Synthesizing transformations from XML schema mappings

Claire David
University Paris-Est Marne
Claire.David@univ-mlv.fr

Piotr Hofman
University of Warsaw
ph209519@mimuw.edu.pl

Filip Murlak
University of Warsaw
fmurlak@mimuw.edu.pl

Michał Pilipczuk
University of Bergen
michal.pilipczuk@ii.uib.no

ABSTRACT

XML schema mappings have been developed and studied in the context of XML data exchange, where a source document has to be restructured under the target schema according to certain rules. The rules are specified with a mapping, which consists of a set of source-to-target dependencies based on tree patterns. The problem of building a target document for a given source document and a mapping has polynomial data complexity, but is still intractable due to high combined complexity.

We consider a two layer architecture for building target instances, inspired by the Church synthesis problem. We view the mapping as a specification of a document transformation, for which an implementation must be found. The static layer inputs a mapping and synthesizes a single XML-to-XML query implementing a valid transformation. The data layer amounts to evaluating this query on a given source document, which can be done by a specialized query engine, optimized to handle large documents.

We show that for a given mapping one can synthesize a query expressed in an XQuery-like language, which can be evaluated in time proportional to the evaluation time of the patterns used in the mapping. In general the involved constant is high, but it can be improved under additional assumptions. In terms of overall complexity, if the arity of patterns is considered constant, we obtain a fixed-parameter tractable procedure with respect to the mapping size, which improves previously known upper bounds.

Categories and Subject Descriptors

H.2.5 [**Database Management**]: Heterogeneous Databases—*Data translation*; I.7.2 [**Document and Text Processing**]: Document Preparation—*XML*

General Terms

Theory, Languages, Algorithms

Keywords

data exchange, building solutions, document transformations, queries returning trees

ICDT'14 March 24-28, 2014, Athens, Greece.

1. INTRODUCTION

One of the challenges of data management is dealing with heterogeneous data. A typical scenario is that of data exchange, in which the source instance of a database has to be restructured under the target schema according to certain rules. The rules are specified in a declarative fashion, as source-to-target dependencies that express properties of the target instance, based on properties of the source instance.

Mapping between relational schemas are well understood (see recent surveys [3, 5, 6, 18]). Prototypes of tools for specifying and managing mappings have been developed and some have been incorporated into commercial ETL (extract-transform-load) systems [14, 20, 23]. In the XML context, while commercial ETL tools often claim to provide support for XML schema mappings, this is typically done by means of dependencies that essentially establish connections between attributes in two schemas of a restricted form. In research literature, a more expressive formalism of XML schema mappings was developed using tree patterns in order to specify complex transformations exploiting the tree structure of XML documents [1, 4].

For such mappings, the problem of constructing a valid target instance for a given source instance is highly non-trivial due to the subtle interplay between the properties imposed by the dependencies and the structural constraints of the target schema. For a fixed mapping the target instance can be constructed in polynomial time, but in terms of combined complexity the problem is NEXPTIME-hard [8]. In this work we analyze the problem in the spirit of parametrized complexity: we cannot beat the NEXPTIME lower bound, but we can still hope for polynomial data complexity with the degree of the polynomial independent of the mapping. Moreover, from the practical point of view it is desirable to separate the static part of the computation, dealing only with the mapping, from the data-dependent part. Ideally, the data stage should rely as much as possible on a specialized query engine, optimized to handle large data.

We consider a generic two-layer architecture for building target instances. Inspired by the Church synthesis problem [10], and later work on schema mappings [17, 21], we view the mapping as a declarative specification of a document transformation, for which a working implementation must be synthesized. The static layer inputs a mapping and synthesizes an XML-to-XML query (in an XQuery-like language) implementing a valid transformation. The data layer amounts to evaluating the query on a given source document. The challenge is to synthesize a query whose data complexity does not exceed drastically the data complexity of queries involved in the dependencies.

Our contributions. We show that given a mapping $\mathcal{M}$ one can synthesize an implementing query $q_{\mathcal{M}}$ that can be evaluated on the source tree T in time $C_{\mathcal{M}} \cdot |T|^{\mathcal{O}(r)}$, where r is the maximal number of variables in the patterns used in $\mathcal{M}$. That is, the complexity is fixed-parameter tractable wrt. the size of the mapping, if r is considered a fixed constant; we refer to the books of Downey and Fellows [13] or Flum and Grohe [15] for an introduction to the parametrized complexity. Constant $C_{\mathcal{M}}$ may be large in general, but we identify a class of tractable mappings, where $C_{\mathcal{M}}$ is polynomial in the size of $\mathcal{M}$ and minimal target documents.

Our approach relies on the idea of splitting the target schema into several templates, which are later filled with data values and multiple instances of generic small fragments of trees in such a way that all the dependencies are satisfied. The most costly part is choosing the constants to fill in the attributes in the fixed part of the template. The brute-force method of trying all possible values from the source tree has unacceptable data complexity. We give three different methods to solve this problem more efficiently:

- a branching algorithm that fixes the attributes iteratively using tuples extracted from the source tree by source-side patterns, and backtracks in case of failure;
- a method exploiting the concept of kernelization, which amounts here to finding a small subset of tuples, sufficient to determine the attributes of the template;
- an algorithm that splits the source schema into templates and uses the fact that for *absolutely consistent* mappings (admitting a valid target instance for each source instance) the attributes of the target template depend only on the attributes of the source template.

The three methods give similar complexity bounds, but the ideas behind them are very different. We believe that together they offer deeper understanding of the problem, as well as a broader spectrum of techniques to be used in solutions tailored for real-life scenarios. Our algorithm for tractable mappings refines the brute-force solution, using ideas similar to the ones behind the third approach.

Related work. In the classical setting of relational data exchange with mappings given by source-to-target tuple-generating dependencies, there is no reason for a two layer architecture since the mapping itself can be used to construct target solutions by means of the chase procedure. A two layer architecture has been considered for a different kind of mappings, describing two-way data flows between databases and applications [21]. These mappings are compiled into Entity SQL views defining the application's data model in terms of the database instance, and *vice versa*.

Most research on the synthesis of XML transformations focuses on building complex transformations from existing ones by means of high level operations [6, 20]. Synthesizing transformations from a declarative specification is considered in [17], but the setting allows only simple schemas in which elements contain several subelements and several collections of subelements of the same type. The dependencies are expressed in terms of child relation and element types. The solution amounts to producing small XML documents which are then merged into a single document conforming to the schema; the focus is on performing the merge efficiently. In our approach there is no merging involved; the structural conditions of the schema are analyzed beforehand and reflected in the templates.

Organization. After recalling the basic notions (Sect. 2) and introducing the transformation language (Sect. 3), we describe a simple approach which essentially casts the solution building algorithm from [8] in our two layer setting. Next we describe the branching algorithm (Sect. 5), the kernelization method (Sect. 6), and the algorithm for absolutely consistent mappings (Sect. 7). Finally, we discuss the tractable case (Sect. 8) and conclude with ideas for future work (Sect. 9). Some arguments are moved to the Appendix.

2. PRELIMINARIES

Data trees. The abstraction of XML documents we use is *data trees*: unranked labelled trees storing in each node a *data value*, i.e., an element of a countable infinite data domain $\mathbb{D}$. For concreteness, we will assume that $\mathbb{D}$ contains the set of natural numbers $\mathbb{N}$. Formally, a *data tree* over a finite labelling alphabet Γ is a structure $\mathcal{T} = \langle T, \downarrow, \downarrow^+, \rightarrow, \rightarrow^+, lab_{\mathcal{T}}, \rho_{\mathcal{T}} \rangle$, where

- the set T is an unranked tree domain, i.e., a prefix-closed subset of $\mathbb{N}^*$ such that $n \cdot i \in T$ implies $n \cdot j \in T$ for all $j < i$;
- the binary relations $\downarrow$ and $\rightarrow$ are the child relation ($n \downarrow n \cdot i$) and the next-sibling relation ($n \cdot i \rightarrow n \cdot (i+1)$);
- $\downarrow^+$ and $\rightarrow^+$ are the transitive closures of $\downarrow$ and $\rightarrow$;
- $lab_{\mathcal{T}} \colon T \rightarrow \Gamma$ is the labelling function;
- $\rho_{\mathcal{T}} \colon T \rightarrow \mathbb{D}$ assigns data values to nodes. We say that a node $s \in T$ stores the value d when $\rho_{\mathcal{T}}(s) = d$.

When the interpretations of $\downarrow, \rightarrow, lab_{\mathcal{T}}, \rho_{\mathcal{T}}$ are understood, we write just T instead of $\mathcal{T}$. We use the terms "tree" and "data tree" interchangeably.[1] We write $|T|$ for the number of nodes of T.

Forests and contexts. A *forest* is a sequence of trees. We write $F + G$ for the concatenation of forests F, G and $L + M$ for $\{F + G \mid F \in L, G \in M\}$ for sets of forests L, M. If $L = \{F\}$ we write simply $F + M$.

A *multicontext* C over an alphabet Γ is a tree over $\Gamma \cup \{\circ\}$ such that $\circ$-labelled nodes have at most one child. The nodes labelled with $\circ$ are called *ports*. A *context* is a multicontext with a single port, which is additionally required to be a leaf. A leaf port u can be *substituted* with a forest F, which means that in the sequence of the children of u's parent, u is replaced by the roots of F. An internal port u can be substituted with a context C' with one port u': first the subtree rooted at u's only child is substituted at u', then the obtained tree is substituted at u. Formally, the ports of a multicontext store data values just like ordinary nodes, but these data values play no role and we will leave them unspecified.

For a context C and a forest F we write $C \cdot F$ to denote the tree obtained by substituting the unique port of C with F. If we use a context D instead of the forest F, the result of the substitution is a context as well. Again, we extend the operation $\cdot$ to two sets of contexts in the natural way.

Schemas. A *document type definition* (DTD) over a labelling alphabet Γ is a pair $\mathcal{D} = \langle r_{\mathcal{D}}, P_{\mathcal{D}} \rangle$, where

- $r_{\mathcal{D}} \in \Gamma$ is a distinguished root symbol;
- $P_{\mathcal{D}}$ is a function assigning regular expressions over Γ to the elements of Γ, usually written as $\sigma \rightarrow e$, if $P_{\mathcal{D}}(\sigma) = e$.

[1] A different abstraction allows several *attributes* in each node, each attribute storing a data value [1, 4]. Attributes can be modelled easily with additional children, without influencing the complexity of the problems we consider.

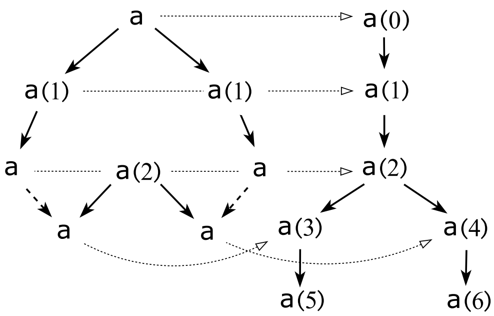

Figure 1: Homomorphisms witness satisfaction (solid and dashed arrows are child and descendant relations).

A data tree T *conforms to* a DTD $\mathcal{D}$, if its root is labelled with $r_{\mathcal{D}}$ and for each node $s \in T$ the sequence of labels of children of s is in the language of $P_{\mathcal{D}}(lab_T(s))$. The set of data trees conforming to $\mathcal{D}$ is denoted $L(\mathcal{D})$. Unless stated otherwise, we assume $r_{\mathcal{D}}$ is a fixed label r.

A *forest DTD* is defined like a DTD, only instead of a single root symbol it has a regular expression. For a forest DTD $\mathcal{D} = \langle e, P_{\mathcal{D}} \rangle$, $L(\mathcal{D})$ is the set of forests of the form $T_1T_2 \dots T_p$ whose sequence of root labels $\sigma_1\sigma_2 \dots \sigma_p$ is a word in the language of e and $T_i \in L(\langle \sigma_i, P_{\mathcal{D}} \rangle)$.

A *context DTD* over Γ is a DTD $\mathcal{D}$ over $\Gamma \cup \{\circ\}$ such that each tree over $\Gamma \cup \{\circ\}$ conforming to $\mathcal{D}$ has exactly one node (a leaf) labelled with $\circ$.

Patterns. Patterns were originally invented as convenient syntax for conjunctive queries on trees [9, 16]. While XML schema mappings literature mostly concentrates on tree-shaped patterns, definable in XPath-like syntax [1, 4], without disjunction the full expressive power of conjunctive queries is only guaranteed by DAG-shaped patterns. Following [8] we base our mappings on DAG-shaped patterns.

A *(pure) pattern* π over Γ can be presented as

$$\pi = \langle V, E_c, E_d, E_n, E_f, lab_\pi, \xi_\pi \rangle$$

where $\langle V, E_c \cup E_d \cup E_n \cup E_f \rangle$ is a finite DAG whose edges are split into child edges E_c, descendant edges E_d, next sibling edges E_n, and following sibling edges E_f; lab_π is a partial function from V to Γ; ξ_π is a partial function from V to some set of variables. The range of ξ_π, denoted $\mathrm{Rg}\,\xi_\pi$, is the set of variables used by π; the *arity* of π is $|\,\mathrm{Rg}\,\xi_\pi|$; $\|\pi\|$ is the size of the underlying DAG.

A data tree $\mathcal{T} = \langle T, \downarrow, \downarrow^*, \to, \to^*, lab_T, \rho_T \rangle$ *satisfies* a pattern $\pi = \langle V, E_c, E_d, E_n, E_f, lab_\pi, \xi_\pi \rangle$ under a valuation $\theta : \mathrm{Rg}\,\xi_\pi \to \mathbb{D}$, denoted $\mathcal{T} \models \pi\theta$, if there exists a homomorphism $h\colon \pi \to \mathcal{T}$ i.e., a function $\mu : V \to T$ such that

- $\mu : \langle V, E_c, E_d, E_n, E_f \rangle \to \langle T, \downarrow, \downarrow^*, \to, \to^* \rangle$ is a homomorphism of relational structures;
- $lab_{\mathcal{T}}(\mu(v)) = lab_\pi(v)$ for all $v \in \mathrm{Dom}\, lab_\pi$; and
- $\rho_{\mathcal{T}}(\mu(u)) = \theta(\xi_\pi(u))$ for all $u \in \mathrm{Dom}\ \xi_\pi$.

We write $\pi(\bar{x})$ to express that $\mathrm{Rg}\,\xi_\pi \subseteq \bar{x}$. For $\pi(\bar{x})$, instead of $\pi\theta$ we usually write $\pi(\bar{a})$, where $\bar{a} = \theta(\bar{x})$. We say that $\mathcal{T}$ satisfies π, denoted $\mathcal{T} \models \pi$, if $\mathcal{T} \models \pi\theta$ for some θ. Figure 1 shows an example of a pattern and a homomorphism.

Note that we use the usual non-injective semantics, where different vertices of the pattern can be witnessed by the same tree node, as opposed to injective semantics, where each vertex is mapped to a different tree node [11]. Under the adopted semantics patterns are closed under conjunction: $\pi_1 \wedge \pi_2$ can be expressed by the disjoint union of π_1 and π_2.

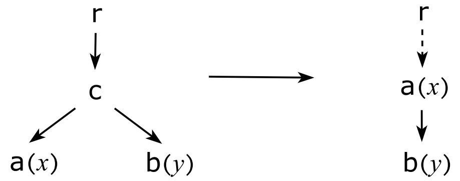

Figure 2: Dependencies are expressed with patterns.

We enrich pure patterns with explicit equalities and inequalities between data variables, i.e., if $\pi(\bar{x})$ is a pure pattern and $\eta(\bar{x})$ is a conjunction of equalities and inequalities over $\bar{x}$, then $\pi'(\bar{x}) = (\pi, \eta)(\bar{x})$ is a (non-pure) pattern. We write $T \models \pi'(\bar{a})$ if $T \models \pi(\bar{a})$ and $\eta(\bar{a})$ holds.

Schema mappings. A *schema mapping* $\mathcal{M} = \langle \mathcal{D}_s, \mathcal{D}_t, \Sigma \rangle$ consists of a source DTD $\mathcal{D}_s$, a target DTD $\mathcal{D}_t$, and a set Σ of *(source-to-target) dependencies* that relate source and target instances. Dependencies are expressions of the form:

$$\pi(\bar{x}) \;\longrightarrow\; \pi'(\bar{x}, \bar{y}),$$

where π, π' are patterns and each variable in $\bar{x}$ is used in the pure pattern underlying π (the usual safety condition).

A pair of trees (T, T') satisfies the dependency above if for all $\bar{a}$, $T \models \pi(\bar{a})$ implies $T' \models \pi'(\bar{a}, \bar{b})$ for some $\bar{b}$. Given a source $T \in L(\mathcal{D}_s)$, a target $T' \in L(\mathcal{D}_t)$ is a *solution* for T under $\mathcal{M}$ if (T, T') satisfies each dependency in Σ. We let $\mathcal{M}(T)$ stand for the set of all solutions for T.

EXAMPLE 1. Let $\mathcal{M} = \langle \mathcal{D}_s, \mathcal{D}_t, \Sigma \rangle$, where $\mathcal{D}_s$ is $\mathsf{r} \to \mathsf{c}$; $\mathsf{c} \to \mathsf{a}^*\mathsf{b}^*$, $\mathcal{D}_t$ is $\mathsf{r} \to (\mathsf{c}|\mathsf{d})\mathsf{a}^*$; $\mathsf{a} \to \mathsf{b}^*$, and Σ consists of the single dependency in Fig. 2. Under $\mathcal{M}$ each source tree has a solution. On the other hand, if we replace the target DTD with $\mathsf{r} \to (\mathsf{c}|\mathsf{d})\mathsf{a}$; $\mathsf{a} \to \mathsf{b}^*$, only trees that store the same data value in all a-nodes have solutions. □

3. TRANSFORMATION LANGUAGE

For the transformation language we choose a fragment of XQuery, extended with an additional construct for manipulating contexts. We use the following streamlined syntax:

$$\begin{aligned}
q(\bar{x}) ::= \;& \sigma(x_i)\big[q'(\bar{x})\big] \;\big|\; q'(\bar{x}), q''(\bar{x}) \;\big|\; \mathsf{first}(q'(\bar{x})) \\
\big|\;& e(\bar{x}) \;\big|\; \rho(e(\bar{x})) \\
\big|\;& \mathsf{if}\ b(\bar{x})\ \mathsf{then}\ q'(\bar{x})\ \mathsf{else}\ q''(\bar{x}) \\
\big|\;& \mathsf{let}\ y := q'(\bar{x})\ \mathsf{return}\ q''(\bar{x}, y) \\
\big|\;& \mathsf{for}\ y\ \mathsf{in}\ q'(\bar{x})\ \mathsf{where}\ b(\bar{x}, y)\ \mathsf{return}\ q''(\bar{x}, y) \\
b(\bar{x}) ::= \;& q(\bar{x}) = q'(\bar{x}) \;\big|\; \mathsf{empty}(q(\bar{x})) \\
\big|\;& \neg b'(\bar{x}) \;\big|\; b'(\bar{x}) \vee b''(\bar{x}) \;\big|\; b'(\bar{x}) \wedge b''(\bar{x}) \\
e(\bar{x}) ::= \;& (x_i \mid .)\big(step \mid [f]\big)^* \\
step ::= \;& \downarrow \mid \downarrow^+ \mid \uparrow \mid \uparrow^+ \mid \to \mid \to^+ \mid \leftarrow \mid \leftarrow^+ \\
f ::= \;& \sigma \mid e \mid \neg f' \mid f' \vee f'' \mid f' \wedge f''
\end{aligned}$$

where q's are the queries, b's are the Boolean tests, e's are the CoreXPath expressions (starting in a node x_i or in the root). We adopt the standard XQuery semantics. The queries return sequences of trees or atomic values (or nodes, identified with subtrees), and variables can store all of these as well. The expression $\sigma(x_i)\big[q'(\bar{x})\big]$

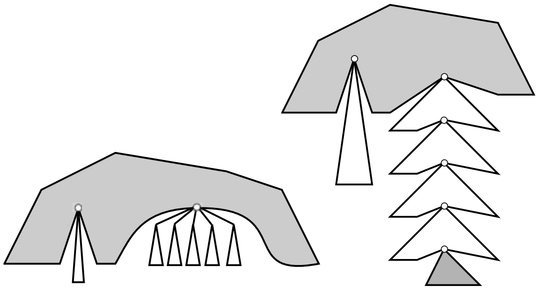

Figure 3: Combining trees horizontally and vertically.

returns the tree obtained by substituting the forest returned by q' at the port of the context consisting of the port and the root labelled with $\sigma \in \Gamma$ and storing the data value x_i; $q'(\bar{x}), q''(\bar{x})$ returns the concatenation of the results of $q'(\bar{x})$ and $q''(\bar{x})$; $\mathsf{first}(q(\bar{x}))$ gives the first element of the sequence returned by q; $\rho(e(\bar{x}))$ returns the sequence of data values stored in the sequence of nodes returned by $e(\bar{x})$; $\mathsf{let}\ y := q'(\bar{x})\ \mathsf{return}\ q''(\bar{x}, y)$ returns the sequence returned by $q''(\bar{x}, y)$ where y is evaluated to the sequence returned by $q'(\bar{x})$; $\mathsf{for}\ y\ \mathsf{in}\ q'(\bar{x})\ \mathsf{where}\ b(\bar{x}, y)\ \mathsf{return}\ q''(\bar{x}, y)$ returns the concatenation of the sequences returned by $q''(y, \bar{x})$ for all values of y returned by $q'(\bar{x})$ that satisfy $b(\bar{x}, y)$. In Sect. 6 and Sect. 7 we use additional standard features of XQuery; we explain them there.

