beginning of the run, and never modified. In the states the automaton additionally stores a set $\Delta_{\alpha'(\bar{x})}$ of tuples representing partial matchings of $\alpha'(\bar{x})$ such that the matched nodes store the initially guessed values (symbolic values $\top_i$ are not used in $\Delta_{\alpha'(\bar{x})}$). The automaton behaves like before, additionally checking that $\Delta_{\alpha'(\bar{x})}$ contains a tuple without $\bot$ that satisfies $\neg\eta'(\bar{x})$.

The argument does not change if $\eta'(\bar{x})$ mixes predicates from $\mathsf{sig}_\sim$ and $\mathsf{sig}_\nsim$. ◀

A singly exponential fragment

A closer look at the complexity of our algorithm reveals that it is double exponential only in the maximal number ℓ of variables in the constraints, which appears in two roles: in the

exponent of the bound on the number of data values stored in registers, and as the length of tuples representing partial matchings of selectors. A slightly more detailed analysis of the proof of Lemma 5 shows that in the first role ℓ could be replaced by the maximal number ℓ' of variables used in the assertions. Indeed, since data equalities involve only variables used in assertions, everything is in fact happening in a space of dimension at most ℓ'. While limiting the size of selector queries to lower complexity makes little sense, limiting the number of variables in assertions seems acceptable. But what about the second role of ℓ?

The need to represent all partial matchings (up to data equality type) comes from the fact that the automaton is evaluating conjunctive queries all over the tree. The standard technique to lower complexity in such cases is to replace conjunctive queries with tree patterns, which are essentially tree-structured conjunctive queries. In the most basic form, with only $\downarrow$ and $\downarrow^+$ axes allowed, a *tree pattern* is a conjunctive query π over signature $\{\downarrow,\downarrow^+\}\cup\Gamma$, such that graph $(A_\pi,\downarrow_\pi\cup\downarrow^+_\pi)$ is a directed tree, where $\mathbb{A}_\pi=(A_\pi,\downarrow_\pi,\downarrow^+_\pi,\{a_\pi\}_{a\in\Gamma})$ is the canonical relational structure associated to query π in the usual way: the universe A_π is the set of variables of π, and relations are given by the respective atoms in π. For non-mixing integrity constraints, restricting selectors to tree patterns does not suffice to lower the complexity: the reduction in Appendix C uses only such constraints (and no assertions over $\mathsf{sig}_{\nsim}$). But together with the bound on the number of variables in assertions—it does (see Appendix D).

▶ **Proposition 8.** *For non-mixing constraints whose selectors are tree patterns and whose assertions use constantly many variables, consistency and entailment are* ExpTime*-complete.*

Static analysis of unions of conjunctive queries

Our results can be reinterpreted in the framework of static analysis of unions of conjunctive queries (UCQs). Note that $t\not\models\alpha(\bar x)\Rightarrow\eta(\bar x)$ if and only if $t\models\exists\bar x\,\alpha(\bar x)\wedge\neg\eta(\bar x)$. It follows immediately that the problem of validity of UCQs over signature sig_{dt} that never mix predicates from $\mathsf{sig}_\sim$ and $\mathsf{sig}_{\nsim}$—call them *non-mixing UCQs*—reduces in polynomial time to *inconsistency* of non-mixing constraints. Similarly, containment of such queries reduces to entailment of non-mixing constraints. The converse reduction is also possible, but it involves exponential blow-up, caused by rewriting arbitrary Boolean combinations in disjunctive normal form. This correspondence brings our results very close to the work by Björklund, Martens, and Schwentick on static analysis for UCQs over signature $\mathsf{sig}_{nav}\cup\{\sim,\nsim\}$ [6].

On one hand, our results immediately give the following decidability result for the setting considered by Björklund, Martens, and Schwentick (constraints used in the lower bound of Proposition 11 can be rewritten without blow-up).

▶ **Theorem 9.** *Over $\mathsf{sig}_{nav}\cup\{\sim,\nsim\}$, both validity of non-mixing UCQs and containment of UCQs in non-mixing UCQs (with respect to a given tree automaton) are* 2ExpTime*-complete.*

Results of Björklund, Martens, and Schwentick give 2ExpTime upper bound for containment in UCQs over $\mathsf{sig}_{nav}\cup\{\sim\}$ and UCQs over $\mathsf{sig}_{nav}\cup\{\nsim\}$. The original work is on CQs, but arguments for UCQs are the same [9]. Essentially, they amount to an observation that in counter-examples to containment $p\subseteq q$, all data values can be set equal (in the case with $\nsim$) or different (in the case with $\sim$), except for a bounded number of them needed to witness satisfaction of p; such counter-examples can be easily encoded as trees over a finite alphabet, and recognized by an automaton evaluating p and q in the usual way. Theorem 9 extends both these results. Since we have both $\sim$ and $\nsim$ in query q, we cannot assume that all data values are equal, nor that all are different; our more involved approach seems necessary.

The third relevant result of [6] is that containment of CQ p over $\mathsf{sig}_{nav}\cup\{\sim\}$ in CQ q over $\mathsf{sig}_{nav}\cup\{\sim,\nsim\}$ is 2ExpTime-complete. It looks stronger than ours because query q can

mix $\sim$ and $\nsim$. In fact, it is much weaker, depending entirely on the fact that q is a single CQ, not a UCQ. More specifically, the argument is as follows: if q uses $\nsim$, the answer is *yes* if and only if p is not satisfiable with respect to the tree automaton (a witness can have all data values equal, so it definitely does not satisfy q); if q does not use $\sim$, we are back in the case of UCQs over $\mathsf{sig}_{nav} \cup \{\sim\}$.

On the other hand, some results of Björklund, Martens, and Schwentick give a broader context to our results. They show that validity with respect to a given automaton is already 2ExpTime-complete for unions of conjunctive queries over signature sig_{nav}, that is, for trees without data. Consequently, restricting only assertions of non-mixing constraints would not lower the complexity. This is complementary to our lower bound of Proposition 11, which shows 2ExpTime-hardness for constraints using tree patterns as selectors. Hence, the only way to lower the complexity is to restrict both, selectors and assertions. Björklund, Martens, and Schwentick also show that for UCQs over $\mathsf{sig}_{nav} \cup \{\sim, \nsim\}$ validity is undecidable; this means that we cannot go beyond non-mixing assertions.

XML constraints

Non-mixing constraints form an instance of the general framework of XML-to-relational (X2R) constraints proposed by Niewerth and Schwentick [25]: tuple selectors are conjunctive queries over sig_{nav}, schemas are tree automata, and relational constraints are positive quantifier-free formulas over $\mathsf{sig}_{\sim}$ or $\mathsf{sig}_{\nsim}$. Niewerth and Schwentick investigate two classes of relational constraints: functional dependencies (FDs) and XML-key FDs (XKFDs). In the X2R setting, tuple selectors return nodes and data values, in separate "columns". In an FD

$$A_1\, A_2\, \ldots\, A_m \to B\,,$$

$A_1, A_2, \ldots, A_m, B$ are arbitrary columns (referring either to nodes or to data values); in an XKFD, B is required to be a node column. Our setting captures XKFDs, but not general FDs. Consider an X2R constraint given by a CQ $\alpha(x_1, \ldots, x_n)$ populating a table with tuples $(x_1, \ldots, x_n, @x_1, \ldots, @x_n)$, where $@x_i$ stands for the data value stored in the node represented by variable x_i, and an XKFD $x_1, \ldots, x_j, @x_{j+1}, \ldots, @x_{n-1} \to x_n$ (it makes no sense to use both x_i and $@x_i$ in the same constraint). Such constraint can be rewritten as

$$\alpha(x_1, \ldots, x_n) \wedge \alpha(x_1, \ldots, x_j, x'_{j+1}, \ldots, x'_n) \wedge x_n \neq x'_n \;\Rightarrow\; x_{j+1} \nsim x'_{j+1} \vee \cdots \vee x_{n-1} \nsim x'_{n-1}\,,$$

which can be turned into a set of five non-mixing constraints by replacing $x_n \neq x'_n$ with simple subqueries describing possible ways of arranging two different nodes in a tree, as explained in Section 2. Note that these constraints do not use $\sim$. Hence, for XKFDs with UCQs over sig_{nav} as tuple selectors decidability of entailment follows already from the results on containment of UCQs over $\mathsf{sig}_{nav} \cup \{\sim\}$, discussed in the previous subsection; the challenge tackled by Niewerth and Schwentick is to determine the exact complexity and identify tractable fragments.

If we replace the XKFD above with an FD $x_1, \ldots, x_j, @x_{j+1}, \ldots, @x_{n-1} \to @x_n$ we have

$$\alpha(x_1, \ldots, x_n) \wedge \alpha(x_1, \ldots, x_j, x'_{j+1}, \ldots, x'_n) \;\Rightarrow\; x_{j+1} \nsim x'_{j+1} \vee \cdots \vee x_{n-1} \nsim x'_{n-1} \vee x_n \sim x'_n\,,$$

which cannot be expressed without mixing $\sim$ and $\nsim$. As we have explained, consistency and entailment is undecidable for such constraints, but one can investigate fragments with restricted schemas and tuple-selectors. This is what Niewerth and Schwentick do.