Consider the mapping $\mathcal{M}$ defined in Example 1. In order to implement it with a query, we need to assume that a function $\mathsf{freshnull}()$ returning a fresh null value at each call is available. An implementing query can be written as

$$\begin{array}{l}
\mathsf{r}\big[\ \mathsf{let}\ z := \mathsf{freshnull}()\ \mathsf{return}\ \mathsf{c}(z)\,, \\
\quad \mathsf{for}\ v\ \mathsf{in}\ .\downarrow[\mathsf{c}]\ \mathsf{return} \\
\qquad \mathsf{for}\ x\ \mathsf{in}\ \rho(v\downarrow[\mathsf{a}])\ \mathsf{return} \\
\qquad\quad \mathsf{for}\ y\ \mathsf{in}\ \rho(v\downarrow[\mathsf{b}])\ \mathsf{return}\ \mathsf{a}(x)[\mathsf{b}(y)]\ \big]
\end{array}$$

In queries implementing mappings patterns must be expressed as queries. For this, it is convenient to assume that queries can return tuples of data values, e.g., the source side pattern of $\mathcal{M}$ (see Fig. 2) could be expressed with query q_{src}:

$$\begin{array}{l}
\mathsf{for}\ v\ \mathsf{in}\ .\downarrow[\mathsf{c}]\ \mathsf{return} \\
\quad \mathsf{for}\ x\ \mathsf{in}\ \rho(v\downarrow[\mathsf{a}])\ \mathsf{return} \\
\qquad \mathsf{for}\ y\ \mathsf{in}\ \rho(v\downarrow[\mathsf{b}])\ \mathsf{return}\ (x, y)
\end{array}$$

and the implementing query $q_{\mathcal{M}}$ can be written as

$$\begin{array}{l}
\mathsf{r}\big[\ \mathsf{let}\ z := \mathsf{freshnull}()\ \mathsf{return}\ \mathsf{c}(z)\,, \\
\quad \mathsf{for}\ (x, y)\ \mathsf{in}\ q_{src}\ \mathsf{return}\ \mathsf{a}(x)[\mathsf{b}(y)]\ \big]
\end{array}$$

This can be simulated in XQuery by returning flat trees with as many children as the tuples have entries, and selecting the data values from the children with path expressions.

Since each DAG pattern can be expressed as a disjunction of exponentially many tree patterns [16], each DAG pattern can be expressed as a query returning tuples of data values.

LEMMA 1 ([16]). *For each pattern $\pi(\bar{x})$ there exists a query q_π that returns exactly those tuples $\bar{a}$ for which $\pi(\bar{a})$ is satisfied in the tree. The query can be synthesized in time $2^{poly(||\pi||)}$ and evaluated over T in time $2^{poly(||\pi||)} \cdot |T|^r$ where r is the number of variables in the pattern. If π is a tree-shaped pattern, the synthesis time is $poly(||\pi||)$ and the evaluation time is $poly(||\pi||) \cdot |T|^r$.*

Finally, in order to construct trees conforming to arbitrary recursive DTDs, we need a way to produce and concatenate contexts, not just forests. For instance, if the target DTD in $\mathcal{M}$ is changed to $\mathsf{r} \to \mathsf{a}; \mathsf{a} \to \mathsf{ab} \,|\, \mathsf{db}$ then the only way to obtain a solution is to go deeper and deeper in the tree, as shown in the right hand tree in Fig. 3. To enable this, we extend the transformation language with **context expressions**

$$\begin{array}{rl}
c(\bar{x}) ::= & \circ \ \big|\ \sigma(x_i)[\circ]\ \big|\ c'(\bar{x})\big[c''(\bar{x})\big]\ \big|\ q(\bar{x}), c'(\bar{x}), q'(\bar{x}) \\
& \big|\ \mathsf{let}\ y := q'(\bar{x})\ \mathsf{returnC}\ c'(\bar{x}, y) \\
& \big|\ \mathsf{for}\ y\ \mathsf{in}\ q'(\bar{x})\ \mathsf{where}\ b(\bar{x}, y)\ \mathsf{returnC}\ c'(\bar{x}, y)
\end{array}$$

and replace $\sigma(x_i)[q'(\bar{x})]$ in the productions for q as follows:

$$q(\bar{x}) ::= \quad \dots \ \big|\ c(\bar{x})[q'(\bar{x})]\ \big|\ \dots$$

The semantics of $\mathsf{for}\ y\ \mathsf{in}\ q'(\bar{x})\ \mathsf{where}\ b(\bar{x}, y)\ \mathsf{returnC}\ c'(\bar{x}, y)$ is a context obtained by combining all results of $c'(\bar{x}, y)$ vertically, plugging one in another. Using this construct, the modified mapping can be implemented with the query

$$\mathsf{r}\big[\ \big(\mathsf{for}\ (x_1, x_2)\ \mathsf{in}\ q_{src}\ \mathsf{returnC}\ \mathsf{a}(x_1)[\circ, \mathsf{b}(x_2)]\ \big)[q_{\mathsf{d}}]\ \big]\,,$$

where q_{d} is $\mathsf{let}\ y := \mathsf{freshnull}()\ \mathsf{return}\ \mathsf{d}(y)$.

4. SIMPLE SOLUTION

In this section we show how the general solution building algorithm from [8] can be used to synthesize a query implementing a given mapping $\mathcal{M} = \langle \mathcal{D}_s, \mathcal{D}_t, \Sigma \rangle$, i.e., a query $q_{\mathcal{M}}$ such that $q_{\mathcal{M}}(T) \in \mathcal{M}(T)$ for each source tree T that admits a solution. Building a solution for T amounts to producing a tree $T' \models \mathcal{D}_t$ that satisfies each pattern from

$$\Delta = \big\{ \psi(\bar{a}, \bar{y}) \ \big|\ \varphi(\bar{x}) \longrightarrow \psi(\bar{x}, \bar{y}) \in \Sigma, T \models \varphi(\bar{a}) \big\}\ ,$$

which is an instance of the satisfiability problem for patterns. Satisfiability is well-known to be NP-complete, so this gives an algorithm exponential in $\|\Delta\|$, which can be as large as $|T|^r$, where r is the maximal arity of patterns in $\mathcal{M}$. We are aiming at an algorithm polynomial in $|T|^r$. We shall exploit the fact that patterns in Δ have size independent of T.

The algorithm from [8] essentially works as follows:

1. for each $\delta \in \Delta$ build $T_\delta \in L(\mathcal{D}_t)$ such that $T_\delta \models \delta$,
2. combine the T_δ's into $T' \in L(\mathcal{D}_t)$ such that $T' \models \Delta$.

Step (1) can be done in time independent from T for each δ, but (2) is not obvious: how do we combine the T_δ's into a solution? While some parts of $\mathcal{D}_t$ may be flexible enough to accommodate corresponding fragments from all T_δ's, some other parts require that all the T_δ's agree. For instance, according to the modified target DTD $\mathsf{r} \to (\mathsf{c}|\mathsf{d})\mathsf{a}$; $\mathsf{a} \to \mathsf{b}^*$ in Example 1, in each solution T' the root, the a-node, and its sibling are unique, and if the T_δ's are to be combined, they need to agree on the data values stored in these nodes and on the label of the a-node's sibling. On the other hand, T' can contain multiple b-nodes with different data values.

The idea of the algorithm is to split the target schema $\mathcal{D}_t$ into so-called *kinds*, in which the fixed and the flexible parts are clearly identified, and try to find T_δ's consistent with a single kind. The only requirement for the flexible parts is that they allow easy combination of smaller fragments. A natural condition would be closure under concatenation, but for complexity reasons we use weaker conditions that allow additional padding between the combined fragments.

DEFINITION 1 (KIND). *A kind $\mathcal{K}$ is a multicontext whose each port u is equipped with a language L_u of compatible forests or contexts that can be substituted at u. If u is a leaf, then one of the following holds:*

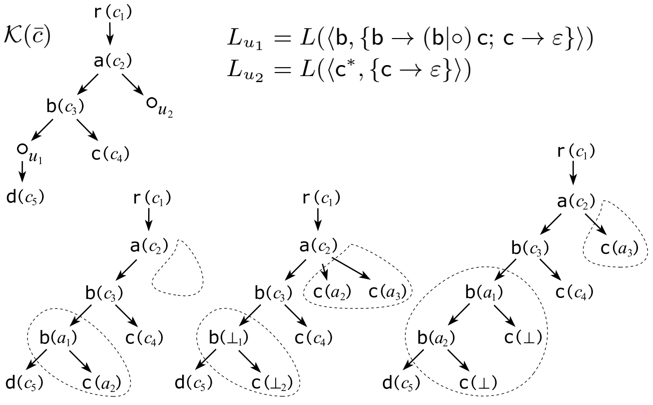

Figure 4: A data kind $\mathcal{K}(\bar{c})$ and three trees in $L(\mathcal{K}(\bar{c}))$.

(1) L_u is a DTD-definable set of forests and for all $F \in L_u$,

$$F + F' + L_u \subseteq L_u$$

for some forest F'; or

(2) L_u is DTD-definable set of trees and for all $T \in L_u$,

$$C'(T, L_u) \subseteq L_u$$

for some multicontext C' with two ports u_1, u_2, where $C'(T, L_u)$ is the set of trees obtained by substituting T at u_1 and some $T' \in L_u$ in u_2.

If u is an internal node, then

(3) L_u is a DTD-definable set of contexts and for all $C \in L_u$,

$$C \cdot C' \cdot L_u \subseteq L_u$$

for some context C'.

Depending on the type, we distinguish forest (1), tree (2) and context ports (3). We assume that the root of $\mathcal{K}$ is not a forest port, i.e., a single forest port is not a kind.

We write $L(\mathcal{K})$ for the set of trees T obtained from $\mathcal{K}$ by substituting at each port u a compatible forest, tree, or context T_u according to the type of u. We call sequence $(T_u)_u$ a *witnessing substitution*. A *witnessing decomposition* of T is a sequence of disjoint sets $(Z_u)_u$ of nodes of T such that T restricted to Z_u is a copy of T_u and T restricted to the complement of $\bigcup_u Z_u$ is a copy of $\mathcal{K}$. We shall identify T_u and $\mathcal{K}$ with their copies in T (the *components* of the decomposition) and speak of the witnessing decomposition $(T_u)_u$.

As we have seen, data values in the copy of $\mathcal{K}$ have to agree in all T_δ's, so they have to be determined in advance. By filling in the data values we obtain a *data kind*. We write $\mathcal{K}(\bar{c})$ to denote the data kind obtained from $\mathcal{K}$ by assigning $\bar{c}$ to the ordinary nodes of $\mathcal{K}$, assuming some implicit order on them. Each $\mathcal{K}(\bar{c})$ defines language $L(\mathcal{K}(\bar{c}))$ of data trees. Figure 4 shows a data kind $\mathcal{K}(\bar{c})$ and some trees in $L(\mathcal{K}(\bar{c}))$.

Definition 1 ensures that sequences of compatible forests or contexts can be combined into one compatible forest or context: for compatible forests $F_1, F_2, \dots, F_n$ there are forests $I_1, I_2, \dots, I_{n-1}$ such that $F_1 + I_1 + F_2 + I_2 + \ldots + F_n$ is compatible; for compatible trees $S_1, S_2, \dots, S_n$ there are multicontexts $I_1, I_2, \dots, I_{n-1}$ with two ports such that $I_1(S_1, \circ) \cdot I_2(S_2, \circ) \cdot \ldots \cdot I_{n-1}(S_{n-1}, S_n)$ is compatible, where $I_j(S_j, \circ)$ is a context obtained by substituting S_j at the first port of I_j; and for compatible contexts $C_1, C_2, \dots, C_n$ there are contexts $I_1, I_2, \dots, I_{n-1}$ such that $C_1 \cdot I_1 \cdot C_2 \cdot I_2 \cdot \ldots \cdot C_n$ is compatible. This gives a natural way to combine trees from $L(\mathcal{K}(\bar{c}))$: a *combination* of $T^1, T^2, \dots, T^n \in L(\mathcal{K}(\bar{c}))$ with decompositions $(T^j_u)_u$ is a tree from $L(\mathcal{K}(\bar{c}))$ obtained by substituting at each port u a compatible forest or context combining $T^1_u, T^2_u, \dots, T^n_u$. In general there is no guarantee that a combination of the T_δ's satisfies each δ, but we can ensure it by assuming that δ is matched in T_δ in a special way defined below.

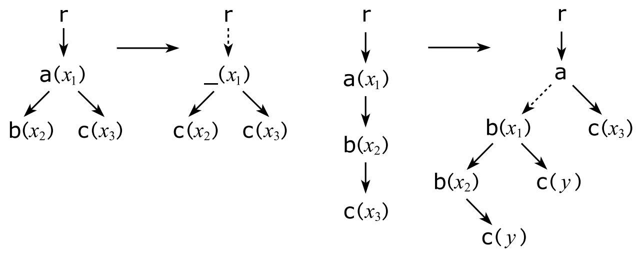

Figure 5: A generic mapping.

DEFINITION 2 (NEAT MATCHING). *Let $T \in L(\mathcal{K})$ and let $(T_u)_u$ be a witnessing decomposition of T. A pattern π is matched neatly in T (with respect to $(T_u)_u$) if there exists a* neat homomorphism *$\mu\colon \pi(\bar{a}) \to T$, i.e., a homomorphism such that for all vertices x, y of π*

- *if $E_n(x, y)$ then $\mu(x)$ and $\mu(y)$ are in the same component;*
- *if $E_c(x, y)$ then either $\mu(x)$ and $\mu(y)$ are in the same component, or $\mu(x)$ is in the copy of $\mathcal{K}$ in T and $\mu(y)$ is a root of a forest component;*
- *if $E_f(x, y)$ then either $\mu(x)$ and $\mu(y)$ are in the same component, or each is a root of a forest component or a node in the copy of $\mathcal{K}$ in T.*

It is easy to see that neat matchings guarantee that each combination of all T_δ's satisfies each δ (see Appendix A).

LEMMA 2. *If T' is a combination of $T_\delta \in L(\mathcal{K}(\bar{c}))$ with decomposition $(T^\delta_u)_u$ for $\delta \in \Delta$ and each δ is matched neatly in T' with respect to $(T^\delta_u)_u$, then $T' \models \Delta$.*

As we shall see later, it suffices to consider kinds for which neat matchings always exist. A kind $\mathcal{K}$ is a *target kind* for $\mathcal{M}$ if $L(\mathcal{K}) \subseteq L(\mathcal{D}_t)$, and for each target-side pattern π in $\mathcal{M}$ if $\pi(\bar{a})$ can be matched in a tree from $L(\mathcal{K}(\bar{c}))$, then it can also be matched *neatly* in some tree from $L(\mathcal{K}(\bar{c}))$. For a target kind $\mathcal{K}$, the two step algorithm discussed above computes a solution in $L(\mathcal{K}(\bar{c}))$, if there is one. The following lemma shows that one can synthesize a query that implements this algorithm. We write $|\mathcal{K}|$ for the number of nodes of $\mathcal{K}$ and $\|\mathcal{K}\|$ for the maximal size of DTDs in $\mathcal{K}$.

LEMMA 3. *For each mapping $\mathcal{M}$ and target kind $\mathcal{K}$ there is a query $sol_{\mathcal{K}}(\bar{z})$ such that for each tree T that admits a solution in $L(\mathcal{K}(\bar{c}))$, $sol_{\mathcal{K}}(\bar{c})(T)$ is a solution for T. The synthesis time for $sol_{\mathcal{K}}$ is $2^{poly(\|\mathcal{K}\|, \|\mathcal{M}\|)} \cdot |\mathcal{K}|^{\mathcal{O}(p+r)}$ and the evaluation time is $2^{poly(\|\mathcal{M}\|, \|\mathcal{K}\|)} \cdot |\mathcal{K}|^{r+1} \cdot |T|^r$ where r and p are the maximal arity and size of patterns in $\mathcal{M}$.*

The proof can be found in Appendix B. Here we give an example for a relatively generic mapping.

EXAMPLE 2. Let $\mathcal{M}$ be a mapping with source DTD $\mathcal{D}_s$: $\mathsf{r} \to \mathsf{a}^*$; $\mathsf{a} \to \mathsf{b}\,\mathsf{c}$; $\mathsf{b} \to \mathsf{c}$, target DTD $\mathcal{D}_t$: $\mathsf{r} \to \mathsf{a}$; $\mathsf{a} \to \mathsf{b}\,\mathsf{c}^*$; $\mathsf{b} \to$

(b | d)c, and dependencies $\pi_1(\bar{x}) \longrightarrow \pi'_1(\bar{x})$, $\pi_2(\bar{x}) \longrightarrow \pi'_2(\bar{x}, y)$ shown in Fig. 5. The kind $\mathcal{K}(\bar{c})$ with $\bar{c} = c_1, c_2, c_3, c_4$, shown in Fig. 4, is a target kind for $\mathcal{M}$.

First we need $T_{\pi'_1(\bar{a})} \in L(\mathcal{K}(\bar{c}))$ for all $\bar{a} = a_1, a_2, a_3$ such that $\pi_1(\bar{a})$ holds in in the source tree, and similarly for π'_2. When we synthesize the query we have no access to the source tree; we provide generic trees that depend only on the equality type of the entries of $\bar{a}$. There are two essentially different ways to match neatly $\pi'_1(\bar{a})$ in a tree from $\mathcal{K}(\bar{c})$: match the vertex without label to one of the b-nodes outside $\mathcal{K}$ and both c-vertices to its only c-child (left tree in Fig. 4), or match the vertex without label to the unique a-node and the c-vertices to some of its c-children (middle tree in Fig. 4; nodes storing nulls $\bot_1, \bot_2$ are required by L_{u_1}). The first matching allows arbitrary a_1, but a_2 and a_3 have to be equal, the second one allows arbitrary a_2 and a_3, but a_1 has to be equal to c_2. For $\pi'_2(\bar{a})$ the only choice is where to match the b-nodes: inside or outside of $\mathcal{K}$. In a neat matching both have to be mapped outside of $\mathcal{K}$ (right tree in Fig. 4; null value $\bot$ realises the variable y).