As XKFDs with tree patterns as tuple-selectors can express XML Schema key and unique constraints [14], XML keys by Arenas, Fan, and Libkin [2], and XFDs by Arenas and

Libkin [3], so can non-mixing constraints. A technical subtlety is that some of these classes of constraints apply to nodes of a specified type (playing the role of a state in XML Schemas). As proposed by Niewerth and Schwentick, we can deal with it by annotating tree nodes with types (verified by the automaton encoding the schema), and let the patterns refer to types and labels. This slight extension does not affect our complexity bounds. Also, XML Schema key constraints demand that each field path selects at most one node, and XML Schema key constraints demand exactly one node; this can be checked by the automaton too. In practice, one often wants at most (or exactly) one *data value*, not tree node. This may or may not be equivalent. To express that at most one data value is selected, we can use the singleton constraints discussed in Section 2. Note that this requires assertions over $\mathsf{sig}_{\sim}$.

Consistency of XML schema mappings

Schema mappings are a formalism used in data exchange scenarios to specify relations between instances of two database schemas, a *source* schema and a *target* schema [1, 11, 21]. In the basic setting for XML [4], schemas can be abstracted as tree automata, and the relation between source and target instances can be defined by a set Σ of dependencies of the form

$$\alpha(\bar{x}) \Rightarrow \alpha'(\bar{x})$$

where α, α' are conjunctive queries over sig_{nav}, treated as queries selecting data values, not nodes. That is, a pair of data trees (t, t') satisfies dependency σ of the form above, written as $(t, t') \models \sigma$, if

$$\left\{\mathsf{val}_t(\bar{u}) \;\middle|\; t \models \alpha(\bar{u})\right\} \subseteq \left\{\mathsf{val}_{t'}(\bar{u}) \;\middle|\; t' \models \alpha'(\bar{u})\right\}.$$

The *consistency problem for XML schema mappings* [4] is to decide for a given schema mapping $\mathcal{M} = (\mathcal{A}, \mathcal{A}', \Sigma)$, whether there exists a tree t accepted by automaton $\mathcal{A}$ and a tree t' accepted by automaton $\mathcal{A}'$ such that $(t, t') \models \Sigma$. This problem is known to be decidable: without loss of generality one may assume that all data values in t and t' are equal, and use standard automata techniques ignoring data values. This is not only uninspiring theoretically, but also not very practical: an instance with all data values equal is not a convincing witness that the mapping makes sense. What if the source schema includes constraints, say XML Schema key or unique constraints? We cannot assume that all data values are equal any more. As we have argued in the previous subsection, such constraints can be expressed with non-mixing constraints, which leads us to the problem of *consistency with source constraints*, a common generalization of consistency of constraints and schema mappings: given a schema mapping $\mathcal{M} = (\mathcal{A}, \mathcal{A}', \Sigma)$ and a set of non-mixing constraints Σ_{src}, decide if there exist a tree t accepted by automaton $\mathcal{A}$ and a tree t' accepted by automaton $\mathcal{A}'$ such that $t \models \Sigma_{src}$ and $(t, t') \models \Sigma$.

The following lemma gives the connection between XML schema mappings and non-mixing constraints that allows us to apply our decidability result. It was proved in a slightly different but equivalent form in [10]. A non-mixing constraint with *free data value predicates* uses additional unary predicate symbols in the assertions. A data tree t satisfies a set Σ of such constraints (possibly sharing some additional predicate symbols) if it satisfies Σ' obtained from Σ by replacing each additional predicate symbol with some $d \in \mathbb{D}$. Free data value predicates are not problematic for the consistency algorithm, as it can guess the data values to replace them; up to equality type with respect to data values already used in Σ, there are only exponentially many possibilities.

▶ **Lemma 10.** *For each schema mapping $\mathcal{M} = (\mathcal{A}, \mathcal{A}', \Sigma)$ one can compute in doubly exponential time sets $\Sigma^1_{\sim}, \Sigma^2_{\sim}, \ldots, \Sigma^m_{\sim}$ of non-mixing constraints with free data value predicates, each*

obtained from Σ *by replacing target-side queries* $\alpha'(\bar{x})$ *with assertions* $\eta_\sim(\bar{x})$ *of exponential size, such that for each data tree* t, $t \models \Sigma_i$ *for some* $1 \leq i \leq m$ *if and only* $(t, t') \models \Sigma$ *for some data tree* t' *accepted by automaton* $\mathcal{A}'$.

Thus, mapping $\mathcal{M}$ is consistent with source constraints Σ_{src} if and only if at least one of the sets $\Sigma^i_\sim \cup \Sigma_{src}$ obtained via Lemma 10 is consistent with respect to automaton $\mathcal{A}$. Since the number of variables in each involved constraint is linear, the latter can be tested in 2ExpTime, as the algorithm from Section 3 is doubly exponential only in the maximal number of variables. As Lemma 10 translates mappings into constraints with assertions over $\mathsf{sig}_\sim$, even if Σ_{src} is just a set of key constraints (expressible with assertions over $\mathsf{sig}_{\not\sim}$), we need the full power of non-mixing constraints, allowing assertions over $\mathsf{sig}_\sim$ and $\mathsf{sig}_{\not\sim}$.

5 Conclusions

We have shown that consistency and entailment of non-mixing constraints are decidable. Both problems are 2ExpTime-complete, but become ExpTime-complete when we restrict selector queries to tree patterns and bound the number of variables in assertions. We have reinterpreted these results in terms of validity and containment of conjunctive queries, as well as consistency of schema mappings. The latter setting best illustrates the benefits of combining assertions over $\mathsf{sig}_\sim$ and $\mathsf{sig}_{\not\sim}$. Indeed, equalities are involved even in the simplest schema mappings, and inequalities allow to cover key constraints over the source database.

We worked with ordered trees, but our results immediately carry over to unordered trees: as long as the signature does not contain the horizontal axes, one can freely move back and forth between ordered and unordered trees by forgetting the sibling order or introducing it arbitrarily. As the 2ExpTime lower bound does not use the horizontal axes, it holds also for unordered trees. The reduction can be adapted to the case of unlabelled trees: one can simulate labels with unique small tree gadgets attached to the main nodes of the tree and use the automaton to ensure that each main node has exactly one gadget attached. However, referring to the gadgets with selector queries requires either the next sibling or the following sibling relation. For unordered unlabelled trees the complexity might drop.

Our decidability results can be pushed further to constraints with selector queries defined in monadic second order logic (MSO) over the signature sig_{nav} (an extension of first order logic with second-order quantification over sets). To this end, one reproves Lemma 5 using compositionality of MSO instead of Lemma 4; this requires considering MSO types and makes the bound on the data cut non-elementary. One concludes by using decidability of MSO over the signature $\mathsf{sig}_{nav} \cup \mathsf{sig}_\sim \cup \mathsf{sig}_{\not\sim}$ on trees of bounded data cut [7].

Yet another context in which our results can be applied is the static analysis of XPath queries [5]. In their basic form, our results immediately give decidability of the containment problem (in the presence of a schema) for unions of XPath queries without negation, where each query uses either equality or inequality, but never both. The extension to MSO discussed above allows free use of negation, as long as equalities and inequalities are not used under negation. But it does not give satisfying complexity bounds. It seems worthwhile to investigate the complexity issue deeper, and look at ways of extending this towards full XPath, for which decidability of satisfiability remains open.

Acknowledgements. We thank the anonymous referees for their insightful questions.

References

1 Marcelo Arenas, Pablo Barceló, Leonid Libkin, and Filip Murlak. *Foundations of Data Exchange*. Cambridge University Press, 2014.

2 Marcelo Arenas, Wenfei Fan, and Leonid Libkin. On the complexity of verifying consistency of XML specifications. *SIAM J. Comput.*, 38(3):841–880, 2008.

3 Marcelo Arenas and Leonid Libkin. A normal form for XML documents. *ACM Trans. Database Syst.*, 29:195–232, 2004.

4 Marcelo Arenas and Leonid Libkin. XML data exchange: Consistency and query answering. *J. ACM*, 55(2), 2008.

5 Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath satisfiability in the presence of DTDs. *J. ACM*, 55(2), 2008.

6 Henrik Björklund, Wim Martens, and Thomas Schwentick. Optimizing conjunctive queries over trees using schema information. In *Proc. MFCS 2008*, pages 132–143, 2008.

7 Mikołaj Bojańczyk, Filip Murlak, and Adam Witkowski. Containment of monadic datalog programs via bounded clique-width. In *Proc. ICALP 2015*, pages 427–439, 2015.

8 Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. STOC 1977*, pages 77–90, 1977.

9 Claire David, Amélie Gheerbrant, Leonid Libkin, and Wim Martens. Containment of pattern-based queries over data trees. In *Proc. ICDT 2013*, pages 201–212, 2013.

10 Claire David, Piotr Hofman, Filip Murlak, and Michał Pilipczuk. Synthesizing transformations from XML schema mappings. In *Proc. ICDT 2014*, pages 61–71, 2014.

11 Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

12 Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang Chiew Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Trans. Database Syst.*, 30(4):994–1055, 2005.

13 Ronald Fagin and Moshe Y. Vardi. The theory of data dependencies - a survey. In *Mathematics of Information Processing*, volume 34 of *Proceedings of Symposia in Applied Mathematics*, pages 19–71, Providence, Rhode Island, 1986. American Mathematical Society.