The query $sol_{\mathcal{K}}(\bar{z})$ computes tuples for which π_1 and π_2 hold in the input tree and returns a combination of the appropriate instances of the generic trees. It generates fresh nulls $\bar{y} = y_1, y_2, y_3$ and returns $\mathcal{K}(\bar{c})$ with $\bar{c}$ replaced by $\bar{z}$ and ports u_1, u_2 replaced by a context expression $q_{u_1}(\bar{y})$ and a subquery q_{u_2}:

```
let $y_1 :=$ freshnull() return
  let $y_2 :=$ freshnull() return
    let $y_3 :=$ freshnull() return
      $\mathsf{r}(z_1)\big[\mathsf{a}(z_2)\big[\,\mathsf{b}(z_3)\big[q_{u_1}(\bar{y})[\mathsf{d}(z_5)], \mathsf{c}(z_4)\big],\ q_{u_2}\,\big]\big]$ .
```

In $q_{u_1}(\bar{y}) = q^1_{u_1}[q^2_{u_1}(y_1, y_2)[q^3_{u_1}(y_3)]]$, expression $q^1_{u_1}$ combines substitutions at port u_1 coming from the first way of matching π'_1,

$$\text{for } \bar{x} \text{ in } q_{\pi_1} \text{ where } x_2 = x_3 \text{ returnC } \mathsf{b}(x_1)\big[\,\circ\,, \mathsf{c}(x_2)\big]\ ,$$

$q^2_{u_1}(y_1, y_2)$ combines those coming from the second way,

$$\text{for } \bar{x} \text{ in } q_{\pi_1} \text{ where } x_1 = z_2 \text{ returnC } \mathsf{b}(y_1)\big[\,\circ\,, \mathsf{c}(y_2)\big]\ ,$$

and $q^3_{u_1}(y_3)$ combines those coming from matching π'_2,

$$\text{for } \bar{x} \text{ in } q_{\pi_2} \text{ returnC } \mathsf{b}(x_1)\big[\mathsf{b}(x_2)\big[\,\circ\,, \mathsf{c}(y_3)\big], \mathsf{c}(y)\big]\ .$$

Note that $q^2_{u_1}(y_1, y_2)$ can be optimized to $\mathsf{b}(y_1)\big[\,\circ\,, \mathsf{c}(y_2)\big]$. In $q_{u_2} = \big(q^1_{u_2}, q^2_{u_2}\big)$, subquery $q^1_{u_2}$ combines substitutions at port u_2 coming from the second way of matching π'_1,

$$\text{for } \bar{x} \text{ in } q_{\pi_1} \text{ where } x_1 = z_2 \text{ return } \ \mathsf{c}(x_2), \mathsf{c}(x_3)\ ,$$

and $q^1_{u_2}$ combines substitutions coming from matching π'_2,

$$\text{for } \bar{x} \text{ in } q_{\pi_2} \text{ return } \ \mathsf{c}(x_3)\ .$$

Clearly, $sol_{\mathcal{K}}(\bar{c})(T)$ is a solution for T, unless $T \models \pi_1(\bar{a})$ for some $\bar{a}$ such that neither $a_1 = c_2$ nor $a_2 = a_3$. But then T admits no solution in $L(\mathcal{K}(\bar{c}))$ at all. □

It remains to compute the data values $\bar{c}$ to be put in the ordinary nodes of $\mathcal{K}$. Tuple $\bar{c}$ depends on the input tree T: in Example 2, $\bar{c}$ is good if $T \models \pi_1(\bar{a})$ implies that either $a_1 = c_2$ or $a_2 = a_3$. A similar characterisation is always a by-product of $sol_{\mathcal{K}}(\bar{z})$ (see Appendix B).

LEMMA 4. *Let $\mathcal{M}$ be a mapping with dependencies $\pi_i(\bar{x}_i) \to \pi'_i(\bar{x}_i, \bar{y}_i)$ for $i = 1, 2, \ldots, n$ and let $\mathcal{K}$ be a target kind. There exist formulae $\alpha_i(\bar{x}_i, \bar{z})$ such that*

- *$\alpha_i(\bar{x}_i, \bar{z})$ is a disjunction of at most $|\mathcal{K}|^r r^r$ conjunctions of $\mathcal{O}(|\pi'_i|)$ equalities and inequalities among $\bar{x}_i$ and $\bar{z}$, where r is the maximal arity of patterns in $\mathcal{M}$;*
- *for each $\bar{c}$, each source tree T admits a solution in $L(\mathcal{K}(\bar{c}))$ iff $T \models \pi_i(\bar{a})$ implies $\alpha_i(\bar{a}, \bar{c})$ for all i.*

The α_i's can be computed from $sol_{\mathcal{K}}$ in polynomial time.

We shall call the α_i's the *potential expressions* for $\mathcal{K}$. Note that $\bar{z}$ are common for all α_i; we refer to them as the *constants* of $\mathcal{K}$. In the symbols of Lemma 4 we can write the following simple query $const_{\mathcal{K}}$, computing a suitable valuation of the constants of $\mathcal{K}$, if it exists:

```
first(for $\bar{z}$ in $values_{|\bar{z}|}$ where
          empty(for $\bar{x}_1$ in $q_{\pi_1}$ where $\neg\alpha_1(\bar{x}_1, \bar{z})$ return $\bar{x}_1$)
            ⋮
        $\wedge$ empty(for $\bar{x}_n$ in $q_{\pi_n}$ where $\neg\alpha_n(\bar{x}_n, \bar{z})$ return $\bar{x}_n$)
      return $\bar{z}$)
```

where $values_{|\bar{z}|}$ is a query that returns all possible tuples of length $|\bar{z}|$ with entries from the set of data values used in the input tree or a fixed set of nulls of size $|\bar{z}|$. The nulls are needed, since inequalities may enforce some constants to be different from any data value used in the source document.

The evaluation time of $const_{\mathcal{K}}$ on T is proportional to $|T|^{|\mathcal{K}|}$, which is highly impractical; in the following sections we shall optimize it so that the evaluation time does not drastically exceed that of $sol_{\mathcal{K}}$. For now, let us finish the construction of the implementing query $q_{\mathcal{M}}$.

We say that $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_k$ cover a language L if $L \subseteq \bigcup_{i=1}^{k} L(\mathcal{K}_i)$. The following lemma shows that the target domain of any mapping can be covered with small target kinds (see Appendix C for the proof). For a DTD $\mathcal{D}$, the *branching* is the maximal size of regular expressions used in $\mathcal{D}$, and the *height* is the maximal number of different labels on a branch in any tree from $L(\mathcal{D})$.

LEMMA 5. *For each mapping $\mathcal{M}$ there exist target kinds $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_k$ covering $L(\mathcal{D}_t)$ such that $|\mathcal{K}_i| \leq K$, $\|\mathcal{K}_i\| \leq \|D_t\|$, and the whole sequence of kinds can be computed in time $2^{K \cdot poly(\|\mathcal{M}\|)}$; here $K = (2pb + b)^{2ph+h}$, where b and h are the branching and height of $\mathcal{D}_t$, and p is the maximal size of target side patterns in $\mathcal{M}$.*

If $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_k$ are the target kinds guaranteed by Lemma 5, the query $q_{\mathcal{M}}$ can be defined as:

```
if $\neg$empty($const_{\mathcal{K}_1}$) then let $\bar{z} := const_{\mathcal{K}_1}$ return $sol_{\mathcal{K}_1}(\bar{z})$ else
        ⋮
if $\neg$empty($const_{\mathcal{K}_k}$) then let $\bar{z} := const_{\mathcal{K}_k}$ return $sol_{\mathcal{K}_k}(\bar{z})$ .
```

Using the bounds of Lemmas 3–5, we have that the synthesis time for $q_{\mathcal{M}}$ is $2^{K \cdot poly(\|\mathcal{M}\|)}$ and the evaluation time over T is $2^{K \cdot poly(\|\mathcal{M}\|)} \cdot |T|^{K+r}$.

5. OPTIMIZING VIA BRANCHING

In this section we show an optimization of the solution given in the previous section. The query $q_{\mathcal{M}}$ presented there runs through all valuations of the constants of target kind $\mathcal{K}$ with data values from the input document and nulls. This can be highly inefficient if $\mathcal{K}$ is large: the resulting number of valuations can be much larger than the space of tuples considered in the dependencies. We present a simple branching strategy that avoids enumeration of all valuations.

Our algorithm executes some queries, whose number depends only on $\mathcal{M}$, such that each query has linear data complexity and runs over the set of tuples selected by a single source-side pattern, instead of all valuations of the constants of $\mathcal{K}$. This gives running time $f(|\mathcal{K}|) \cdot |T|^r$, rather than $\mathcal{O}(|T|^{\mathcal{O}(|\mathcal{K}|)})$, for some function f, where T is the source tree and r is the maximal arity of patterns in $\mathcal{M}$. Thus, the presented solution is *fixed-parameter tractable* in the sense of Downey and Fellows [13], when r is treated as a constant and $\|\mathcal{M}\|$ is treated as a parameter (the solution in Sect. 4 is not fixed-parameter tractable). Rigorous bounds on function f will be still quite intractable (double exponential in $\|\mathcal{M}\|$); in Sect. 8 we improve them under additional assumptions.

By Lemma 4, finding the constants of kind $\mathcal{K}$ amounts to solving the following more general *tuple covering problem*: given potential expressions $\alpha_i(\bar{x}_i, \bar{z})$ and sets $D_i \subseteq \mathbb{D}^{|\bar{x}_i|}$ for $i = 1, 2, \ldots, n$, find a tuple $\bar{c}$ such that $\alpha_i(\bar{a}, \bar{c})$ holds for all i and $\bar{a} \in D_i$, or assert that such $\bar{c}$ does not exist (D_i plays the role of the set of tuples selected by pattern $\pi_i(\bar{x}_i)$).

LEMMA 6. *The tuple covering problem for potential expressions $\alpha_1(\bar{x}_1, \bar{z}), \ldots, \alpha_n(\bar{x}_n, \bar{z})$ and sets $D_1, \ldots, D_n$ can be solved by an algorithm executing at most*

$$n \cdot (1 + \max_{i=1,\ldots,n} k_i)^{\mathcal{O}(|\bar{z}|^2)}$$

linear queries over single sets D_i, where k_i is the number of clauses in expression α_i. Moreover, if expressions α_i use no inequalities over $\bar{z}$, the number of queries is bounded by

$$n \cdot (1 + \max_{i=1,\ldots,n} k_i)^{2|\bar{z}|}.$$

PROOF. Let $\alpha_i(\bar{x}_i, \bar{z}) = \bigvee_{j=1}^{k_i} P^i_j(\bar{x}_i, \bar{z})$, where each clause P^i_j is a conjunction of equalities and inequalities. We implement a simple branching strategy. The algorithm maintains the following information: (i) a tuple $\bar{c} \in (\mathbb{D} \cup \{\bot\})^{|\bar{z}|}$ valuating $\bar{z}$, where $c_i = \bot$ means that z_i has not been assigned a value yet; (ii) a consistent set $\mathcal{E}$ of constraints enforced on variables z_i that have not been valuated so far: these constraints may be of the from $z_i = z_j$, $z_i \neq z_j$, or $z_i \neq d$ for $d \in \mathbb{D}$. We assume that information propagates, e.g., if $c_1 \neq \bot$ and $z_1 = z_2$ is present in $\mathcal{E}$, we have $c_2 = c_1$.

A tuple $\bar{a} \in D_i$ is *covered* by clause P^i_j under $(\bar{c}, \mathcal{E})$, if the conjuncts of $P^i_j(\bar{a}, \bar{c})$ satisfy the following conditions:

1. conjuncts of the form $x_\ell = x_{\ell'}$ and $x_\ell \neq x_{\ell'}$ hold;
2. conjuncts of the form $x_\ell = z_{\ell'}$ hold, i.e., $c_{\ell'} = a_\ell \in \mathbb{D}$;
3. conjuncts of the form $x_\ell \neq z_{\ell'}$ hold, i.e., $z_{\ell'}$ is valuated to something different from a_ℓ, or is not valuated yet;
4. conjuncts of form $z_\ell = z_{\ell'}$ and $z_\ell \neq z_{\ell'}$ hold if z_ℓ, $z_{\ell'}$ are valuated, and if not, they are implied by $\mathcal{E}$.

Note that conjuncts $x_\ell \neq z_{\ell'}$ do not impose any conditions on the future values of not yet valuated $z_{\ell'}$. Hence, some tuples may cease to be covered when $z_{\ell'}$ finally gets its value.

The algorithm begins with empty partial valuation $\bar{c} = (\bot, \ldots, \bot)$ and $\mathcal{E} = \emptyset$, and refines them iteratively so that some uncovered tuple gets covered at each step. While there are uncovered tuples, pick one of them, say $\bar{a} \in D_i$, and branch into k_i subcases, choosing a clause P^i_j to cover $\bar{a}$. Try *fixing* P^i_j at $\bar{a}$ by extending $(\bar{c}, \mathcal{E})$ so that $\bar{a}$ is covered by P^i_j: fix the values of all $z_{\ell'}$ considered in condition 2, add to $\mathcal{E}$ all the equalities and inequalities considered in condition 4, propagate information from $\mathcal{E}$ and remove all the constraints referring to valuated variables only. Note that fixing P^i_j at $\bar{a}$ may be impossible due to inconsistency with $(\bar{c}, \mathcal{E})$. In that case, we discard the sub-branch. If no P^i_j can be fixed at $\bar{a}$, we discard the whole branch. When all tuples are covered, it remains to valuate the missing $z_{\ell'}$ so that each tuple actually satisfies the covering clause. In particular, we need to satisfy all the constraints of the form $x_\ell \neq z_{\ell'}$ that were ignored so far. This is achieved by valuating all not yet valuated variables $z_{\ell'}$ to fresh nulls (respecting the equalities in $\mathcal{E}$). The obtained $\bar{c}$ is a correct answer to the tuple covering problem.

To see that the algorithm is complete, assume that $\alpha_i(\bar{a}, \bar{c})$ holds for all $\bar{a} \in D_i$ and all i. Then, the branch where for each picked tuple $\bar{a} \in D_i$ we fix a clause P^i_j such that $P^i_j(\bar{a}, \bar{c})$ holds, is never discarded. Hence, it outputs a correct valuation (possibly different from $\bar{c}$).

Finally, let us analyze the complexity. Observe that fixing clause P^i_j at a picked $\bar{a}$ that is not covered so far results in one of the following: either (i) one of the constants of $\bar{z}$ is assigned a value, or (ii) an equality is added to the set $\mathcal{E}$, or (iii) an inequality is added to the set $\mathcal{E}$. If expressions α_i contain no inequalities over $\bar{z}$, then (iii) actually never happens. On a single branch of the algorithm, (i) happens at most $|\bar{z}|$ times, (ii) happens at most $|\bar{z}|$ times since equalities are propagated in a transitive manner, and (iii) happens at most $\binom{|\bar{z}|}{2}$ times. Hence, the depth of the branching tree is bounded by $|\bar{z}| + |\bar{z}| + \binom{|\bar{z}|}{2} = \mathcal{O}(|\bar{z}|^2)$, and by $2|\bar{z}|$ in case there are no inequalities over $\bar{z}$ in expressions α_i.

Since at each step the algorithm branches to at most $\max_{i=1}^{n} k_i$ subcases, the total size of the branching tree is at most $(1 + \max_{i=1}^{n} k_i)^{\mathcal{O}(|\bar{z}|)^2}$, or $(1 + \max_{i=1}^{n} k_i)^{2|\bar{z}|}$ if there are no inequalities over $\bar{z}$. In each node we execute n linear queries identifying uncovered tuples, one for each α_i. The bounds on the total number of queries follow. □

This algorithm can be easily encoded in XQuery (see Appendix G). The resulting query can be plugged in instead of $const_{\mathcal{K}}$ in the the query $q_{\mathcal{M}}$ from Section 4, with D_i replaced by the results of queries q_{π_i}. Moreover, if we assume that there are no inequalities involving variables introduced on the target side of $\mathcal{M}$, then the potential expressions given by Lemma 4 do not contain any inequalities between constants, and thus the algorithm of Lemma 6 uses less queries. Hence, by applying the bounds of Lemma 4 we obtain the following (note here that $\log K = poly(\|\mathcal{M}\|)$).

THEOREM 1. *For each mapping $\mathcal{M}$ one can compute in time $2^{K \cdot poly(\|\mathcal{M}\|)}$ an implementing query $q_{\mathcal{M}}$ whose evaluation time over T is*

$$2^{K^2 \cdot poly(\|\mathcal{M}\|)} \cdot |T|^r,$$

where $K = (2pb + b)^{2ph+h}$, b and h are the branching and height of $\mathcal{D}_t$, while p and r are the maximal size and arity of patterns in $\mathcal{M}$. Moreover, the evaluation time may be reduced to $2^{K \cdot poly(\|\mathcal{M}\|)} \cdot |T|^r$ in case when there are no inequalities involving variables introduced on the target side.

6. OPTIMIZING VIA KERNELIZATION

In this section we present yet another approach of optimizing the brute-force approach of Sect. 4, which can turn out to be more efficient than the one presented in Sect. 5. Unfortunately, our solution does not cope with the full generality of mappings considered in the previous sections, as we have to exclude some inequality constraints.

Our idea is to shrink the set of interesting data values from the input document. We prove that one can find a small, that is of cardinality independent of the size of the input document, subset of data values, about which we can safely assume that constants in the kind can be valuated only to elements of this subset. The original motivation of our approach is the concept of *kernelization*, a notion widely used in the parameterized complexity. Although our framework is not exactly compatible with the notion of kernel used there, the technique is very similar in principles. Again, we refer to the textbooks by Downey and Fellows [13] and by Flum and Grohe [15] for a more extensive introduction to kernelization; a direct inspiration is the work of Langerman and Morin [19]. The crucial concept is the notion of a *kernel*.

DEFINITION 3. *Let $\alpha(\bar{x}, \bar{z})$ be a potential expression and let $D \subseteq \mathbb{D}^{|\bar{x}|}$. We say that $D' \subseteq D$ is a* kernel for D with respect to α *if for every $\bar{c}$, $\forall_{\bar{a} \in D}\ \alpha(\bar{a}, \bar{c}) \iff \forall_{\bar{a} \in D'}\ \alpha(\bar{a}, \bar{c})$.*

Intuitively, a kernel is therefore a small subset of tuples that can replace the whole database for the purpose of solving the tuple covering problem. The following simple claim follows directly from the definition.

LEMMA 7. *If D' is a kernel for D w.r.t. α and D'' is a kernel for D' wr.t. α, then D'' is a kernel for D w.r.t. α.*

We now prove that if the potential expressions use no inequality, we can obtain a surprisingly small kernel. As we will later see, applying the brute-force method of Sect. 4 on this kernel gives an algorithm with comparable performance as the branching algorithm of Theorem 1.

THEOREM 2. *Let $\alpha(\bar{x}, \bar{z})$ be a potential expression with k clauses, using only equality, and let $D \subseteq \mathbb{D}^r$, $r = |\bar{x}|$. Then there exists a kernel D' for D with respect to α of size at most $2 \cdot (2k)^r$. Moreover, D' can be found by an algorithm making $\mathcal{O}(kr(2k)^r \cdot \log|D|)$ quadratic calls to D, deleting some tuples from D until D' is obtained.*

PROOF. We begin by reformulating the tuple covering problem in terms of linear algebra. By identifying data values with natural numbers we may treat D as a subset of the r-dimensional real space $\mathbb{R}^r$. Recall that an *affine* subset of $\mathbb{R}^r$ is a subset of the form $\Pi = \{\bar{a} \in \mathbb{R}^r \mid A\bar{a} = \bar{b}\}$ where A is an $d \times r$ real matrix and $\bar{b} \in \mathbb{R}^d$; the dimension of Π is $r - d$ for the minimal d such that Π can be presented this way. Assume $\alpha(\bar{x}, \bar{z}) = \bigvee_{i=1}^k P_i(\bar{x}, \bar{z})$. Observe that the set

$$P_i^{\bar{c}} = \{\bar{a} \in \mathbb{R}^r \mid P_i(\bar{a}, \bar{c})\}$$

is affine for each $\bar{c}$ and i; indeed, it is defined by a conjunction of linear equations. We say that a set $S \subseteq \mathbb{R}^r$ *covers* a set $S' \in \mathbb{R}^r$ if $S \supseteq S'$. Thus we can restate the tuple covering problem as follows:

$$\text{given } D \subseteq \mathbb{R}^r \text{, find } \bar{c} \text{ such that } \textstyle\bigcup_{i \le k} P_i^{\bar{c}} \text{ covers } D.$$

We are now ready to present the algorithm. Owing to Lemma 7, we can refine the kernel iteratively, starting from D: as long as the current kernel is not small enough, we identify a subset that can be removed to obtain a smaller kernel. The final size of the kernel is the minimal size for which we can still find points to remove.

In each iteration, the algorithm identifies a large subset X of the current kernel, such that a constant fraction of X can be removed. We claim that if

$$\text{for all } i, \text{ for all } \bar{c},\ X \subseteq P_i^{\bar{c}} \ \text{ or } \ |X \cap P_i^{\bar{c}}| < \frac{|X|}{2k} \qquad (1)$$

then any subset Y of X with at most $\frac{|X|}{2}$ elements can be removed. Indeed, assume that $D \setminus Y$ is covered for some $\bar{c}$. If $X \subseteq P_i^{\bar{c}}$ for some i, then Y is covered, and we are done. Assume this is not the case. Then, by (1), each $P_i^{\bar{c}}$ covers strictly less then $\frac{|X|}{2k}$ elements of X. Hence, $\bigcup_{i \le k} P_i^{\bar{c}}$ covers strictly less then $\frac{|X|}{2}$ elements of X. This contradicts the fact that $\bigcup_{i \le k} P_i^{\bar{c}}$ covers $X \setminus Y$ (and $D \setminus Y$).

We identify an appropriate set X by means of the following iterative procedure, which refines a candidate for X. We begin with $X_0 = D$. In iteration j, we input candidate X_j and test if satisfies property (1). If so, we return $X = X_j$. If not, we find a new (smaller) candidate X_{j+1}: since (1) does not hold, some affine subset $P_i^{\bar{c}}$ covers at least $\frac{|X_j|}{2k}$ elements of X_j, but not all of them; let $X_{j+1} = X_j \cap P_i^{\bar{c}}$.