14 S. Gao, C. M. Sperberg-McQueen, H.S. Thompson, N. Mendelsohn, D. Beech, and M. Maloney. W3C XML Schema Definition Language (XSD) 1.1, Part 1: Structures. Technical report, World Wide Web Consortium, April 2009.

15 Tomasz Gogacz and Jerzy Marcinkowski. All-instances termination of chase is undecidable. In *Proc. ICALP 2014*, pages 293–304, 2014.

16 Tomasz Gogacz and Jerzy Marcinkowski. The hunt for a red spider: Conjunctive query determinacy is undecidable. In *Proc. LICS 2015*, pages 281–292, 2015.

17 Sven Hartmann and Sebastian Link. More functional dependencies for XML. In *Proc. ADBIS 2003*, pages 355–369, 2003.

18 Sven Hartmann, Sebastian Link, and Thu Trinh. Solving the implication problem for XML functional dependencies with properties. In *Proc. WoLLIC 2010*, pages 161–175, 2010.

19 Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

20 Michael Kaminski and Tony Tan. Tree automata over infinite alphabets. In Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800 of *Lecture Notes in Computer Science*, pages 386–423. Springer, 2008.

21 Phokion G. Kolaitis. Schema mappings, data exchange, and metadata management. In *Proc. PODS 2005*, pages 61–75, 2005.

22 Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proc. PODS 2002*, pages 233–246, 2002.

23 Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, 2004.

24 Frank Neven and Thomas Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Log. Meth. Comput. Sci.*, 2(3), 2006.

25 Matthias Niewerth and Thomas Schwentick. Reasoning about XML constraints based on XML-to-relational mappings. In *Proc. ICDT 2014*, pages 72–83, 2014.

26 Moshe Y. Vardi. Fundamentals of dependency theory. In E. Borger, editor, *Trends in Theoretical Computer Science*, pages 171–224. Computer Science Press, 1987.

A Compositionality for conjunctive queries over trees

▶ **Lemma 4**. *Let $\alpha(\bar{x},\bar{y})$ be a conjunction of atoms over sig_{nav}, where $\bar{x}$ and $\bar{y}$ are disjoint, and let w be a node of a data tree t. For all tuples $\bar{u},\bar{u}'$ of nodes from t_w and tuples $\bar{v},\bar{v}'$ of nodes from $t-t_w$, if $t\models\alpha(\bar{u},\bar{v})$ and $t\models\alpha(\bar{u}',\bar{v}')$, then $t\models\alpha(\bar{u},\bar{v}')$ and $t\models\alpha(\bar{u}',\bar{v})$.*

Proof. Let $\bar{u},\bar{u}',\bar{v},\bar{v}'$ be as in the statement of the lemma. Since $\alpha(\bar{x},\bar{y})$ is a conjunction of atoms, we only need to check that each atom of $\alpha(\bar{u},\bar{v}')$ and $\alpha(\bar{u}',\bar{v})$ is satisfied in t. Given that $t\models\alpha(\bar{u},\bar{v})$ and $t\models\alpha(\bar{u}',\bar{v}')$, it is enough to examine atoms using variables from both $\bar{x}$ and $\bar{y}$. That excludes unary relations and leaves us with atoms of the forms $x_i\downarrow y_j, x_i\downarrow^+ y_j, x_i\to y_j, x_i\to^+ y_j$, and symmetrical. Given that variables $\bar{x}$ are matched within t_w, and variables $\bar{y}$ are matched within $t-t_w$, atoms $x_i\downarrow y_j$, $x_i\downarrow^+ y_j$, $y_j\to x_i$, and $y_j\to^+ x_i$ are excluded by the combination of two things: the way t_w and $t-t_w$ are positioned within tree t, and the fact that $t\models\alpha(\bar{u},\bar{v})$. That is, it remains to consider $y_j\downarrow x_i$, $y_j\downarrow^+ x_i$, $x_i\to y_j$, and $x_i\to^+ y_j$. Suppose $y_j\downarrow x_i$ occurs in α. We know that $v_j\downarrow u_i$ and $v'_j\downarrow u'_i$. Since nodes u_i,u'_i are from t_w and nodes v_j,v'_j are from $t-t_w$, it follows immediately that v_j and v'_j are equal to the parent of node w, and u_i,u'_i are siblings of w or w itself. Consequently, $v_j\downarrow u'_i$ and $v'_j\downarrow u_i$. For the remaining three kinds of atoms the reasoning is similar. If $v_j\downarrow^+ u_i$ and $v'_j\downarrow^+ u'_i$, then v_j, v'_j are ancestors of w and u_i,u'_i are nodes in t_w, so $v_j\downarrow^+ u'_i$ and $v'_j\downarrow^+ u_i$ follows. If $u_i\to v_j$ and $u'_i\to v'_j$, then $u_i=u'_i=w$ and $v_j=v'_j$ is w's next sibling. Finally, if $u_i\to^+ v_j$ and $u'_i\to^+ v'_j$, then u_i,u'_j are preceding siblings of w (or w itself) and v_j,v'_j are following siblings of w. ◀

B Emptiness of register tree automata in polynomial time

▶ **Theorem 6**. *It is decidable in polynomial time whether a register tree automaton $\mathcal{A}$ accepts at least one data tree.*

Proof. We show that there exists a data tree accepted by $\mathcal{A}$ if and only if there exists a tree accepted by the (standard) tree automaton $\mathcal{A}_{fin}$ obtained from $\mathcal{A}$ by dropping all registers (we ignore k, and from each transition in $(q_{lc},q_{ps},a,q_{cur},E)\in\delta$ we drop the equivalence relation E). It is widely known that emptiness of tree automata can be checked in polynomial time, which immediately gives a polynomial-time algorithm for our problem.

Obviously, if some data tree is accepted by $\mathcal{A}$, then the tree obtained from t by ignoring data values attached to its nodes will be accepted by $\mathcal{A}_{fin}$. We only need to check the opposite direction that from every tree accepted by $\mathcal{A}_{fin}$, we can define a data tree accepted by the register automaton $\mathcal{A}$. Let ρ_Q be an accepting run of $\mathcal{A}_{fin}$ on a tree t. We extend t into a data tree (that is, we will define $\mathsf{val}_t:\mathsf{dom}_t\to\mathbb{D}$), and we extend ρ_Q to an accepting run $(\rho_Q,\rho_{\mathbb{D}})$ of $\mathcal{A}$ on this data tree. We define val_t and $\rho_{\mathbb{D}}$ starting from the root, and going down and to the left. During this process we enforce that the registers store different data values that is, $\rho_{\mathbb{D}}(x,i)\neq\rho_{\mathbb{D}}(x,j)$ whenever $i\neq j$. First, we define $\rho_{\mathbb{D}}$ in the root, assigning to the registers arbitrary (but different) values. Then, consider some node $x\in\mathsf{dom}_t$ where $\rho_{\mathbb{D}}$ is already defined. We define $\mathsf{val}_t(x)$, and $\rho_{\mathbb{D}}$ in the previous sibling x_{ps} and in the last child x_{lc} of x. Because ρ_Q is a run of $\mathcal{A}_{fin}$, there has to be a transition $(\rho_Q(x_{ps}),\rho_Q(x_{lc}),\mathsf{lab}_t(x),\rho_Q(x),E)\in\delta$ for some E. We choose some assignment of data values $as:\{val\}\cup(\{ps,lc,cur\}\times\{1,\dots,k\})\to\mathbb{D}$ such that $E=\{(a,b)\mid as(a)=as(b)\}$ and $as(cur,i)=\rho_{\mathbb{D}}(x,i)$ for all i. This is possible because each equivalence class of E contains at most one element from $\{cur\}\times\{1,\dots,k\}$, and simultaneously $\rho_{\mathbb{D}}(x,i)\neq\rho_{\mathbb{D}}(x,j)$ for $i\neq j$. We define $\mathsf{val}_t(x)=as(val)$, $\rho_{\mathbb{D}}(x_{ps},i)=as(ps,i)$, and $\rho_{\mathbb{D}}(x_{lc},i)=as(lc,i)$. Because each

equivalence class of E contains at most one element from $\{ps\} \times \{1, \ldots, k\}$ and at most one from $\{lc\} \times \{1, \ldots, k\}$, we obtain that $\rho_{\mathbb{D}}(x_{ps}, i) \neq \rho_{\mathbb{D}}(x_{ps}, j)$ and $\rho_{\mathbb{D}}(x_{lc}, i) \neq \rho_{\mathbb{D}}(x_{lc}, j)$ for $i \neq j$, so the process can be continued smoothly till the end. By construction, $(\rho_Q, \rho_{\mathbb{D}})$ is an accepting run of $\mathcal{A}$ on the data tree obtained by extending t with val_t. Altogether emptiness of $\mathcal{A}$ is equivalent to emptiness of $\mathcal{A}_{fin}$. ◀

C The lower bound for the consistency problem

▶ **Proposition 11.** *Consistency of non-mixing constraints is* 2ExpTime*-hard.*

Proof. Recalling that 2ExpTime = AExpSpace, we will be using alternating Turing machines. Such machines can be defined in multiple similar ways, and we use a definition that is most convenient for our encoding; it is standard to check that our definition is equivalent to other commonly used definitions. We do not divide states of our machine into existential and universal; we only distinguish accepting states. Instead we use the following notion of a run tree, requiring that from every configuration two different transitions can be applied. A *run tree* of an alternating Turing machine M on input word w is a tree labelled by configurations of M, where

- the root is labelled by the initial configuration for the input word w;
- every node not labelled by an accepting configuration has exactly two children, labelled by successors of this configuration, reached by applying to it two different transitions;
- every node labelled by an accepting configuration is a leaf.