We claim that after at most r iterations the procedure outputs some X of size at least $\frac{|D|}{(2k)^r}$. Note that $|X_{j+1}| \ge \frac{|X_j|}{2k}$ for all j. The claim follows immediately from the fact that X_j is contained in an affine subset of dimension $r - j$. To prove this fact, we proceed by induction. The base case $j = 0$ is trivial. Assume X_j is contained in an affine subset Π_j of dimension $r - j$. Note that X_{j+1} is the intersection of X_j and some affine subset $P_i^{\bar{c}}$ that does not contain X_j. Consequently, Π_j is not contained in $P_i^{\bar{c}}$. Hence, the intersection $\Pi_j \cap P_i^{\bar{c}}$ is an affine subset of dimension smaller than the dimension of Π_j, i.e., at most $r - (j + 1)$.

To make sure that we can actually delete a nonempty set of points Y, we need to assume that $|X| > 2$. This is guaranteed as long as $|D| > 2(2k)^r$. How many times do we need to apply the kernelization procedure to obtain a kernel of size at most $2(2k)^r$? After each $\mathcal{O}((2k)^r)$ iterations the cardinality of the set D is halved, which means that we need only $\mathcal{O}((2k)^r \cdot \log|D|)$ iterations.

It remains to compute X with $\mathcal{O}(rk)$ quadratic queries over D. For a clause P_i and a tuple $\bar{a} \in \mathbb{D}^r$, define $\hat{P}_i^{\bar{a}}$ as

$$\hat{P}_i^{\bar{a}} = \{\bar{b} \in \mathbb{R}^r \mid \exists \bar{z}\, P_i(\bar{b}, \bar{z}) \wedge P_i(\bar{a}, \bar{z})\} = \bigcup_{\bar{c}:\ \bar{a} \in P_i^{\bar{c}}} P_i^{\bar{c}}\,.$$

It follows from the definition that affine subsets $P_i^{\bar{c}}, P_i^{\bar{d}}$ are either disjoint or equal for all $\bar{c}, \bar{d}$. Consequently, $\hat{P}_i^{\bar{a}} = P_i^{\bar{c}}$ whenever $\bar{a} \in P_i^{\bar{c}}$. Hence, condition (1) is equivalent to:

$$\text{for all } i, \text{ for all } \bar{a} \in D,\ X \subseteq \hat{P}_i^{\bar{a}} \text{ or } |X \cap \hat{P}_i^{\bar{a}}| < \frac{|X|}{2k} \qquad (2)$$

and we can use $\hat{P}_i^{\bar{a}}$ instead of $P_i^{\bar{c}}$ in the search of X. Crucially, $\hat{P}_i^{\bar{a}}$ can be defined by a quantifier free formula: $\exists \bar{z}\, P_i(\bar{x}, \bar{z}) \wedge P_i(\bar{a}, \bar{z})$ is equivalent to the conjunction $C_i^{\bar{a}}(\bar{x})$ of all equalities over $\bar{x}$ and $\bar{a}$ entailed by $P_i(\bar{x}, \bar{z}) \wedge P_i(\bar{a}, \bar{z})$. Consequently, set $X_j = D \cap \bigcap_{\ell=0}^{j} \hat{P}_{i_\ell}^{\bar{a}_\ell}$ can be represented by the conjunction $\bigwedge_{\ell=0}^{j} C_{i_\ell}^{\bar{a}_\ell}(\bar{x})$. Hence, using at most k quadratic queries over D, we can test whether condition (2) holds for X_j, and if not, compute the representation of X_{j+1}. Since this is repeated at most r times, the total number of queries is $\mathcal{O}(rk)$. □

Theorem 2 can be used to show that also some inequality constraints can be incorporated into the framework, at a cost of inflating the kernel size and the number of queries (see Appendix F). Unfortunately, we are only able to handle inequalities between variables.

THEOREM 3. *Let $\alpha(\bar{x}, \bar{z})$ be a potential expression with k clauses, using inequality only over $\bar{x}$, and let $D \subseteq \mathbb{D}^r$, $r = |\bar{x}|$. Then there exists a kernel D' for D w.r.t. α of size $2 \cdot (2kr)^r$. Moreover, D' can be found by an algorithm making $\mathcal{O}(kr(2kr)^r \cdot \log|D|)$ quadratic calls to D.*

To complete the computation we can apply the brute force method to the obtained kernels. Let $\alpha_1, \alpha_2, \ldots, \alpha_k$ be potential expressions given by Lemma 4 for the kind $\mathcal{K}$. Let $kernel_i$

be the query implementing the algorithm above for the potential expression α_i (the query uses recursion and simple arithmetic; see Appendix H). The query $q_{\mathcal{M}}$ is obtained like in Sect. 4, with the subquery $const_{\mathcal{K}}$ modified by replacing q_{π_i} with $kernel_i$, and $values_{|\bar{z}|}$ defined as a query returning the set of all tuples of data values of length $|\bar{z}|$ with entries taken from $kernel_1, kernel_2, \ldots, kernel_k$. Observe that since there are no inequalities involving variables introduced on the target side, there is no need for the use of nulls in the valuations. The combined complexity of the resulting $q_{\mathcal{M}}$ is comparable to that of the query in Sect. 5.

THEOREM 4. *Let $\mathcal{M}$ be a mapping that contains no inequalities involving variables introduced on the target side. Then we can compute in time $2^{K \cdot poly(\|\mathcal{M}\|)}$ an implementing query $q_{\mathcal{M}}$ whose evaluation time over T is*

$$2^{K \cdot poly(\|\mathcal{M}\|)} \cdot |T|^{2r} \cdot \log |T| \,,$$

where $K = (2pb + b)^{2ph+h}$, b and h are the branching and height of $\mathcal{D}_t$, while p and r are the maximal size and arity of patterns in $\mathcal{M}$.

7. OPTIMIZING VIA SOURCE KINDS

In this section we propose a very different idea for optimization, based on splitting the source domain into kinds. This method works under the assumption that the mapping is *absolutely consistent*, i.e., each source tree has a solution.

By a *source kind* for a mapping $\mathcal{M}$ we mean a kind $\mathcal{K}$ such that $L(\mathcal{K}) \subseteq L(\mathcal{D}_s)$ and for each source-side pattern π, if $\pi(\bar{a})$ can be matched in a tree from $L(\mathcal{K})$ then it can be matched *neatly* in a tree from $L(\mathcal{K})$. Lemma 5 obviously holds also for source kinds, so we can compute source kinds $\mathcal{K}_1^s, \ldots, \mathcal{K}_k^s$ covering $L(\mathcal{D}_s)$. Our idea is based on the following theorem.

THEOREM 5 ([8]). *For absolutely consistent mapping $\mathcal{M}$, target kinds $\mathcal{K}_1^t, \ldots, \mathcal{K}_k^t$ covering $L(\mathcal{D}_t)$, and source kind $\mathcal{K}^s(\bar{c})$, there are i and $\bar{d}$ such that each tree in $L(\mathcal{K}^s(\bar{c}))$ has a solution in $L(\mathcal{K}_i^t(\bar{d}))$. Moreover, for each i, such $\bar{d}$ can be found in time $K^{K \cdot poly(\|\mathcal{M}\|)}$, where $K = \max(|\mathcal{K}^s|, |\mathcal{K}_i^t|)$, assuming $\|\mathcal{K}^s\| + \|\mathcal{K}^t\| = 2^{poly(\|\mathcal{M}\|)}$.*

Thus if we want to determine the right data kind for $T \in L(\mathcal{K}_i^s)$, the only interesting data values in T are those in the nodes corresponding to the ordinary nodes of $\mathcal{K}_i^s$. If we could compute these values, we would be done. In general, it may be difficult, but it becomes easy under the following additional assumption on the source kinds.

We write $T.v$ for the subtree of tree T rooted at node v. Thus, if $\mathcal{K}$ is a kind, so is $\mathcal{K}.v$. For siblings v_1, v_n in $\mathcal{K}$, language $L(\mathcal{K}, v_1, v_n)$ contains all forests that appear between siblings v_1, v_n (including v_1, v_n) in trees from $L(\mathcal{K})$. For a context port u, by P_u we denote the set of labels that can occur on the shortest root-to-port path in a context from L_u.

DEFINITION 4. *A kind $\mathcal{K}$ is* explicit *if:*

(1) no two forest ports are consecutive siblings and for each forest port u that has a next sibling, and each maximal sequence of nodes $u \to v_1 \to v_2 \to \ldots \to v_n$ such that v_i are not forest ports, for each $G \in L_u$ and each $F \in L(\mathcal{K}, v_1, v_n)$, no proper prefix of the word of root labels of $G + F$ contains the word of root labels of F;

(2) for each context port u there is a sequence of ordinary nodes $v_1 \downarrow v_2 \downarrow \ldots \downarrow v_n$ with $n > 1$, $u \downarrow v_1$, $lab(v_i) \in P_u$, such that for each node $v \notin \{u, v_1, v_2, \ldots, v_n\}$ in $\mathcal{K}.u$, $lab(v) \notin P_u$ and if v is a port then no element of L_v uses a label from P_u.

The following lemma shows that we can extract the interesting data values with path expressions.

LEMMA 8. *Let $\mathcal{K}$ be an explicit kind. For each tree $T \in L(\mathcal{K})$ there is a unique witnessing decomposition $(T_u)_u$. Moreover, for each ordinary node v in $\mathcal{K}$ there exists a path expression selecting in each tree from $L(\mathcal{K})$ the unique node corresponding to v in the witnessing decomposition. The size of the path expression is $\mathcal{O}(bh|\Gamma|)$, where h is the depth of the node v, and b is the maximal number of children of any node in $\mathcal{K}$. The expression can be computed in polynomial time.*

PROOF. We prove both claims simultaneously by induction on the height of $\mathcal{K}$. A kind of height 0 is an ordinary node or a tree port, and consequently it admits exactly one witnessing decomposition. The second part of the claim is trivial. Let us assume that the height of $\mathcal{K}$ is non-zero. We consider two cases depending on whether the root of $\mathcal{K}$ is an ordinary node or a context port (it cannot be a tree port or a forest port, because it is not a leaf).

Suppose the root of $\mathcal{K}$ is an ordinary node and let $v_1, v_2, \ldots, v_n$ be all its children, the forest ports among them being exactly $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ for some $i_1 < i_2 < \cdots < i_k$. Suppose that $T \in L(\mathcal{K})$ and let F be the forest obtained by cutting of the root of T. Clearly $F \in L(\mathcal{K}, v_1, v_n)$, so there is a decomposition of F into

$$F_1 + G_1 + F_2 + G_2 + \cdots + F_k + G_k + F_{k+1}$$

such that $G_j \in L_{v_{i_j}}$ and $F_j \in L(\mathcal{K}, v_{i_{j-1}+1}, v_{i_j - 1})$ with $i_0 = 0$, $i_{k+1} = n + 1$. An inductive argument using Definition 4 (1) shows that this decomposition is unique. Using the inductive hypothesis for $\mathcal{K}.v_j$ with $j \notin \{i_1, i_2, \ldots, i_k\}$ we obtain uniqueness of the witnessing decomposition for T.

Let us now move to the second part of the induction thesis. If v is the root of $\mathcal{K}$, the claim is trivial. Otherwise, v is contained in $\mathcal{K}.v_\ell$ for some ℓ satisfying $i_{m-1} < \ell < i_m$. It suffices to write a query that identifies the node $\tilde{v}_\ell$ in T corresponding to v_ℓ, and then use the inductive hypothesis to locate the node corresponding to v in the tree $T.\tilde{v}_\ell \in L(\mathcal{K}.v_\ell)$. Let α_j be the word of root labels of the forest F_j in the decomposition above. Note that this word is common for all forests in $L(\mathcal{K}, v_{i_{j-1}+1}, v_{i_j - 1})$. Indeed, the labels of ordinary nodes among $v_{i_{j-1}+1}, v_{i_{j-1}+2}, \ldots, v_{i_j - 1}$ are given, and for each tree port and context port u, all trees/contexts in L_u have the same label, fixed by the DTD representing L_u. By Definition 4, no proper prefix of the word of root symbols of a forest from $L_{v_{i_j}} + L(\mathcal{K}, v_{i_{j-1}+1}, v_{i_j - 1})$ contains α_{j+1} as an infix. Based on this we can locate in T the node corresponding to v_ℓ as follows: find the first occurrence of α_2 after α_1, and then the first occurrence of α_3 after that, etc., until α_m is found. This is done with a path expression

$$. \downarrow [\neg \leftarrow] \, \widehat{\alpha}_1 \to^+ \widehat{\alpha}_2 \to^+ \ldots \to^+ \widehat{\alpha}_m [\neg f] \leftarrow^p$$

$$\text{for} \quad f = \leftarrow \widehat{\alpha}_m^{-1} \leftarrow^+ \ldots \leftarrow^+ \widehat{\alpha}_2^{-1} \leftarrow^+ \widehat{\alpha}_1^{-1}$$

where p is such that $v_{i_{m-1}} \leftarrow^p v_\ell$ and for $\alpha = \sigma_1 \sigma_2 \ldots \sigma_q$, $\widehat{\alpha}$ is the expression $[\sigma_1] \to [\sigma_2] \to \ldots \to [\sigma_q]$ and $\widehat{\alpha}^{-1}$ is $[\sigma_q] \leftarrow [\sigma_{q-1}] \leftarrow \ldots \leftarrow [\sigma_1]$.

Suppose now that the root of $\mathcal{K}$ is a context port u. By Definition 4, there is a sequence of ordinary nodes $v_1 \downarrow v_2 \downarrow \ldots \downarrow v_n$ with $u \downarrow v_1$, $lab(v_i) \in P_u$, such that for each other node v in $\mathcal{K}.u$, $lab(v) \notin P_u$ and if v is a port then no element of L_v uses a label from P_u. Observe that no label from P_u can occur in any context from L_u outside of the shortest root-to-port path. Indeed, if this was the case, one could easily construct a multicontext with two ports conforming to the DTD defining L_u, which is forbidden by the definition of a context DTD. Hence, the set of P_u labelled nodes in each tree $T \in L(\mathcal{K})$ is a $\downarrow$-path. The last element of this path corresponds to v_n in each witnessing decomposition. From this it follows immediately that T is uniquely decomposed into $C \cdot T'$ such that $C \in L_u$ and $T' \in L(\mathcal{K}.v_1)$ (by the definition of multicontexts, v_1 is the unique child of u), and the unique decomposition for T follows by induction hypothesis for $T.v_1$. Moreover, in each $T \in L(\mathcal{K})$ we can identify the node $\tilde{v}_1$ corresponding to v_1 using the expression

$$. \downarrow^+ [(\sigma_1 \vee \sigma_2 \vee \cdots \vee \sigma_q) \wedge \neg \downarrow [\sigma_1 \vee \sigma_2 \vee \cdots \vee \sigma_q]] \uparrow^{n-1}$$

where $P_u = \{\sigma_1, \sigma_2, \dots, \sigma_q\}$. $\square$

Now, assuming that the source tree is in $L(\mathcal{K}^s)$ for some explicit source kind $\mathcal{K}^s$, we build a solution with query $q_{\mathcal{K}^s}$ obtained according to the general recipe for $q_{\mathcal{M}}$ (Sect. 4), using the following query $const_{\mathcal{K}^s,\mathcal{K}^t}$ instead of $values_{|\bar{z}|}$

$$\begin{aligned}&\mathsf{let}\ \bar{c} := const'_{\mathcal{K}^s}\ \mathsf{return}\\ &\quad \mathsf{if}\ \mathcal{E}_1(\bar{c})\ \mathsf{then}\ \bar{d}_1\ \mathsf{else}\ \dots\ \mathsf{if}\ \mathcal{E}_k(\bar{c})\ \mathsf{then}\ \bar{d}_k\,.\end{aligned}$$

The subquery $const'_{\mathcal{K}^s}$ selects the tuple of data values $\bar{c}$ stored in the copy of $\mathcal{K}^s$ in the input tree; it is obtained via Lemma 8. The tuple $\bar{d}_j$ is such that $\mathcal{K}^t(\bar{d}_j)$ is a suitable target data kind for the source data kind $\mathcal{K}^s(\bar{c})$ whenever $\mathcal{E}_j(\bar{c})$ holds; its entries come from $\bar{c}$ or a set of fresh nulls. Expressions $\mathcal{E}_j$ range over all equality types over $\bar{c}$ for which such a tuple $\bar{d}_j$ exists. The equality types $\mathcal{E}_j$ and the tuples $\bar{d}_j$ can be computed from $\mathcal{K}^s$ and $\mathcal{K}^t$ by Theorem 5.

Assuming $\|\mathcal{K}^s\| + \|\mathcal{K}^t\| = 2^{poly(\|\mathcal{M}\|)}$, the synthesis time for query $const_{\mathcal{K}^s,\mathcal{K}^t}$ is $K^{K \cdot poly(\|\mathcal{M}\|)}$ and the evaluation time over T is $K^{K \cdot poly(\|\mathcal{M}\|)} \cdot |T|$, where $K = \max(|\mathcal{K}^s|, |\mathcal{K}^t|)$. For $q_{\mathcal{K}^s}$ the respective bounds given by the general recipe are $K^{K \cdot poly(\|\mathcal{M}\|)}$ and $K^{K \cdot poly(\|\mathcal{M}\|)} \cdot |T|^r$, where $K = \max(|\mathcal{K}^s|, (2pb+b)^{2ph+h})$.

It remains to show that we can compute explicit source kinds covering $L(\mathcal{D}_s)$. To do this, we need to relax the conditions imposed on L_u for forest ports u. This modification does not influence Definition 4 or Lemma 8 at all, and Theorem 5 generalizes easily (see Appendix E).

DEFINITION 5 (m-KINDS). *The definition of m-kind is obtained by replacing the condition (1) in Definition 1 with*

(1') L_u is a DTD-definable set of forests and whenever $F + G + H \in L_u$ and G consists of at most m trees,

$$F' + G + H' + L_u \subseteq L_u$$

for some forests F', H'.

LEMMA 9. *For each mapping $\mathcal{M}$ there exist explicit source p-kinds $\mathcal{K}^s_1, \mathcal{K}^s_2, \dots, \mathcal{K}^s_n$ covering $L(\mathcal{D}_s)$, such that $|\mathcal{K}^s_i| \leq K$, $\|\mathcal{K}^s_i\| = \mathcal{O}(\|\mathcal{D}_s\| \cdot |\Gamma|^p)$, and they can be computed in time $2^{K \cdot poly(\|\mathcal{M}\|)}$; here $K = (3pb+b)^{2ph+h}$, b and h are the branching and height of $\mathcal{D}_s$, and p is the maximal size of source side patterns in $\mathcal{M}$.*

The proof can be found in Appendix D. In the notation of Lemma 9, $q_{\mathcal{M}}$ can be defined as

$$\mathsf{if}\ L(\mathcal{K}^s_1)\ \mathsf{then}\ q_{\mathcal{K}^s_1}\ \mathsf{else}\ \ \mathsf{if}\ L(\mathcal{K}^s_2)\ \mathsf{then}\ q_{\mathcal{K}^s_2}\ \mathsf{else}\ \dots$$

where $L(\mathcal{K}^s_i)$ stands for the Boolean test checking if the source tree is in $L(\mathcal{K}^s_i)$. As $\mathcal{K}^s_i$ can be easily converted to an equivalent tree automaton [22], this check can be done in XQuery. We obtain the following bounds.

THEOREM 6. *For each absolutely consistent mapping $\mathcal{M}$ one can compute in time $2^{K \cdot poly(\|\mathcal{M}\|)}$ an implementing query $q_{\mathcal{M}}$ whose evaluation time is $2^{K \cdot poly(\|\mathcal{M}\|)} \cdot |T|^r$; here $K = (3pb + b)^{2ph+h}$, b is the maximum of the branchings of $\mathcal{D}_s$ and $\mathcal{D}_t$, h is the maximum of the heights of $\mathcal{D}_s$ and $\mathcal{D}_t$, and p, r are the maximal size and arity of patterns in $\mathcal{M}$.*

8. TRACTABLE CASE

In this short section we present a combination of restrictions under which the transformation synthesis problem is tractable. In order to temper the expectations, let us recall that solutions are not polynomial in general. Typically, the solution will need to satisfy $\mathcal{O}(|T|^r)$ valuations of each target pattern. Moreover, the target DTD $\mathcal{D}_t$ enforces adding additional nodes, not specified by the patterns. For instance, each added node with a label σ, must come with a subtree conforming to the DTD $\langle \sigma, P_t \rangle$ where $\mathcal{D}_t = \langle \mathsf{r}, P_t \rangle$. This is reflected in the complexity bounds we obtain.

In *simple threshold DTDs* productions are of the form $\sigma \to \hat{\tau}_1 \hat{\tau}_2 \dots \hat{\tau}_n$, where $\tau_1, \tau_2, \dots, \tau_n$ are distinct labels from Γ and $\hat{\tau}$ is τ, $\tau? = (\tau + \varepsilon)$, τ^+, or τ^*.[2] A *fully-specified* pattern is connected, uses only child relation, all its nodes have labels (i.e., wildcard is not allowed), and some node is labelled with r, the root symbol of the target DTD.

THEOREM 7. *For mappings $\mathcal{M}$ using tree-shaped patterns only, with fully specified target-side patterns, and a simple threshold target DTD $\mathcal{D}_t = \langle \mathsf{r}, P_t \rangle$, one can compute in time $poly(\|\mathcal{M}\|)$ an implementing query $q_{\mathcal{M}}$ whose evaluation time over tree T is $poly(\|\mathcal{M}\|) \cdot N \cdot |T|^r$, where $N = \max_{\sigma \in \Gamma} \min_{S \in L(\langle \sigma, P_t \rangle)} |S|$ and r is the maximal arity of patterns.*

The proof can be found in Appendix I. Without changing the complexity one could allow the use of following-sibling and limited use of next-sibling on the target side, but for simple threshold DTDs it has rather limited use. The restriction to tree-shaped patterns can be lifted at the cost of a factor exponential in the size of the used patterns (cf. Lemma 1). If we allow more expressive target schemas or non-fully specified target patterns, the solution existence problem becomes NEXPTIME-complete [8]. Hence, Theorem 7 cannot be extended to these cases without showing NEXPTIME = EXPTIME.

9. CONCLUSIONS

We have shown that an implementing query can be constructed in the general case and we give two methods to build more efficient queries. Precise bounds on the constants are quite intractable in the general setting, but we believe they can be improved by heuristics tailored to the parameters of mappings arising in practise. For instance, it is reasonable to believe that the size of kinds will not be really large for the simple schemas prevailing in practical applications. It would be interesting to have a closer look at practical settings.

We work with DTDs, but the results of Sections 4–6 carry over to more expressive schema languages relying on tree automata. It would be interesting to see if the approach from Sect. 7 can be applied to such schemas as well.

One natural feature missing in our setting is key constraints. It seems plausible that our approach can be extended to handle unary keys in target schemas.

Another issue is the quality of the proposed transformation. A natural criterion is the evaluation time of the query over source tree,

[2] Simple threshold DTDs resemble nested-relational DTDs, except that the non-recursiveness restriction is lifted.

but other criteria could refer to the size and redundancy of the produced solution. Redundancy is closely related to universality of target instances, which is essential in evaluation of queries under the semantics of certain answers. For XML data, classical universal solutions usually do not exist [12], and more refined notions would be needed.

Finally, we point out a combinatorial challenge: is there a kernel of size $\mathcal{O}(k^{\mathcal{O}(r)})$ even if the potential expressions α_i can contain inequalities between constants and variables?

10. REFERENCES

[1] S. Amano, L. Libkin, F. Murlak. XML schema mapping. *PODS 2009*, 33–42.

[2] S. Amer-Yahia, S. Cho, L. Lakshmanan, D. Srivastava. Tree pattern query minimization. *VLDB J.* 11 (2002), 315–331.

[3] M. Arenas, P. Barceló, L. Libkin, F. Murlak *Relational and XML Data Exchange*. Morgan&Claypool Publishers, 2010.

[4] M. Arenas, L. Libkin. XML data exchange: consistency and query answering. *J. ACM* 55(2): (2008).

[5] P. Barceló. Logical Foundations of Relational Data Exchange. *SIGMOD Record 38,* 1 (2009): 49–58.

[6] Ph. A. Bernstein, S. Melnik, Model management 2.0: manipulating richer mappings. *ACM SIGMOD 2007*, 1–12.

[7] G. J. Bex, F. Neven, J. Van den Bussche. DTDs versus XML Schema: a practical study. *WebDB'04*, 79–84.

[8] M. Bojańczyk, L. A. Kołodziejczyk, F. Murlak. Solutions in XML data exchange. *ICDT 2011*, 102–113.

[9] H. Björklund, W. Martens, T. Schwentick. Conjunctive query containment over trees. *DBPL 2007*, 66–80.

[10] A. Church. Logic, arithmetic, and automata. Proc. Int. Congr. Math. 1962. Inst. Mittag-Leffler, Djursholm, Sweden, 1963, 23–35.

[11] C. David. Complexity of data tree patterns over XML documents. *MFCS 2008*, 278–289.

[12] C. David, L. Libkin, F. Murlak. Certain answers for XML queries. *PODS 2010*, 191–202.

[13] R. G. Downey, M. R. Fellows. *Parameterized Complexity*. Springer, 1999.

[14] R. Fagin, L. Haas, M. Hernandez, R. Miller, L. Popa, Y. Velegrakis Clio: Schema mapping creation and data exchange. In *Conceptual Modeling: Foundations and Applications*, Essays in Honor of John Mylopoulos. LNCS vol. 5600. Springer-Verlag, 2009, 198–236.

[15] J. Flum, M. Grohe. *Parameterized Complexity Theory*. Springer, 2006.

[16] G. Gottlob, C. Koch, K. Schulz. Conjunctive queries over trees. *J. ACM* 53 (2006), 238–272.

[17] H. Jiang, H. Ho, L. Popa, W.-S. Han. Mapping-driven XML transformation. *WWW 2007*, 1063–1072.

[18] P. G. Kolaitis. Schema Mappings, Data Exchange, and Metadata Management. *PODS 2005*, 61–75.

[19] S. Langerman, P. Morin. Covering Things with Things. *Discrete & Computational Geometry*, 33(4) (2005), 717–729.

[20] B. Marnette, G. Mecca, P. Papotti, S. Raunich, D. Santoro. ++Spicy: an opensource tool for second-generation schema mapping and data exchange. *PVLDB 4,* 12 (2011), 1438–1441.

[21] S. Melnik, A. Adya, Ph. A. Bernstein. Compiling mappings to bridge applications and databases. *ACM Trans. Database Syst.* 33 (4), 2008.

[22] F. Neven. Automata Theory for XML Researchers. *SIGMOD Record* 31(3): 39-46 (2002).

[23] L. Popa, Y. Velegrakis, R. Miller, M. Hernández, R. Fagin. Translating web data. *VLDB 2002*, 598–609.

APPENDIX

A. NEAT MATCHINGS ARE PRESERVED UNDER COMBINATIONS

Lemma 2 follows immediately from the definition of combinations and the following more general property.

LEMMA 10. *If $\pi(\bar{a})$ is matched neatly in $T \in L(\mathcal{K})$ with respect to a decomposition $(T_u)_u$, then $\pi(\bar{a})$ is matched (neatly) in any tree $T' \in L(\mathcal{K})$ with decomposition $(T'_u)_u$ such that*

- *for all forest ports u, $T'_u = F + T_u + F'$ for some forests F, F',*
- *for all context ports u, $T'_u = C \cdot T_u \cdot C'$ for some contexts C, C',*
- *for all tree ports u, $T'_u = C \cdot T_u$ for some context C.*

PROOF. It is easy to see that a neat homomorphism from $\pi(\bar{a})$ to T witnessing the decomposition $(T_u)_u$ is also a neat homomorphism from $\pi(\bar{a})$ to any T' with such a decomposition $(T'_u)_u$. □

B. CONSTRUCTING SOLUTIONS OF A GIVEN KIND

We first prove an auxiliary lemma showing how to construct witnesses for neat matchings.

LEMMA 11. *For any data kind $\mathcal{K}(\bar{c})$, any pure pattern π, and a tuple $\bar{a}$, either $\pi(\bar{a})$ cannot be matched neatly in any tree from $\mathcal{K}(\bar{c})$ or can be matched neatly in a tree $T \in \mathcal{K}(\bar{c})$ with a witnessing decomposition $(T_u)_u$ such that $|T_u| \leq |\pi| \cdot b^{\mathcal{O}(h)}$, where b and h are the maximal branching and height of DTDs representing languages L_u in $\mathcal{K}$. Moreover, T, $(T_u)_u$ and a neat homomorphism $\pi(\bar{a}) \to T$ can computed in time $\|\mathcal{K}\|^{\mathcal{O}(bh|\pi|^2)} \cdot |\mathcal{K}|^{|\pi|+1}$.*

PROOF. Consider all possible ways of mapping vertices of π to nodes of $\mathcal{K}$. There is $|\mathcal{K}|^{|\pi|}$ choices, and we check them one by one. Fix one mapping and let V_u denote the set of vertices of π mapped to a port u. First check that the mapping respects the labelling and data values in the ordinary nodes of $\mathcal{K}$ and that it does not violate any edge in π:

- each relation edge have to be preserved, unless both ends are mapped to the same port;
- if x is mapped to a context or tree port, it cannot be connected with $\to, \to^+, \downarrow$ with any node mapped elsewhere;
- if x is mapped to a forest port, it cannot be connected with $\to$ to any node mapped elsewhere.

This check can be done in time $|\pi|^2 \cdot |\mathcal{K}|$. If it succeeds, we can move on to filling in the ports of $\mathcal{K}$. This can be done independently for each port. For most ports u, V_u is empty and we can fill u with any compatible forest/context T_u. This can be done in time $\mathcal{O}(|\mathcal{K}| \cdot b^h)$. Then there are at most $|\pi|$ ports left to fill, and for each of them we need to make sure that $\pi \restriction_{V_u}$ (π restricted to V_u) can be satisfied in a compatible T_u in a way that gives a matching of π in the constructed tree T.

Assume that u is a tree port and let $\mathcal{D}$ be a DTD representing L_u, let T_u be a tree conforming to $\mathcal{D}$, and let μ be a homomorphism from $\pi \restriction_{V_u}$ to T_u. The *support* of μ is the set of nodes of T_u that can be reached from the image of V_u by going up, left and right. A simple pumping argument shows that we can assume that the support has size $4bh|V_u|$. The algorithm can iterate over all trees U of size at most $4bh|V_u|$ and all homomorphisms μ from $\pi \restriction_{V_u}$ to U in time $|\Gamma|^{\mathcal{O}(bh|V_u|^2)}$. Testing that μ does not violate edges of π with only one endpoint in V_u can be done in time polynomial in the size of U and π. Completing U to a tree conforming to $\mathcal{D}$ is easy: just replace each leaf labelled with σ with the smallest tree conforming to $\mathcal{D}_\sigma$, i.e., the DTD obtained by replacing the root symbol of $\mathcal{D}$ with σ. The size of such tree can be bounded by b^h. If this procedure succeeds, we obtain a tree T_u of size $\mathcal{O}(bh|V_u| \cdot b^h) = |\pi| \cdot b^{\mathcal{O}(h)}$ in time $|\Gamma|^{\mathcal{O}(bh|V_u|^2)} \cdot \mathit{poly}(|\pi|, \|\mathcal{D}\|, b^h) = \|\mathcal{K}\|^{\mathcal{O}(bh|\pi|^2)}$. The data values in T_u that were not determined by the mapped vertices of π can be set to fresh nulls.

For forest ports and context ports the argument is analogous and the bounds are the same. Altogether the procedure takes time $\Big(\|\mathcal{K}\|^{\mathcal{O}(bh|\pi|^2)} + |\pi|^2 \cdot |\mathcal{K}|\Big) \cdot |\mathcal{K}|^{|\pi|} = \|\mathcal{K}\|^{\mathcal{O}(bh|\pi|^2)} \cdot |\mathcal{K}|^{|\pi|+1}$. □

We are now ready to prove Lemma 3.

LEMMA 3. *For each mapping $\mathcal{M}$ and target kind $\mathcal{K}$ there is a query $sol_{\mathcal{K}}(\bar{z})$ such that for each tree T that admits a solution in $L(\mathcal{K}(\bar{c}))$, $sol_{\mathcal{K}}(\bar{c})(T)$ is a solution for T.*
The synthesis time for $sol_{\mathcal{K}}$ is $2^{poly(\|\mathcal{K}\|, \|\mathcal{M}\|)} \cdot |\mathcal{K}|^{\mathcal{O}(p+r)}$ and the evaluation time is $2^{poly(\|\mathcal{M}\|, \|\mathcal{K}\|)} \cdot |\mathcal{K}|^{r+1} \cdot |T|^r$ where r and p are the maximal arity and size of patterns in $\mathcal{M}$.

PROOF. Notice the kind $\mathcal{K}$ we consider is a target kind. Thus, if a target pattern can be matched in a tree from $L(\mathcal{K})$, it can be matched neatly in a tree from $L(\mathcal{K})$. The main idea of the query is to build a solution by filling ports of $\mathcal{K}$ with combined pieces of trees each of them satisfying (neatly) a different target constraint. Lemma 10 ensures that the final tree satisfies all the constraints.

Let $\pi(\bar{x}), \eta(\bar{x}) \to \pi'(\bar{x}, \bar{y}), \eta'(\bar{x}, \bar{y})$ be a dependency from $\mathcal{M}$. For a tree $T \in L(\mathcal{K}(\bar{c}))$, a witnessing decomposition $(T_u)_u$, and a homomorphism $\mu \colon \pi' \to T$, we define the *trace of* μ as the conjunction of equalities and inequalities between $\bar{x}$ and $\bar{c}$ induced by μ, defined as follows. Consider first the conjunction $\alpha(\bar{x}, \bar{y})$ that contains the equality $z = c_i$ if z is mapped to the node of $\mathcal{K}$ storing c_i, and equality $z = z'$ if z and z' are mapped to the same node outside of $\mathcal{K}$. The trace of μ is the projection of $\alpha(\bar{x}, \bar{y}) \wedge \eta'(\bar{x}, \bar{y})$ to $\bar{x}$ and $\bar{c}$, i.e., a conjunction of equalities and inequalities $\mathcal{E}(\bar{x})$ such that for all $\bar{c}$ and $\bar{a}$, $\mathcal{E}(\bar{a}) \iff \exists \bar{b}\ \alpha(\bar{a}, \bar{b}) \wedge \eta'(\bar{a}, \bar{b})$.

Consider all possible traces $\mathcal{E}(\bar{x})$ of neat matchings of $\pi'(\bar{x}, \bar{y})$ in trees from $L(\mathcal{K}(\bar{c}))$. The number of traces is at most $|\mathcal{K}|^r \cdot r^r$. For each trace $\mathcal{E}$ compute a tree $T_{\mathcal{E}}$, decomposition $(T_{\mathcal{E}})_u$, and a neat homomorphism $\mu_{\mathcal{E}} \colon \pi' \to T_{\mathcal{E}}$ yielding $\mathcal{E}$. This can be done in time

$2^{poly(\|\mathcal{K}\|,|\pi'|)} \cdot |\mathcal{K}|^{|\pi'|+1}$ by Lemma 11. Choose $T_\mathcal{E}$ such that outside of $\mathcal{K}$ the data values of $T_\mathcal{E}$ are distinct nulls, except when η' enforces equality between two nulls, or a null and some c_i.

For a tuple $\bar{a}$ satisfying $\mathcal{E}$ we write $T_\mathcal{E}(\bar{a})$ for the tree obtained from $T_\mathcal{E}$ by substituting each occurrence of the data value $\mu_\mathcal{E}(x_i)$ with a_i. Note that if $\mu_\mathcal{E}(x_i)$ is one of the constants c_i, the operation will have no effect at all, as $\mathcal{E}$ enforces that in this case $a_i = c_j$. Clearly, $T_\mathcal{E}(\bar{a}) \models \pi'(\bar{a}, \bar{b}), \eta'(\bar{a}, \bar{b})$ for some $\bar{b}$.

The query $sol_\mathcal{K}(\bar{z})$ combines the trees $T_\mathcal{E}(\bar{a})$ for all tuples $\bar{a}$ returned by $\pi(\bar{x}), \eta(\bar{x})$ evaluated on the source tree, and $\mathcal{E}$ such that $\bar{a}$ satisfies $\mathcal{E}$. The query guarantees that the condition in Lemma 10 is satisfied and thus satisfiability of the target constraints is preserved. Let us first describe the subqueries q_u for each port of $\mathcal{K}$.

Let us assume that u is a *forest port*, and let $\mathcal{D}$ be the forest DTD representing L_u. We first construct a query $q_{u,\mathcal{E}}(\bar{x})$ such that $q_{u,\mathcal{E}}(\bar{a})$ returns $(T_\mathcal{E}(\bar{a}))_u + F$, for some forest F such that $(T_\mathcal{E}(\bar{a}))_u + F + L_u \subseteq L_u$. The existence of such F is guaranteed by the condition imposed on L_u in the definition of kinds. A standard pumping argument shows that the size of F can be bounded by $2^{poly(\|\mathcal{K}\|)}$, so the evaluation time of $q_{u,\mathcal{E}}(\bar{x})$ is $p \cdot 2^{poly(\|\mathcal{K}\|)}$.
The query q_u is obtained by concatenating the queries

$$\textsf{for } \bar{x} \textsf{ in } q_{\pi,\eta\wedge\mathcal{E}} \textsf{ return } q_{u,\mathcal{E}}(\bar{x})$$

for $\pi(\bar{x}), \eta(\bar{x}) \to \pi'(\bar{x}, \bar{y}), \eta'(\bar{x}, \bar{y})$ ranging over dependencies of $\mathcal{M}$ and $\mathcal{E}$ ranging over all possible traces of neat matchings of π', η', followed by a query returning a small forest from L_u. Since the evaluation time of $q_{\pi,\eta\wedge\mathcal{E}}$ is $|\eta \wedge \mathcal{E}| \cdot 2^{poly(|\pi|)} \cdot |T|^r$, the evaluation time of q_u is then $2^{poly(\|\mathcal{M}\|,\|\mathcal{K}\|)} \cdot |T|^r \cdot |\mathcal{K}|^r$.

For *tree ports* and *context* ports the construction is similar and the bounds are the same. The query $q_{u,\mathcal{E}}(\bar{x})$ returns contexts $C(\,\circ\,, (T_\mathcal{E}(\bar{a}))_u)$ or $C \cdot (T_\mathcal{E}(\bar{a}))_u \cdot C'$, accordingly, and q_u is obtained by concatenating vertically the queries

$$\textsf{for } \bar{x} \textsf{ in } q_{\pi,\eta\wedge\mathcal{E}} \textsf{ returnC } q_{u,\mathcal{E}}(\bar{x})$$

for all $\pi(\bar{x}), \eta(\bar{x}) \to \pi'(\bar{x}, \bar{y}), \eta'(\bar{x}, \bar{y})$ and $\mathcal{E}$, followed by a query outputting a small element of L_u, a tree or a context, depending on the type of the port u.

To get $sol_\mathcal{K}$, plug in at each port u of $\mathcal{K}$ the query q_u. To verify that $sol_\mathcal{K}(\bar{c})$ returns a solution, if there is one, observe that for each $\bar{a}$ returned by $\pi(\bar{x}), \eta(\bar{x})$, the target constraint $\pi'(\bar{a}, \bar{b}), \eta'(\bar{a}, \bar{b})$ must be satisfiable in a tree from $L(\mathcal{K}(\bar{c}))$ for some $\bar{b}$. By the definition of target kinds, it must also be matched neatly in some tree from $L(\mathcal{K}(\bar{c}))$. By construction of the query and Lemma 10 the output of the query satisfies every target constraints. The complexity bounds follow easily. □

LEMMA 4. *Let $\mathcal{M}$ be a mapping with dependencies $\pi_i(\bar{x}_i) \to \pi'_i(\bar{x}_i, \bar{y}_i)$ for $i = 1, 2, \ldots, n$ and let $\mathcal{K}$ be a target kind with m ordinary nodes. There exist $\alpha_i(\bar{x}_i, \bar{z})$ such that*

- *$\alpha_i(\bar{x}_i, \bar{z})$ is a disjunction of at most $|\mathcal{K}|^r r^r$ conjunctions of $\mathcal{O}(|\pi'_i|)$ equalities and inequalities among $\bar{x}_i$ and $\bar{z}$, where r is the maximal arity of patterns in $\mathcal{M}$;*
- *for each $\bar{c}$, each source tree T admits a solution in $L(\mathcal{K}(\bar{c}))$ iff $T \models \pi_i(\bar{a})$ implies $\alpha_i(\bar{a}, \bar{c})$ for all i.*

The α_i's can be computed from $sol_\mathcal{K}$ in polynomial time.

PROOF. The claim follows immediately from the proof of Lemma 3: α_i is the disjunction of all possible traces of neat matchings of π'_i in trees from $L(\mathcal{K})$. □

C. COVERING TARGET DOMAIN

Lemma 5 essentially follows from [8], where a similar result is proved for mappings between schemas given with tree automata. Here we give a sketch, for the convenience of the reader. The proof is based on the notion of margins, given in Definition 6 below, which are areas of ordinary nodes around the ports, that enable rearranging the matchings of patterns.