We say that an input word w is accepted by M if there is a finite run tree of M for w.

Consider an alternating Turing machines M (of the above form) that works in space bounded by $2^{|w|}$, where w is the input word, and an input word w (although we limit the space to $2^{|w|}$ instead of considering any exponential function, already among such machines there is one solving an AExpSpace-hard problem). We show that we can create (in polynomial time) an input to the consistency problem: a tree automaton $\mathcal{A}$ and a set Σ of non-mixing constraints such a that $\mathcal{A}$ and Σ are consistent if and only if M accepts w. More precisely, every tree $t \in L(\mathcal{A})$ such that $t \models \Sigma$ describes a run-tree of M on w. Below we specify how such a tree t encodes a run-tree of M on w, simultaneously saying how these properties are ensured by $\mathcal{A}$ or Σ.

Nodes labelled by s form a prefix of t that is a binary tree: the parent of every s-labelled node (if exists) is s-labelled, and every s-labelled node has zero or two s-labelled children. This is ensured by the automaton $\mathcal{A}$. This part of t is called the *skeleton*, and will have the same shape as the run-tree.

Additionally, each node of the skeleton has a c-labelled child (beside of the zero or two children from the skeleton). The subtree starting in this child forms a path, whose labels match to the regular expression

$$\mathsf{c}\, A\, Q\, A\, Q\, (\mathsf{l} + \mathsf{n} + \mathsf{r})\, (\$\, (\mathsf{h} + \mathsf{n})\, (\mathsf{p} + \mathsf{n})\, \mathsf{c}^{2n+1})^+ \#\,,$$

where Q and A are the state set and the tape alphabet of M, and $n = |w|$. Again, this is ensured by $\mathcal{A}$. This path, called a *configuration path* describes a configuration of M assigned to the corresponding node of the skeleton (that is simultaneously a node of the run tree).

At the beginning of each configuration path we have a transition used to reach this configuration: the second node is labelled by the letter present on the tape under the head in the previous configuration; the third node by the previous state; the fourth by the letter written on the tape; the fifth by the new state; the sixth by the direction in which the head

was moved (left, no movement, right). The automaton ensures that this is indeed a valid transition of M (except of the configuration path directly below the root, where we only ensure that the fifth node contains the initial state); that the label of the third node (previous state) is equal to the label of the fifth node (current state) of the parent configuration; that the transitions assigned to sibling configurations are different (as required in the definition of a run tree); that states in all leaf configurations are accepting.

The next part of a configuration path consists of multiple *blocks* of length $2n+4$; each of them describes a single letter on the tape. To identify a block, we use the first n c-labelled nodes for a binary counter encoding the position in the tape, using data values $0, 1 \in \mathbb{D}$. We assign data values $0, \ldots, 0, 0$ to these nodes in the first block, $0, \ldots, 0, 1$ in the second block, and so on, until $1, \ldots, 1, 1$ in the last block (we have 2^n blocks, which equals to the length of the tape). The next n nodes of the block also contain such a counter, but going back: we assign $1, \ldots, 1, 1$ to these nodes in the first block, and $0, \ldots, 0, 0$ in the last block. Notice that when one counter of a block contains bits $b_1, \ldots, b_n$, then the other counter contains their inverses $1 - b_1, \ldots, 1 - b_n$. This double encoding of the position is the key trick that enables to use non-mixing integrity constraints to check correctness of the run between two consecutive configurations. To enforce this behaviour of counters we use non-mixing integrity constraints. It is easy to write a conjunctive query $\alpha_{fb}(\bar{x})$ that matches nodes of the first block in any configuration path, where $\bar{x} = (x_\$, x_h, x_p, x_1, \ldots, x_n, \hat{x}_1, \ldots, \hat{x}_n, x_d)$. We take to Σ the constraint $\alpha_{fb}(\bar{x}) \Rightarrow 0(x_1) \wedge \cdots \wedge 0(x_n) \wedge 1(\hat{x}_1) \wedge \cdots \wedge 1(\hat{x}_n)$. Similarly we do for the last block. Then, using a conjunctive query $\alpha_{cb}(\bar{x}, \bar{y})$ that matches nodes of two consecutive blocks we take a constraint $\alpha_{cb}(\bar{x}, \bar{y}) \Rightarrow \eta_\sim(\bar{x}, \bar{y})$ ensuring that counters in these two blocks encode consecutive numbers. It is a standard task to express this property as a positive Boolean combination $\eta_\sim(\bar{x}, \bar{y})$ of atoms over $\{\sim, 0, 1\}$, of a quadratic size.