For the purpose of Definition 6, it is convenient to extend the notion of kind in such a way that it can define forests and contexts. A *forest kind* is simply a sequence of kinds; it naturally defines a language of forests. A *context kind* is a kind whose exactly one leaf port is annotated with $\perp$ instead of some language L_u; it defines the set of contexts obtained by legal substitutions at all ports except the one annotated with $\perp$.

We also introduce the following notation. For two siblings $v \to^* w$ in $\mathcal{K}$, $L(\mathcal{K}, v, w)$ stands for the language defined by the forest kind obtained by concatenating the subtrees of $\mathcal{K}$ rooted as the subsequent siblings between v and w. Similarly for $v \downarrow^+ w$, $L(\mathcal{K}, v, w)$ denotes the set of contexts defined by the context kind obtained by taking the subtree of $\mathcal{K}$ rooted at v and replacing the subtree rooted at v with a port marked with $\perp$.

DEFINITION 6. *A kind $\mathcal{K}$ has margins of size m if for each port u*

(1) if u is a forest port, then there exist v, w such that $v \to^m u$, $u \to^m w$, the only port among the segment of siblings from v to w is u, and

$$F + L(\mathcal{K}, v, w) + F' \subseteq L_u$$

for some forests F, F';

(2) if u is a tree port, then there exists v such that $v \downarrow^m u$, the only port on the shortest path from v to u is u, and

$$C \cdot L(\mathcal{K}.v) \subseteq L_u$$

for some context C;

(3) if u is a context port, then there are nodes v, w such that $v \downarrow^m u$ and $u \downarrow^{m+1} w$, the only port on the shortest path from v to w is u, and

$$C \cdot L(\mathcal{K}, v, w) \cdot C' \subseteq L_u$$

for some contexts C, C'.

LEMMA 12. *Let π be a pattern of size p and let $\mathcal{K}$ be a kind with margins of size p. If $\pi(\bar{a})$ is satisfiable in a tree from $L(\mathcal{K})$, then there exists $T \in L(\mathcal{K})$ and a witnessing decomposition $(T_u)_u$ such that $\pi(\bar{a})$ can be matched neatly in T.*

PROOF. Let S be any tree in which $\pi(\bar{a})$ is matched. Define T by substituting at port u the forest/context T_u defined as

$$F + S.(v_u, w_u) + F' \,, \qquad C \cdot (S.v_u) \,, \qquad \text{or} \qquad C \cdot (S.v_u \setminus S.w_u) \cdot C' \,,$$

depending on the character of the port u, where v_u, w_u are the nodes in S corresponding to the nodes in $\mathcal{K}$ guaranteed by Definition 6, and F, F', C, and C, C' are the appropriate forests or context, again guaranteed by Definition 6.

In the formulas above by $S.(v_u, w_u)$ we mean the forest obtained by taking the sequence of trees rooted at the sequence of consecutive siblings beginning with v_u and ending in w_u; by $S.v_u \setminus S.w_u$ we denote the context obtained from $S.v_u$ (the subtree of S rooted at v_u) by replacing the subtree rooted at w_v with a port.

Assume that u is a tree port. By pigeon-hole principle, if π is matched in such a way that it touches S_u, one can find a node u' on the shortest path between u and v_u, such that no node of π is matched to u'. It is easy to see that one can find a set of vertices of π, containing all those mapped to S_u, that is connected to other vertices of π only in such a way, that one can "move" the image of this whole set in to the copy of S_u contained in T_u, without violating the relations in π. Identical argument applies to context and forest ports. The matching (homomorphism) obtained this way is neat. □

Lemma 5 now follows from the following fact.

LEMMA 13. *For each mapping $\mathcal{M}$ there exist kinds $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_k$ with margins of size p, covering $L(\mathcal{D}_t)$, such that $L(\mathcal{K}_i) \subseteq L(\mathcal{D}_t)$, $|\mathcal{K}_i| \leq K = (2pb + b)^{2ph+h}$, where b and h are the branching and height of $\mathcal{D}_t$, and p is the maximal size of target side patterns in $\mathcal{M}$. Moreover, DTDs representing languages L_u in all $\mathcal{K}_i$ have branching and height bounded by the branching and height of $\mathcal{D}_t$, $n \leq \|\mathcal{D}_t\|^{2K}$, and $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_k$ can be computed in time $\|\mathcal{D}_t\|^{\mathcal{O}(K)}$.*

This fact was proved in [8], for the setting in which tree automata are used instead of DTDs. The claim carries over to DTDs immediately. The only delicate issue is the size of the root expressions in the forest DTDs representing L_u for forest ports u. The proof involves computing left and right quotients of these languages, by words of length m. This operation is costly for regular expressions, but very cheap for NFAs. For this purpose, in the kinds we represent all regular languages with NFAs. This does not cause any loss of generality, since a standard representation with regular expressions can be turned into one with NFAs (of linear size) in polynomial time. Moreover, kinds are only used inside our computations, so the NFAs never need to be converted back to regular expressions. In this representation, the branching of a DTD is the maximal number of states of the automata representing the productions and the root language.

D. COVERING THE SOURCE DOMAIN WITH EXPLICIT KINDS

The proof goes again via the notion of margins, but we have to redefine them for the extended kinds.

DEFINITION 7 (MARGINS FOR m-KINDS). *The definition of* m-kind with margins *(of size m) is obtained by replacing condition (1) in Definition 6 with*

(1') if u is a forest port, then there exist v, w such that $v \rightarrow^m u$, $u \rightarrow^m w$, u is the only port among the segment of siblings from v to w, and whenever $F + G + H \in L(\mathcal{K}, v, w)$ and G consists of at most m trees,

$$F' + G + H' \in L_u$$

for some forests F', H'.

The following lemma is a variant of Lemma 12 for m-kinds.

LEMMA 14. *Let π be a pattern of size p and let $\mathcal{K}$ be a p-kind with margins of size p. If $\pi(\bar{a})$ is satisfiable in a tree from $L(\mathcal{K})$, then there exists $T \in L(\mathcal{K})$ and a witnessing decomposition $(T_u)_u$ such that $\pi(\bar{a})$ can be matched neatly in T.*

PROOF. Let S be any tree in which $\pi(\bar{a})$ is matched, and let $\mu \colon \pi(\bar{a}) \to S$ be the witnessing homomorphism. The construction of T given in Lemma 12 has to be modified only for forest ports.

Let u be a forest port in $\mathcal{K}$, and let v_u, w_u be the nodes of S corresponding to the nodes of $\mathcal{K}$ guaranteed by Definition 7. In order to define T_u we first look at the image of π under μ within the forest $S.(v_u, w_u)$. By the pigeon-hole principle, between any $p+1$ subsequent roots in $S.(v_u, w_u)$ we can find a root that is not in the image of π. It follows, that we can decompose $S.(v_u, w_u)$ into nonempty forests $F_1 + F_2 + \cdots + F_k$, each consisting of at most p roots, such that for all vertices x, y of π, if $\mu(x) \in F_i$ and $\mu(y) \in F_j$ for $i < j$, then the only kind of edge between x and y that π can contain is $E_f(x, y)$. (If such an edge exists, x and y must be mapped to some roots of F_i and F_j, respectively.)

By Definition 7, we can easily construct a forest

$$T_u = G_1 + F_2 + G_2 + F_3 + G_4 + \cdots + G_{k-1} + F_{k-1} + G_k \in L_u \,.$$

Now we move the matching of π from $F_2 + F_3 + \cdots + F_{k-1}$ to T_u: for each $i = 2, 3, \ldots, k-1$, we move the image of vertices mapped to F_i to the copy of F_i contained in T_u. This gives a (partial) neat matching: since the forests F_1 and F_k are not moved, no edge of π is violated and the conditions for neat matching are satisfied.

The same procedure is carried out for all forest ports. For context and tree ports, we apply the simpler procedure described in the proof of Lemma 12. It is easy to see that the partial homomorphisms are compatible and together give a neat homomorphism. $\square$

Given the lemma above, we obtain Lemma 9 immediately from the following fact.

LEMMA 15. *For each mapping $\mathcal{M}$ there exist explicit p-kinds with margins $\mathcal{K}_1^s, \mathcal{K}_2^s, \ldots, \mathcal{K}_n^s$ covering $L(\mathcal{D}_s)$, such that $L(\mathcal{K}_i^s) \subseteq L(\mathcal{D}_s)$, $|\mathcal{K}_i^s| \le K = (3pb+b)^{2ph+h}$, where b is the maximal size of regular expressions used in $\mathcal{D}_s$, h is the maximal number of different labels on a tree from $L(\mathcal{D}_s)$, and p is the maximal size of source side patterns in $\mathcal{M}$. Moreover, DTDs representing languages L_u in all $\mathcal{K}_i^s$ have size $\mathcal{O}(\|\mathcal{D}_s\| + b \cdot |\Gamma|^p)$, $n \le \|\mathcal{D}_s\|^{\mathcal{O}(pK)}$, and $\mathcal{K}_1^s, \mathcal{K}_2^s, \ldots, \mathcal{K}_n^s$ can be computed in time $\|\mathcal{D}_t\|^{\mathcal{O}(pK)}$.*

The key point of the proof of Lemma 15 is performing the split for kinds that are essentially words: trees of height one, whose root is an ordinary node. The technical argument needed is expressed in terms of words in Lemma 19, to which the following definitions and lemmas lead. The proof of Lemma 15 is given afterwards.

DEFINITION 8. *For $L \subseteq \Gamma^*$ and a natural number m, we define $[L]_m$ as the set of infixes of length at most m of words from L, i.e,*

$$[L]_m = \Big\{ v \in \Gamma^{\le m} \;\big|\; \exists u\; \exists w \;\; uvw \in L \Big\} \; .$$

We write $[w]_m$ instead of $\big[\{w\}\big]_m$.

DEFINITION 9. *A language L is m-repeatable if for all k and all $v_1, v_2, \ldots, v_k \in [L]_m$ there exist $u_0, u_1, \ldots, u_k$ such that $u_0 v_1 u_1 v_2 u_2 \ldots v_k u_k \in L$.*

DEFINITION 10. *An m-*frame *F is an expression of the form*

$$w_0 L_0 w_1 L_1 \ldots w_n L_n w_{n+1} \; ,$$

where $|w_i| \ge m$ for all i and each L_j is regular, m-repeatable, and satisfies

$$[\mathsf{suf}_m(w_j) L_j \mathsf{pref}_m(w_{j+1})]_m \subseteq [L_j]_m \; ,$$

where by $\mathsf{suf}_m(u)$ and $\mathsf{pref}_m(u)$ we denote the sufix and prefix of u of length m. The length of F is $n+1$. For the sake of convenience, an m-frame of length 0 is a word w_0.

DEFINITION 11. *A frame $F = w_0 L_0 w_1 L_1 \ldots w_n L_n w_{n+1}$ is* explicit *if $[w_{i+1}]_m \not\subseteq [L_i]_m$ for all $i < n$.*

LEMMA 16. *Let $M \subseteq \Gamma^*$ be an m-repeatable regular language and let $v \in \Gamma^{\ge m}$ be a word such that $[M \mathsf{pref}_m(v)]_m \subseteq [M]_m$ but $[v]_m \not\subseteq [M]_m$. Then for every $u \in M$ no proper prefix of uv contains v as an infix, i.e., there is exactly one occurrence of v in uv.*

PROOF. Since $[v]_m \not\subseteq [M]_m$, we can present v as $v = xyz$ such that $y \notin [M]_m$, and no proper prefix of xy contains a word in $\Gamma^{\le m} - [M]_m$. Since $[M\mathsf{pref}_m(v)]_m \subseteq [M]_m$, we have $|xy| > m$.

Towards a contradiction, assume that there exists $u'u'' \in M$, with $u'' \neq \epsilon$ such that $u'v$ is a prefix of $u'u''v$. It follows that xy is a prefix of $u''xy$. If $|u''| + m \ge |xy|$, $u''\mathsf{pref}_m(xy)$ contains $y \notin [M]_m$, which contradicts the fact that $[M\mathsf{pref}_m(v)]_m \subseteq [M]_m$. Hence, $|u''| + m < |xy|$, and since $|y| \le m$ we have $|u''| < |x|$. Now, since $u'' \neq \varepsilon$, a proper prefix of xy contains y, which is a contradiction. $\square$

LEMMA 17. *For each explicit m-frame $F = v_0 M_0 v_1 M_1 \ldots v_n M_n v_{n+1}$ and each word w in F there exist unique words $u_0 \in M_0, u_1 \in M_1, \ldots, u_n \in M_n$ such that*

$$w = v_0 u_0 v_1 u_1 \ldots v_n u_n v_{n+1} \; .$$

Moreover, no proper prefix of $u_i v_{i+1}$ contains v_{i+1}.

PROOF. By induction on n. If $n = 0$, the claim is straightforward. Suppose $n > 0$ and assume that $w = v_0 u_0 v_1 u_1 \ldots v_n u_n v_{n+1}$ and $w = v_0 u_0' v_1 u_1' \ldots v_n u_n' v_{n+1}$ for some $u_i, u_i' \in M_i$. Suppose that $|u_0| \le |u_0'|$. By Lemma 16, $u_0 = u_0'$. We obtain the main claim of the lemma by invoking the induction hypothesis for w' and $v_1 M_1 \ldots v_n M_n v_{n+1}$ where $w = v_0 u_0 w'$. The additional claim follows directly from Lemma 16. $\square$

LEMMA 18. *Let $F = v_0 M_0 v_1 M_1 \ldots v_n M_n v_{n+1}$ be an m-frame such that for each $i > 0$ the language M_i is recognized by an NFA with a single strongly connected component (SCC) and let the maximal size of these NFAs be k.*

Then F can be presented as a union of at most $((nk+3) \cdot |\Gamma|^{m+1})^n$ explicit m-frames of length at most $n+1$, each beginning with v_0 and represented with NFAs of size at most $k \cdot |\Gamma|^m$.

Moreover, the explicit frames can be computed in time polynomial in $((nk+3) \cdot |\Gamma|^{m+1})^n$.

PROOF. We proceed by induction on n. If $n = 0$, we are done. Suppose that $n > 0$.

Assume that $[v_1]_m \not\subseteq [M_0]_m$. By the inductive hypothesis we present $v_1 M_1 \ldots v_n M_n v_{n+1}$ as the union of explicit m-frames $G_1 \cup G_2 \cup \cdots \cup G_p$. Replacing G_i with $v_0 M_0 G_i$ we obtain a presentation of F as a union of explicit m-frames.

The remaining case is $[v_1]_m \subseteq [M_0]_m$. We shall now organize the words $w \in M_1$ into four sets according to the first occurrence of a word from $[M_1]_m - [M_0]_m$ in $\mathsf{suf}_m(v_1)\, w\, \mathsf{pref}_m(v_2)$:

- none at all,
- within $v_1\,\mathsf{pref}_m(w)$,
- within w,
- within $\mathsf{suf}_m(w)v_2$.

Let $\mathcal{A}$ be the automaton with a single SCC recognizing M_1. Then M_1 can be written as the union of the following sets:

- $\big\{w \in M_1 \mid \big[\mathsf{suf}_m(v_1)\, w\, \mathsf{pref}_m(v_2)\big]_m \subseteq [M_0]_m\big\}$;
- $u\big(u^{-1}M_1\big)$ for $u \in \Gamma^m$ such that $[v_1u]_m \not\subseteq [M_0]_m$;
- $\Big\{w \in L(A^{\{q\}}) \mid \big[\mathsf{suf}_m(v_1)\, w\, \mathsf{pref}_m(u)\big]_m \subseteq [M_0]_m\Big\}\, u\big(u^{-1}L(\mathcal{A}_{\{q\}})\big)$ for $q \in Q^{\mathcal{A}}$, $u \in \Gamma^{m+1}$ such that $[u]_m \not\subseteq [M_0]_m$;
- $\big\{w \in M_1u^{-1} \mid \big[\mathsf{suf}_m(v_1)\, w\, \mathsf{pref}_m(u)\big]_m \subseteq [M_0]_m\big\}\, u$ for $u \in \Gamma^m$ such that $[uv_2]_m \not\subseteq [M_0]_m$.

In consequence, we can present F as a union of expressions obtained by replacing M_1 with one of the sets above. We shall deal with each such expression separately.

The first set, $M_1' = \big\{w \in M_1 \mid \big[\mathsf{suf}_m(v_1)\, w\, \mathsf{pref}_m(v_2)\big]_m \subseteq [M_0]_m\big\}$, gives an expression that can be written as

$$v_0\tilde{M}_0v_2M_2\dots v_nM_nv_{n+1} \tag{3}$$

where $\tilde{M}_0 = M_0v_1M_1'$. Combining the facts that $[v_1]_m \subseteq [M_0]_m$, and that F has margins, we obtain $[\tilde{M}_0]_m = [M_0]_m$, which implies that $\tilde{M}_0$ is m-repeatable. As $|v_1| \geq m$,

$$\begin{aligned}\big[\mathsf{suf}_m(v_0)\, \tilde{M}_0\, \mathsf{pref}_m(v_2)\big]_m = \big[\mathsf{suf}_m(v_0)\, M_0\, \mathsf{pref}_m(v_1)\big]_m \cup [v_1]_m \cup{}& \\ \cup \big[\mathsf{suf}_m(v_1)\, M_1'\, \mathsf{pref}_m(v_1)\big]_m \subseteq{}& \\ \subseteq [M_0]_m = [\tilde{M}_0]_m\,.&\end{aligned}$$

Hence, the expression (3) is an m-frame (shorter then F) and we conclude by the induction hypothesis.

The second kind of set, $u\big(u^{-1}M_1\big)$ with $u \in \Gamma^m$ such that $[v_1u]_m \not\subseteq [M_0]_m$, gives rise to the expression

$$v_0M_0(v_1u)\big(u^{-1}M_1\big)v_2M_2\dots v_nM_nv_{n+1}\,. \tag{4}$$

Recall that M_1 is recognized by an NFA with a single SCC. It follows immediately that $u^{-1}M_1$ is m-repeatable and $[u^{-1}M_1]_m = [M_1]_m$. Consequently,

$$\begin{aligned}\big[\mathsf{suf}_m(u)\, \big(u^{-1}M_1\big)\, \mathsf{pref}_m(v_2)\big]_m = \big[M_1\, \mathsf{pref}_m(v_2)\big]_m \subseteq{}& \\ \subseteq \big[M_1\big]_m = [u^{-1}M_1]_m&\end{aligned}$$

and the expression (4) is an m-frame. Since $[v_1u]_m \not\subseteq [M_0]_m$, we can conclude by the induction hypothesis applied to $(v_1u)\big(u^{-1}M_1\big)v_2M_2\dots v_nM_nv_{n+1}$.

The third kind of set results in the expression

$$v_0\tilde{M}_0u\big(u^{-1}L(\mathcal{A}_{\{q\}})\big)v_2M_2\dots v_nM_nv_{n+1} \tag{5}$$

where $\tilde{M}_0 = M_0v_1\Big\{w \in L(A^{\{q\}}) \mid \big[\mathsf{suf}_m(v_1)\, w\, \mathsf{pref}_m(u)\big]_m \subseteq [M_0]_m\Big\}$, $|u| = m+1$, $[u]_m \not\subseteq [M_0]_m$. Like in the first case we show that $\tilde{M}_0$ is m-repeatable, $[\tilde{M}_0]_m = [M_0]_m$, and $\big[\mathsf{suf}_m(v_0)\, \tilde{M}_0\, \mathsf{pref}_m(u)\big]_m = [\tilde{M}_0]_m$, and like in the second case $u^{-1}L(\mathcal{A}_{\{q\}})$ is m-repeatable and $\big[\mathsf{suf}_m(u)\, \big(u^{-1}L(\mathcal{A}_{\{q\}})\big)\, \mathsf{pref}_m(v_2)\big]_m = [u^{-1}L(\mathcal{A}_{\{q\}})]_m$. Since $[u]_m \not\subseteq [M_0]_m = [\tilde{M}_0]_m$, we can conclude by the induction hypothesis applied to $u\big(u^{-1}L(\mathcal{A}_{\{q\}})\big)v_2M_2\dots v_nM_nv_{n+1}$.

The last kind of set yields

$$v_0\tilde{M}_0(uv_2)M_2\dots v_nM_nv_{n+1} \tag{6}$$

where $\tilde{M}_0 = M_0v_1\big\{w \in M_1u^{-1} \mid \big[\mathsf{suf}_m(v_1)\, w\, \mathsf{pref}_m(u)\big]_m \subseteq [M_0]_m\big\}$, $|u| = m$, $[uv_2]_m \not\subseteq [M_0]_m$ and the reasoning is similar to the one for (5).

The bound on the number of explicit m-frames follows immediately from the inductive proof.

The bound on the size of the automata representing each explicit m-frame follows from the fact that for each $X \subseteq \Gamma^{\leq m}$, the language $\Gamma^*X\Gamma^*$ can be recognized with a deterministic automaton of $|\Gamma|^m$ states: the state space of the automaton is $\Gamma^{<m} \cup \{\top\}$, the automaton remembers last $m-1$ letters of the input word. □

LEMMA 19. *A regular language recognized by an NFA with k states can be written as a union of $(k+3)^{2k} \cdot |\Gamma|^{(3m+3)k}$ explicit m-frames of length at most k, each of which can be represented with NFAs of total size $k \cdot |\Gamma|^m$.*

Moreover, the explicit frames can be computed in time polynomial in $(k+3)^{2k} \cdot |\Gamma|^{(3m+3)k}$.

PROOF. Let $\mathcal{A}$ be an NFA with k states. $L(\mathcal{A})$ is a union of languages of the form

$$L(q_1)a_1L(q_2)a_2\dots a_{n-1}L(q_n)\,,$$

where q_1 is an initial state of $\mathcal{A}$, for $i < n$ the states q_i and q_{i+1} are in different SCCs of $\mathcal{A}$ and the language $L(q_i)$ consists of words admitting a run of $\mathcal{A}$ starting in q_i and finishing in some state p_i from the same SCC as q_i satisfying $(p_i, a_i, q_{i+1}) \in \delta^{\mathcal{A}}$. $L(q_n)$ is the set of words admitting a run starting in q_n and finishing in some final state in the same SCC of $\mathcal{A}$.

Note that if the SCC containing q_i is nontrivial (i.e., contains at least one transition), $L(q_i)$ is m-repeatable for any m. If the corresponding SCC is trivial, $L(q_i) = \{\varepsilon\}$. It follows that each such language is an m-frame for any m. Note also that $n \leq k$ and the number of such languages is at most $k^k|\Gamma|^k$.

To obtain the claim, in each expression above replace each $L(q_i) \neq \{\varepsilon\}$ with

$$(L(q_i) \cap \Gamma^{<2m}) \cup \bigcup_{x,y \in \Gamma^m} y\big(y^{-1}L(q_i)x^{-1}\big)x\,.$$

This turns each expression into a union of at most $\big(|\Gamma|^{2m+1}\big)^k$ m-frames, and altogether gives at most $|\Gamma|^{(2m+2)k} \cdot k^k$ m-frames. The claim follows by Lemma 18. □

We are now ready to prove Lemma 15. The proof follows the reasoning from [8] used to obtain Lemma 13, but replaces a simple pumping argument of [8] with Lemma 19.

PROOF OF LEMMA 15. With the DTD $\mathcal{D}_s = \langle \mathsf{r}, P_{\mathcal{D}_s}\rangle$ over the labelling alphabet Γ we associate a graph G over Γ in which there is an edge from σ to τ, if τ occurs in the production for σ (i.e., σ-labelled node can have a τ-labelled child).

We shall work with strongly connected components (SCCs) of G. An SCC X of G is called *branching* if for some label $\sigma \in X$ there exists a word in the language assigned to σ by the production of $\mathcal{D}_s$, such that at two different positions of this word there are letters from X. If a non-trivial SCC is not branching, it is called *non-branching*.

The general idea is that branching SCCs are replaced with tree ports, non-branching SCCs by context ports, and the SCCs of the NFAs representing the productions of $\mathcal{D}_s$ are replaced with forest ports. We will show how to obtain a kind $\mathcal{K}$ (satisfying the conditions given in the statement of the lemma) such that $T \in L(\mathcal{K})$. The bounds on the time of computing kinds covering the whole set $L(\mathcal{D}_s)$ will follow immediately from the described construction.

Let $T \in L(\mathcal{D}_s)$. Prune T introducing ports as stubs according to the following rules.

First look at sequences of children, processing the tree from the root towards the leafs. Let w be a sequence of all children of some node (that has not been removed so far). By Lemma 19 there exists an explicit p-frame $F = v_0M_0v_1M_1 \dots v_nM_nv_{n+1}$ such that $w \in F$. For each i replace

- the forest in T that corresponds to the (unique) subword of w that matches M_i

with

- a forest port u_i with L_{u_i} defined by the forest DTD $(M_i, P_{\mathcal{D}_s})$.

Next, deal with the SCCs of G, again moving top down. Each maximal path of the tree T that is labelled exclusively with labels from a branching SCC X is cut off at depth $p + 1$; under the freshly obtained leaf we put a tree port u with L_u given by $(\sigma, P_{\mathcal{D}_s})$, where σ is the label of the root of the removed subtree.

For non-branching X only consider paths of length at least $2p + 1$. Replace the subtree rooted at the $(p + 1)$st node of the path, say v, with a context port u and under it put the subtree originally rooted at the pth node of the path counting from the bottom, say w. The language L_u is defined by the context DTD $(\mathit{lab}(v), P)$ over the alphabet $\Gamma \cup \{\circ\}$ with P defined as follows. For $\sigma \notin X$, let $P(\sigma) = P_{\mathcal{D}_s}(\sigma)$. For $\sigma \in X$, let $P(\sigma) = \big(P_{\mathcal{D}_s}(\sigma) \cap \Gamma^*X\Gamma^*\big) \cup N_\sigma$, where N_σ is obtained form $P_{\mathcal{D}_s}(\sigma) \cap \Gamma^*X\Gamma^*$ by replacing in each word the only occurrence of $\mathit{lab}(w)$ with $\circ$. Note that $P(\sigma)$ can be recognized by an automaton twice the size of the automaton recognizing $P_{\mathcal{D}_s}(\sigma)$: it suffices to add transitions over $\circ$ between each two states for which there was a transition over $\mathit{lab}(w)$, and keep record of whether a label from X has been seen. Since X is a non-branching SCC,

> nodes labelled with symbols from X form a set of disjoint (simple) paths, such a node from one path is never a descendant of a node from another path. ($\star$)

It follows immediately that in each tree conforming to $(\mathit{lab}(v), P)$ above, there may be at most one node labelled with $\circ$.

Let $\mathcal{K}$ be the result of the procedure above. Note that if a path in T stays in some component of G for at most $2p$ steps (p for branching SCCs), then no port is introduced. In particular, for some small trees T, the resulting object may be T itself (no ports).

By construction, $\mathcal{K}$ is a p-kind and $T \in L(\mathcal{K})$. The bounds on the size and representation of $\mathcal{K}$ are straightforward to check. To see that $\mathcal{K}$ is an *explicit* p-kind note that condition (1) of Definition 4 is guaranteed by Lemma 19 and condition (2) follows from ($\star$). □

E. KIND THEOREM

The proof of the p-kind theorem is very similar to the proof of the original Theorem 5 from [8]. The only change involves the forest ports which now have a generalized form. We begin with two lemmas showing how to combine trees in the languages defined by p-kinds. Next, we show how to limit the set of data values in the target kinds. The proof of the p-kind theorem comes last.

LEMMA 20. *Let $\mathcal{K}(\bar{c})$ be a p-kind. Then, for all $T_1, T_2, \dots, T_n \in L(\mathcal{K}(\bar{c}))$ there exists $T \in L(\mathcal{K}(\bar{c}))$ such that for each $\pi(\bar{x})$ of size at most p, if $\pi(\bar{a})$ is matched neatly in some T_i, then $T \models \pi(\bar{a})$.*

PROOF. In a context or tree port u, use the corresponding condition in Definition 5 to provide a compatible forest/context T_u containing all the forests/contexts $(T_i)_u$. For forest ports, we cannot combine $(T_i)_u$'s directly, because the condition (1') of Definition 5 is to weak. Instead, we reuse the trick from Lemma 14. For each i, let $F^i_1, F^i_2, \dots, F^i_{k_i}$ be all the forests that can be obtained by taking p consecutive trees in the forest $(T_i)_u$, enumerated according to their original positions in $(T_i)_u$. Let $H_1, H_2, \dots, H_m$ be the sequence of forests obtained by concatenating the sequences $F^i_1, F^i_2, \dots, F^i_{k_i}$ for $i = 1, 2, \dots, n$. By (1') in Definition 5, we can easily construct a forest

$$T_u = G_0 + H_1 + G_2 + H_3 + G_4 + \cdots + G_m + H_m + G_{m+1} \in L_u \,. \tag{7}$$

Suppose that some pattern $\pi(\bar{a})$ of size at most p is matched neatly in T_i, and let us observe the image of $\pi(\bar{a})$ in $(T_i)_u$. If u is a tree or context port, we can move the image of $\pi(\bar{a})$ from $(T_i)_u$ to its copy in T_u without violating any edges in π. Suppose that u is a forest port with T_u defined by (7). By the pigeon-hole principle, between any $m + 1$ consecutive roots $(T_i)_u$ we can find a root to which no vertex of $\pi(\bar{a})$ is matched. It follows, that we can decompose $(T_i)_u$ into nonempty forests $F_1 + F_2 + \cdots + F_k$, each consisting of at most m roots, such that for all vertices x, y of π, if x is matched in F_j and y is matched in $F_{j'}$ for $j < j'$, then the only edge between x and y in π can be $E_f(x, y)$. (If such an edge exists, x and y must be mapped to some roots of F_j and $F_{j'}$, respectively.) By the definition of $H_1, H_2, \dots, H_m$, there exist $\ell_1 < \ell_2 < \cdots < \ell_k$ such that $H_{\ell_j} = F'_j + F_j + F''_j$ for all j. Hence, by moving the image of $\pi(\bar{a})$ from F_j to the copy of F_j contained in H_{ℓ_j}, we obtain a partial matching. Since the original matching of $\pi(\bar{a})$ in T_i was neat, all the partial matchings are compatible and give a matching of $\pi(\bar{a})$ in T. □

LEMMA 21. *Let $\mathcal{K}(\bar{c})$ be a source p-kind for $\mathcal{M}$, where p is the maximal size of source-side patterns in $\mathcal{M}$. For all $T_1, T_2, \dots, T_n \in L(\mathcal{K}(\bar{c}))$ there exists $T \in L(\mathcal{K}(\bar{c}))$ such that for each source-side pattern π, if $T_i \models \pi(\bar{a})$ for some i, then $T \models \pi(\bar{a})$.*

PROOF. Since $\mathcal{K}(\bar{c})$ is a source kind, for each source-side pattern $\pi(\bar{x})$ such that $T_i \models \pi(\bar{a})$, there exists $T_{\pi(\bar{a})} \in L(\mathcal{K}(\bar{c}))$ such that $T_{\pi(\bar{a})} \models \pi(\bar{a})$. It follows immediately that the tree T guaranteed by Lemma 20 for the set of trees

$$\big\{T_{\pi(\bar{a})} \;\big|\; \pi \text{ is a source-side pattern of } \mathcal{M} \text{ and } T_i \models \pi(\bar{a}) \text{ for some } i\big\}$$

satisfies the conditions of the lemma. □

LEMMA 22. *Let $\mathcal{M} = (\mathcal{D}_s, \mathcal{D}_t, \Sigma)$ be an absolutely consistent mapping and let p be the maximal size of source-side patterns in Σ. Let $\mathcal{K}^s(\bar{c})$ be a source p-kind, and let $\mathcal{K}^t_1, \mathcal{K}^t_2, \dots, \mathcal{K}^t_k$ be target kinds covering $L(\mathcal{D}_t)$. Then, each tree from $L(\mathcal{K}^s(\bar{c}))$ has a solution in $L(\mathcal{K}^t_i(\bar{d}))$ for some i and some tuple $\bar{d}$ with entries taken from $\bar{c}$ or from a fixed set of nulls of size $K = \max_i |\mathcal{K}^t_i|$.*

PROOF. Let us take a tree $T \in L(\mathcal{K}^s(\bar{c}))$. Consider copies $T_0, T_1, \dots, T_K$ of T obtained by renaming the data values not occurring in $\bar{c}$ (i.e., used only outside of $\mathcal{K}^s$); in T_i the new names are chosen from set A_i such that $\bar{c}$, $A_0, A_1, \dots A_K$ are pairwise disjoint. By Lemma 21 we obtain a tree T' which yields all the valuations of source side patterns yielded by any of the T_i's. In particular, whenever $T \models \pi(\bar{a})$, for each i we have a copy $\pi(\bar{a}_i)$ of $\pi(\bar{a})$ satisfied in T', whose data values come from $A_i \cup \bar{c}$. As $\mathcal{M}$ is absolutely consistent, the combined tree has a solution, say S' in some $L(\mathcal{K}^t_j)$. It follows that for some i_0 the corresponding copy of the target pattern, $\pi'(\bar{a}_{i_0}, \bar{b}_{i_0})$, is matched in such a way that none of the nodes carrying a value from A_i is mapped to the copy of $\mathcal{K}^t_j$ in S'. Indeed, should each $\pi'(\bar{a}_i, \bar{b}_i)$ contain a node carrying a value from A_i, that is mapped to the copy of $\mathcal{K}^t_j$, for each $i = 0, 1, \dots, K$ there would be a node in $\mathcal{K}^t_j$ carrying a value from A_i; this contradicts the fact that there are at most K nodes in $\mathcal{K}^t_j$. Consider a copy T'' of the tree T' that is obtained as follows.

1. Rename all values in $\mathcal{K}^t_j$ that are not from $\bar{c}$ as nulls: a value a becomes a null $\bot_a$. This gives a valuation $\bar{d}$ of data values in $\mathcal{K}^t_j$. Note that $\bar{d}$ does not depend on π', nor $\bar{a}$, nor i_0.
2. Look at the data values that were renamed into nulls, and were touched by the matching of $\pi'(\bar{a}_{i_0}, \bar{b}_{i_0})$ in $\mathcal{K}^t_j$. From the way i_0 was chosen it follows that these values were originally not from A_{i_0}. Thus, if we replace each occurrence of a with $\bot_a$ also outside $\mathcal{K}^t_j$, $\pi'(\bar{a}_{i_0}, \bar{b}'_{i_0})$ is satisfied for tuple b'_{i_0} obtained from $\bar{b}_{i_0}$ by the same renaming.
3. Finally, we change the data values from A_{i_0} back to the original data values used in T.

The obtained tree T'' satisfies $\pi'(\bar{a}, \bar{b})$ for tuple $\bar{b}$ obtained from b'_{i_0} by the same renaming. Also, T'' is a tree of kind $\mathcal{K}^t_j(\bar{d})$ such that $\bar{d}$ only uses data values from $\bar{c}$ and elements of a fixed set of nulls of size K. Hence, for each source side pattern $\pi(\bar{a})$ satisfied in T we have a tree in $\mathcal{K}^t_j(\bar{d})$ that satisfies $\pi'(\bar{a}, \bar{b})$. Combining the trees as usual we obtain a solution for T in $L(\mathcal{K}^t_j(\bar{d}))$. □

The proof of the p-kind theorem is an easy consequence of the last two lemmas. Suppose that for each $\mathcal{K}^t_i(\bar{d})$ with the entries of $\bar{d}$ taken from among the entries of $\bar{c}$ or a fixed set of nulls of size $|\mathcal{K}^t_i|$, there is a tree $T_{i,\bar{d}}$ in $L(\mathcal{K}^s(\bar{c}))$ that does not admit a solution in $\mathcal{K}^t_i(\bar{d})$. By Lemma 21, we combine all these trees to produce a tree T that satisfies $\pi(\bar{a})$ whenever some $T_{i,\bar{d}}$ satisfies $\pi(\bar{a})$ for all source-side patterns of $\mathcal{M}$. By Lemma 22, T has a solution T' in some $\mathcal{K}^t_{i_0}(\bar{d})$. By the definition of T, T' is also a solution for $T_{i_0,\bar{d}}$, which is a contradiction. This shows that there exist i and $\bar{d}$ with entries taken from $\bar{c}$ or a fixed set of nulls of size $|\mathcal{K}^t_i|$, such that each tree from $L(\mathcal{K}^s(\bar{c}))$ has a solution in $L(\mathcal{K}^t_i(\bar{d}))$.

For a given i, we can compute appropriate $\bar{d}$ by testing all possible $\big(|\bar{c}| + |\mathcal{K}^t_i|\big)^{|\mathcal{K}^t_i|} \le (2K)^K$ candidate tuples, where $K = \max(|\mathcal{K}^s|, |\mathcal{K}^t_i|)$. Arguing like in the previous paragraph, we show that all trees in $L(\mathcal{K}^s(\bar{c}))$ have solutions in $L(\mathcal{K}^t_i(\bar{d}))$ iff for each dependency $\pi(\bar{x}) \to \pi'(\bar{x}, \bar{y})$ in $\mathcal{M}$ and each tuple $\bar{a}$ with entries taken from $\bar{c}$ and a set of nulls of size $|\bar{x}|$, if $\pi(\bar{a})$ is satisfiable with respect to $L(\mathcal{K}^s(\bar{c}))$, then $\pi'(\bar{a}, \bar{b})$ is satisfiable with respect to $L(\mathcal{K}^t_i(\bar{d}))$ for some tuple $\bar{b}$ with entries taken from $\bar{c}$, $\bar{d}$, or a set of nulls of size $\bar{y}$. Since $\mathcal{K}^s$ is a source kind, and $\mathcal{K}^t_i$ is a target kind, it is enough to consider neat matchings of these patterns. Consequently, by Lemma 11, the latter test can be performed in time

$$\begin{aligned} n \cdot (|\bar{c}| + r)^r \cdot \Big(\|\mathcal{K}^s\|^{poly(\|\mathcal{M}\|)} \cdot |\mathcal{K}^s|^{p+1} + (|\bar{c}| + |\bar{d}| + r)^r \cdot \|\mathcal{K}^t_i\|^{poly(\|\mathcal{M}\|)} \cdot |\mathcal{K}^t_i|^{p+1} \Big) \le \\ \le K^{poly(\|\mathcal{M}\|)} \cdot N^{poly(\|\mathcal{M}\|)} \end{aligned}$$

where n is the number of dependencies, p and r are the maximal size and arity of patterns, and $N = \max(\|\mathcal{K}^s\|, \|\mathcal{K}^t_i\|)$. Altogehter, the complexity is bounded by $K^{K \cdot poly(\|\mathcal{M}\|)} \cdot N^{poly(\|\mathcal{M}\|)}$. Assuming that $N = 2^{poly(\|\mathcal{M}\|)}$ (and $K > 2$), we obtain complexity $K^{K \cdot poly(\|\mathcal{M}\|)}$.

F. KERNELIZATION WITH INEQUALITIES

We prove that a kernel can be found by taking union of at most r^r kernels given by applications of Theorem 2 to carefully prepared sets of tuples and potential expressions.

Let $\equiv$ be an equivalence relation on $\bar{x}$. By $D_\equiv$ we denote the set of tuples of D that have the following property: two entries in a tuple are equal if and only if the corresponding variables in $\bar{x}$ are $\equiv$-equivalent; note that sets $D_\equiv$ can be constructed using single queries. Moreover, let $\alpha_\equiv$ be a potential expression created from α as follows: we delete all the clauses that contain a constraint $x_i \neq x_j$ for $x_i \equiv x_j$, while from all the other clauses we delete all the remaining inequality constraints. Note that $\alpha_\equiv$ has at most k clauses and uses no inequality constraints.

Using Theorem 2, for every equivalence relation $\equiv$ we compute a kernel $D'_\equiv$ for $D_\equiv$ with respect to $\alpha_\equiv$. Note there are at most r^r equivalence relations over r variables, so we obtain at most r^r kernels of size at most $2 \cdot (2k)^r$ each; the bound on the total number of queries follows in the same manner. Let $D' = \bigcup_\equiv D'_\equiv$. We have that $|D'| \leq 2 \cdot (2kr)^r$, so it remains to show that D' is a kernel for D with respect to α.

Let us fix $\bar{c}$ such that $\alpha(\bar{a}, \bar{c})$ for all $\bar{a} \in D'$. Take a tuple $\bar{a} \in D \setminus D'$. Let $\equiv$ be an equivalence relation over $\bar{x}$ such that $x_i \equiv x_j$ if and only if $a_i = a_j$. As $\bar{a} \notin D'$, it follows that $a \in D_\equiv \setminus D'_\equiv$. We know that every tuple of $D'_\equiv \subseteq D'$ satisfies $\alpha(\bar{x}, \bar{c})$. We claim that the same holds for $\alpha_\equiv(\bar{x}, \bar{c})$. Indeed, no clause deleted while constructing $\alpha_\equiv$ could be satisfied by a tuple from $D'_\equiv$ as the inequality constraints that triggered deletion are automatically not satisfied, while all the deleted inequality constraints in other clauses are automatically satisfied. As $D'_\equiv$ is a kernel for $D_\equiv$, we infer that $\bar{a}$ satisfies $\alpha_\equiv$. We claim that $\bar{a}$ also satisfies α. Indeed, $\bar{a}$ must satisfy some clause of $\alpha_\equiv$, and all the inequality constraints that were removed from its original in α are satisfied automatically. This concludes the proof that D' is a kernel for D with respect to α.