The second node of a block is marked by h if the head of M is placed over this position of the tape, and the third node of block is marked by p if the head was placed over this position in the previous configuration. The automaton ensures that exactly one block of a configuration path is marked by h (by p); that in the initial configuration the head is over the first letter; that the relation between the p and h markers on a configuration path is as described by the sixth node of that path (l, n, or r). To ensure that the position of p corresponds to the position of h in the previous configuration we use the constraint $\alpha_{ch}(\bar{x}, \bar{y}) \Rightarrow x_1 \sim y_1 \wedge \cdots \wedge x_n \sim y_n$, where $\alpha_{ch}(\bar{x}, \bar{y})$ matches to the h-marked block of a configuration and to the p-marked block of a child configuration.

The last node of a block carries the tape letter (from A) in the data value. To ensure that the initial configuration starts with the input word, we write a constraint $\alpha_{ini}(x_1, \ldots, x_n) \Rightarrow \eta_\sim(x_1, \ldots, x_n)$, where $\alpha_{ini}(x_1, \ldots, x_n)$ selects the last node from each of the first n blocks of the topmost configuration path (to make sure that only the topmost configuration path is selected, we can check for the presence of the initial state, assuming w.l.o.g. that M cannot reach the initial state in any transition). Another constraint $\alpha_{bl}(x) \Rightarrow \mathsf{b}(x)$ ensures that the rest of the initial tape contains blanks $\mathsf{b} \in A$, where $\alpha_{bl}(x)$ matches the last node of a block of the topmost configuration path other than the first n blocks. Next $|A|$ constraints ensure that the p-marked block contains the letter written in the fourth node of the configuration path (letter written under the head), and another $|A|$ constraints that the h-marked block of the previous configuration contains the letter written in the second node of the configuration path (letter seen under the head).

Finally, we have to ensure that the content of the tape is preserved (except the single letter under the head). Let $\alpha_{2b}(\bar{x}, \bar{y})$ be a conjunctive query matching some blocks on consecutive configuration paths, where the first of them is not marked by h. For every such pair $\bar{x}, \bar{y}$,

we want to enforce that either the two corresponding blocks carry the same letter or they represent two different positions in the tape. Using the double complementary encoding of the position in the blocks, this can be enforced using only $\sim$ in the following constraints: $\alpha_{2b}(\bar{x}, \bar{y}) \Rightarrow x_1 \sim \hat{y}_1 \vee \cdots \vee x_n \sim \hat{y}_n \vee x_d \sim y_d$. Notice that the property "$\bar{x}$ and $\bar{y}$ encode different positions in the tape" seems to require $\nsim$, but thanks to the inverted counter stored in each block we may use $\sim$ instead, avoiding the illegal mixture of $\sim$ and $\nsim$ in the assertion of the constraint. ◀

We remark that all conjunctive queries used in the above proof could be written using tree patterns, and that the set $\Sigma_{\nsim}$ was empty. Thus the 2ExpTime-hardness result holds already for constraints of this form.

D A singly exponential fragment

▶ **Proposition 8**. *For non-mixing constraints whose selectors are tree patterns and whose assertions use constantly many variables, consistency and entailment are* ExpTime*-complete.*

Proof. We have already argued that the bound on the number of data values to be stored in registers is polynomial. It remains to modify the construction of the register automaton. The method is standard: instead of remembering all partial matchings of selectors, we would like to keep only matchings of full subtrees (linearly many in the size of the selector). For each matched subtree we would like to we store tuples representing data values of those variables from the subtree that are also used in the corresponding assertion, call them *assertion variables*. By the hypothesis of the theorem, the number of such tuples is still polynomial. However, actual matchings of tree patterns in the processed part of the input data tree are not single subtrees, but collections of subtrees, which may share data values the automaton has forgotten in the meantime. We would like a flat representation with all subtrees stored independently, but we must keep the information about shared data values. We go for a compromise. Subtrees without assertion variables may be stored independently, as for them we ignore data values in the matched nodes, and only need the information that they were matched. The remaining subtrees have to be kept together, but this is not problematic any more, as there is only constantly many of them in each matching. The number of sets of such subtrees is polynomial. For each of such sets we store a polynomial set of tuples describing data values stored in tree nodes assigned to assertion variables. Thus, the automaton remembers in each state a collection of independent subtrees without assertion variables and pairs consisting of: a set of subtrees with assertion variables and a set of tuples. Hence, it is exponential.

The lower bound follows from ExpTime-hardness of consistency of schema mappings with trivially unsatisfiable right hand sides of dependencies [1, Proposition 18.2], which can be also reinterpreted as validity of unions of tree patterns (modulo a given tree automaton). ◀