We remark that the bound on the kernel size in Theorem 3 is far from being tight, when the constant depending on r is taken into consideration. To see this, note that the sets $D_\equiv$ for $\equiv$ not consisting of singletons only are contained in subspaces of dimension smaller than r, and hence usage of Theorem 2 can give a kernel with exponent smaller than r. As we treat r as a constant, we omit such a sharper analysis for the sake of simplicity.

G. BRANCHING ALGORITHM IMPLEMENTATION

In this part we explain how to construct a query $Const\mathcal{B}_\mathcal{K}$ which solves tuple covering problem using the branching algorithm described in Section 5 Lemma 6. This query can be used to compute possible valuations of the constants of a kind.

As explained in the proof of Lemma 6, the branching algorithm refines iteratively a partial valuation $\bar{c}$ of the variables $\bar{z}$ together with a set of (in)equality $\mathcal{E}$ so that all tuples from the sets D_i gets covered at the end. At each iteration, it chooses an uncovered tuple $\bar{a}$ from a set D_i (i.e. witnessing the pattern π_i in the source) and tries to fix some clause P^i_j from α_i at $\bar{a}$ by extending the partial valuation $\bar{c}$ and the set $\mathcal{E}$ so that $\bar{a}$ is now covered by P^i_j under $(\bar{c}, \mathcal{E})$.

Also recall that the recursion depth of the algorithm is bounded. A variable z_i from the tuple $\bar{z}$ can be valuated only once and each equality or inequality can be added to $\mathcal{E}$ only once. At each iteration we valuate at least one element of $\bar{z}$ or we add at least one equation to $\mathcal{E}$. Altogether the algorithm refines the partial valuation at most $W = 2|\bar{z}| + \binom{|\bar{z}|}{2}$ times.

In the query, for $\ell \in \{1, 2, \ldots, W\}$ the data structure C_ℓ stores the partial valuation of $\bar{z}$ at the ℓ^{th} refining step. The current chosen tuple $\bar{a} \in D_i$ will be stored in the variables $(i, \bar{x})$ and the chosen clause to cover a is stored in h_ℓ. The list of clauses is encoded in the subquery $Clause$. Notice that we do not store $\mathcal{E}$ explicitly but we can easily reconstruct it from the variables h_i for $i < \ell$. For technical reason C_ℓ is a list of variables which are valuated to elements of $\mathbb{D}$, in particular variables that are not in the list C_ℓ have not been valuated yet (i.e., they are assigned value $\bot$).

The query $Const\mathcal{B}_\mathcal{K}$ can be defined as

$$\text{let } C_0 := \mathit{empty}_{val} \text{ return } Refine_0$$

For each $\ell \in \{1, 2, \ldots, W\}$, the subqueries $Refine_\ell$ intuitively corresponds to the ℓ^{th} refining step. It is formally defined inductively as defined below for $\ell \leq W$ and we define $Refine_{W+1}$ as the empty query.

```
$Refine_\ell =$
     if empty($Find_\ell$) then $Next_\ell$
       else let $(i, \bar{x})$ := first($Find_\ell$) return (
         for $h_\ell$ in $Clause$ where $\alpha_{h_\ell}(h_\ell, i, \bar{x})$ return (
             let $C_{\ell+1} := UpdateC_\ell(h_\ell, \bar{x})$ return $Refine_{\ell+1}$
         ))
```

The subqueries $Find_\ell$.

For any ℓ, the corresponding subquery selects tuples of pairs $(i, \bar{x})$ where $\bar{x} \in D_i$ (that is, $\bar{x}$ satisfies π_i) such that the $\bar{x}$ cannot be covered by any clause P^i_j under the partial valuation C_ℓ. It can be easily construct using q_{π_i} queries from Lemma 1 together with some (in)equality tests on the structure C_ℓ. Note that these tests may depend also on variables h_i for $i \leq \ell$.

The subqueries $Next_\ell$.

The $Next_\ell$ in line 1 intuitively corresponds to the case when every tuple from every D_i is covered by one of clauses under the current partial valuation of $\bar{z}$.

$$Next_\ell = \ \mathsf{let}\ C := C_\ell\ \mathsf{return}\ Clean$$

The subquery $Clean$ outputs a full valuation of the variables $\bar{z}$ according to the current partial valuation stored in C and the (in)equality constraints (corresponding to $\mathcal{E}$) stored in the variables h_i (for $0 \leq i < W$). The values assigned to variables of $\bar{z}$ that were not assigned values in C are obtained using query $\mathsf{freshnull}()$, where variables that are constrained to be equal in $\mathcal{E}$ are assigned the same fresh null.

The subqueries $UpdateC_\ell(h_\ell, \bar{x})$.

For any ℓ, the subquery $UpdateC_\ell(h_\ell, \bar{x})$ outputs the refined partial valuation obtained from C_ℓ and $\bar{x}$ knowing that the clause P^i_j (encoded by the constant h_ℓ) has been chosen to cover $\bar{x}$.

The subexpression $\alpha_{h_\ell}(h_\ell, i, \bar{x})$.

For each ℓ this subexpression tests whether $\bar{x}$ is covered by the clause P^i_j encoded by h_ℓ under the partial valuation C_ℓ and the constraints $\mathcal{E}$ encoded by the variables h_i for $0 \leq i < \ell$.

Theorem 1 now follows easily from the construction of the query $Const\mathcal{B}_{\mathcal{K}}$.

H. KERNELIZATION ALGORITHM IMPLEMENTATION SKETCH

In the query we are constructing, we use recursion and build-in arithmetics. In particular, the construct $\mathsf{count}(q)$ outputs the length of the sequence returned by the query q. The construct $\mathsf{firstHalf}(q)$ returns first $\lfloor \frac{\mathsf{count}(q)}{2} \rfloor$ elements of the sequence returned by q and can be implemented using the for construction and the built-in arithmetics.

As explained in Section 6, the idea of kernelization is to recursively shrink the set of tuples called kernel until we obtain a kernel of small size. In our case, we start with the kernel being the set D of tuples which satisfy some π in the source instance (this set can be computed using the query q_π from Lemma 1). In each recursion call we look for a subset Y of D that can be removed safely until we get small kernel. As the removed sets are of the size proportional to D the recursion depth of the algorithm depends on the size of D (and as a consequence on the size of the input tree). Finding the subset Y can be done inductively using clauses of a potential expression.

Fix a dependency $\pi \to \pi'$. Let r be the arity of the pattern π and $\alpha = \bigvee_{i=1}^{k} P_i$ the corresponding potential expression from Lemma 4. We explain how to construct a query $Kernelz$ which computes from a set D_{ker} a kernel of size smaller than $2 \cdot (2k)^r$.

The query $Kernelz$.

In the query, the structure F_α represents an encoding of the clauses of the potential expression α and is fixed. Constants k and r are fixed as well. The structure D_{ker} refers to the kernel at the current level of recursion. The query $Kernelz$ is defined recursively as

```
$Kernelz =$
1     if count$(D_{ker}) < 2(2k)^r$ return $D_{ker}$
2       else  let $X_0 := D_{ker}$ return
3               let $G^0_\alpha := F_\alpha$ return
4                 let $Y := FindSet_0$ return
5                   let $D_{ker} := D_{ker} \setminus Y$ return $Kernelz$
```

The set difference expressed in line 3 can be easily encoded with a simple query using D_{ker} and Y.
Notice that line 5 is the only place where we use recursion in the whole construction. This is unavoidable as the recursion depth of the kernelization algorithm depends on the size of the input tree.

The subquery $FindSet_0$.

At each level of the kernelization, we compute a set of tuples that can be safely removed from the current kernel to obtain a smaller one. This is done using the subquery $FindSet_0$. The idea is to refine iteratively a set of tuples X_0 using some clauses from the potential expression α. The number of iterations is bounded by r, the dimension of space that contains X_0. The query $FindSet_\ell$ is defined inductively below for $\ell < r$ and we define $FindSet_r$ as $\emptyset$.

```
$FindSet_\ell =$
    if empty($FindBigSet_\ell$) then firstHalf($X_\ell$)
        else let $(\bar{a}_\ell, i_\ell) :=$ first($FindBigSet_\ell$) return
            let $X_{\ell+1} := EvalP^\ell_{\bar{a}_\ell, i_\ell}$ return
              let $G^{\ell+1}_\alpha := Update(G^\ell_\alpha, i_\ell)$ return $FindSet_{\ell+1}$
```

The variables G^ℓ_α refers to the sequences of clauses from the potential expression α which haven't been used yet to refine the set X_ℓ. The variables X_ℓ refers to the current set of tuples we want to refine.

The subqueries $EvalP^\ell_{\bar{y},j}$ and $FindBigSet_\ell$.

For each ℓ, the corresponding subquery $FindBigSet_\ell$ outputs a sequence of pairs $(\bar{y}, j)$ where $\bar{y}$ is an r-tuple from D_{ker} and j encodes a clause from G^ℓ_α witnessing the fact that (2) does not hold, that is:

$$X_\ell \not\subseteq \hat{P}^{\bar{y}}_j \text{ and } |X_\ell \cap \hat{P}^{\bar{y}}_j| \geq \frac{|X_\ell|}{2k}.$$

Recall that $\hat{P}^{\bar{y}}_j$ can be defined using a conjunction $C^{\bar{y}}_j$. Using this conjunction, it is easy to design a series of subqueries $EvalP^\ell_{\bar{y},j}$ which output the tuples from $X_\ell \cap \hat{P}^{\bar{y}}_j$, where $\bar{y}$ can be instantiated to various variables used in the implementation.

The subquery $FindBigSet_\ell$ can now be defined as

```
$FindBigSet_\ell =$
    1 for $j \in G^\ell_\alpha$ return
    2     for $\bar{x} \in X_\ell$
    3         where count$(X_\ell) >$ count$(EvalP^\ell_{\bar{x},j}) \wedge$ count$(EvalP^\ell_{\bar{x},j}) > \frac{\text{count}(D_{ker})}{(2k)^{r-\text{count}(G^\ell_\alpha)}}$ return $(\bar{x}, j)$
```

Notice that in line 2 we consider $\bar{x} \in X_\ell$ instead of $\bar{x} \in D_{ker}$ because we are interested in clauses P_j such that $EvalP^\ell_{\bar{x},j} > 0$. Also notice that in line 3, the test count$(X_\ell) >$ count$(EvalP^\ell_{\bar{x},j})$ is equivalent to $X_\ell \not\subseteq \hat{P}^{\bar{x}}_j$.

I. TRACTABLE CASE

Without loss of generality, we can assume that for each $\sigma \in \Gamma$ there is a finite tree conforming to the DTD $\langle \sigma, P_t \rangle$. This can be guaranteed by a simple polynomial preprocessing, which first computes the set of labels Γ_0 that have this property, and then restricts $\mathcal{D}_t$ to Γ_0 by eliminating all remaining labels from the productions. (If Γ_0 does not contain the root symbol of $\mathcal{D}_t$, the implementing query $q_\mathcal{M}$ is trivial.)

The second step is to adjust each target side patterns to the target DTD by merging vertices that will have to be mapped to the same nodes in any tree conforming to $\mathcal{D}_t$. Recall that target side patterns are fully specified and tree-shaped. If the root node of a target-side pattern π' is not labelled with r, then π' is not satisfiable and we can replace it with a single inequality $y \neq y$. Otherwise, we process the nodes of π' top down, i.e., starting from the root. Pick the next unprocessed node v. Let σ be the label of v and let $\sigma \to \hat{\tau}_1 \hat{\tau}_2 \dots \hat{\tau}_k$ in $\mathcal{D}_t$. If among the children of v there is a node with a label not allowed by the production for σ, then π' is not satisfiable; we abort the procedure and replace π' with a single inequality $y \neq y$. Otherwise, for each j such that $\hat{\tau}_j$ is τ_j or $\tau_j?$, merge all the children of v labelled by τ_j into one τ_j-child and add the induced equalities between variables stored in these children. When this procedure terminates, the resulting pattern is consistent with the target DTD, except that some nodes are missing. They will be added later, when the final query is constructed.

We can now move on to the main argument. Since the target-side patterns are fully specified, they can only access nodes of the target tree up to depth p. Moreover, for each vertex of a target pattern there is a fixed sequence of labels from root to the accessed node. Since $\mathcal{D}_t$ is a simple threshold DTD, this means that the accessed node is either unique in each tree conforming to T or for some two consecutive labels σ, τ in this sequence, τ occurs as τ^* or τ^+ in the production for σ in $\mathcal{D}_t$.

It follows that there exists a single target kind $\mathcal{K}$ such that each source tree that has a solution, has also a solution in $L(\mathcal{K})$. The kind is obtained by unravelling $\mathcal{D}_t$. Begin with a single node labelled with r and then for each ordinary node v (not a port) labelled with σ, where $\sigma \to \hat{\tau}_1 \hat{\tau}_2 \dots \hat{\tau}_k$ in $\mathcal{D}_t$ add children according to the following rules. For each i, if some target pattern can access a τ_i-labelled child of node v in the constructed tree, then

- if $\hat{\tau}_i$ is τ_i or $\tau_i?$, add a τ_i-labelled node;
- if $\hat{\tau}_i$ is τ_i^+ or τ_i^*, add a forest port with the corresponding forest DTD $\langle \hat{\tau}_i, P_t \rangle$.

otherwise,

- if $\hat{\tau}_i$ is τ_i or τ_i^+, add a τ_i-labelled node;
- if $\hat{\tau}_i$ is τ^+ or τ^*, add nothing.

By the initial assumption, this process will terminate at depth at most $p + h$, where p is the maximal size of target-side pattern and h is the height of $\mathcal{D}_t$. The size of the resulting kind $\mathcal{K}$ can be bounded by $M \cdot N$, where $M \leq \|\Sigma\|$ is the total size of target-side patterns, and $N = \max_{\sigma\in\Gamma} \min_{S\in L(\langle\sigma,P_t\rangle)} |S|$ is the bound on the size of minimal subtrees consistent with the target DTD, as defined in the theorem statement. The factor N comes from the fact that we also add nodes inaccessible by target patterns, but enforced by the target DTD. The ordinary nodes of $\mathcal{K}$ can be split into two categories: those accessible by target-side patterns, and those not accessible. The accessible nodes form a strict subtree (an ancestor closed subset) of $\mathcal{K}$, and their number is bounded by M. The data values in the non-accessible nodes are irrelevant and can be set to nulls. The data values in the accessible nodes have to be computed based on the source tree.

Let $v_1, v_2, \ldots, v_m$ be the accessible nodes in $\mathcal{K}$ and let $\bar{z} = z_1, z_2, \ldots, z_m$. We preprocess the mapping $\mathcal{M}$ so that its dependencies are of the form $\pi_i(\bar{x}_i) \longrightarrow \pi'_i(\bar{x}_i, \bar{y}_i, \bar{z})$, $i = 1, 2, \ldots, n$, where in π'_i the vertices that access node v_j carry variable z_j (vertices that access nodes outside of $\mathcal{K}$ carry arbitrary variables) and no equalities involve variables from $\bar{y}_i$. First, we replace the variables in vertices accessing node v_j with z_j and add an appropriate equality. Next, we eliminate equalitites between variables from $\bar{z}$ and $\bar{y}_i$: for each equality $z_j = y$ where y is a variable in $\bar{y}_i$, we replace each occurrence of y with z_j. Similarly, we remove equalities between $\bar{x}_i$ and $\bar{y}_i$, and equalities over $\bar{y}$.

The way we want to think about it is variables $\bar{z}$ are dedicated to constants of $\mathcal{K}$, $\bar{x}_i$ bring a tuple from the source side, and $\bar{y}_i$ are fresh nulls, implicitly assumed to be pairwise different and also different from any data value used on the source side, as well as values stored in the constants of $\mathcal{K}$. Since the target side patterns are merged, we can assume that they are always matched injectively, i.e., no two vertices of a target pattern need to be mapped to the same node of the tree. Thus, by substituting each variable in $\bar{y}_i$ with a fresh null, we satisfy all inequalities involving $\bar{y}_i$.

For each accessible node v_i, if $\pi_j(\bar{a})$ is matched in the source tree T, and $\pi'_j(\bar{x}_j, \bar{y}_j, \bar{z})$ contains equality $z_i = x_\ell$, then v_i must store a_ℓ. In particular, if

$$A_i = \bigcup_{j=1}^{n} \left\{ a_\ell \;\middle|\; T \models \pi_j(\bar{a}) \text{ and } \pi'_j(\bar{x}_j, \bar{y}_j, \bar{z}) \text{ contains equality } z_i = x_\ell \right\}$$

has more then one element, there is no solution at all. The value stored in v_i can be also enforced by equalities over $\bar{z}$ contained in target side patterns, which is reflected in the query constructed below.

The candidates for the constants of $\mathcal{K}$ are computed by the query $\mathit{const}_\mathcal{K}$, shown on the left, based on subquery $\mathit{equalities}_i(\bar{z})$, shown on the right:

```
let $\bar{t}_0 := (A_1, A_2, \ldots, A_m)$ return
  let $\bar{t}_1 := \mathit{equalities}_1(\bar{t}_0)$ return
  let $\bar{t}_2 := \mathit{equalities}_2(\bar{t}_1)$ return
          ⋮
  let $\bar{t}_n := \mathit{equalities}_n(\bar{t}_{n-1})$ return
    let $(z_1, \ldots, z_m) := \bar{t}_n$ return
      $(\mathit{firstOrNull}(z_1), \ldots, \mathit{firstOrNull}(z_m))$
```

```
if empty$(q_{\pi_i})$ then $\bar{z}$ else
  let $x_1 := \bigcup_{\pi'_i \models z_1 = z_j} z_j$ return
  let $x_2 := \bigcup_{\pi'_i \models z_2 = z_j} z_j$ return
      ⋮
  let $x_m := \bigcup_{\pi'_i \models z_m = z_j} z_j$ return
      $(x_1, x_2, \ldots, x_m)$
```

The expression $\mathit{firstOrNull}(x)$ is defined as if empty(x) then freshnull() else first(x). The union symbols in $\mathit{equalities}_i$ are used in lieu of concatenation. The condition $\pi_i \models z_1 = z_j$ means that z_j ranges over all variables such that equality $z_1 = z_j$ is entailed by the equalities over $\bar{z}$ contained in π'_i. Note that this always includes $j = 1$. The sets $A_1, \ldots, A_m$ can be easily computed with a polynomial size query.

If the tree T has a solution at all, then the constants returned by $\mathit{const}_\mathcal{K}$ are correct. But it is also possible that there is no solution, because the equality and inequality constraints imposed by target-side patterns are not satisfiable. This is checked by the additional condition in the final implementing query $q_\mathcal{M}$:

```
let $\bar{z} := \mathit{const}_\mathcal{K}$ return
    if $\bigwedge_{i=1}^{n}$ empty( let $\bar{y}_i :=$ (freshnull(), …, freshnull()) return
                    for $\bar{x}_i$ in $q_{\pi_i}$ where $\neg\eta'_i(\bar{x}_i, \bar{y}_i, \bar{z})$ return $\bar{x}_i$ ) then $\mathit{sol}_\mathcal{K}(\bar{z})$
```

In the query above, $\eta'_i(\bar{x}_i, \bar{y}_i, \bar{z})$ is the conjunction of equalities and inequalities contained in pattern π'_i.

The query $\mathit{sol}_\mathcal{K}(\bar{z})$ is defined like in the general construction in the proof of Lemma 3, but some additional effort is needed to make the construction polynomial. The kind $\mathcal{K}$ and the trees that need to be substituted at its ports can be exponential in the size of $\mathcal{D}_t$. Instead of building them into query $\mathit{sol}_\mathcal{K}(\bar{z})$ explicitly, we construct large parts of them with special queries q_σ, that return the smallest tree conforming to $\langle\sigma, P_t\rangle$. These trees may be exponential, but the query is polynomial: if the production in $\mathcal{D}_t$ is $\sigma \to \hat{\tau}_1\hat{\tau}_2 \ldots \hat{\tau}_k$, the query q_σ is

$$\text{let } y := \text{freshnull() return } \sigma(y)[q_{\tau_{i_1}}, q_{\tau_{i_2}}, \ldots, q_{\tau_{i_\ell}}]\,,$$

where $i_1 < i_2 < \cdots < i_\ell$ are all the indices i such that $\hat{\tau}_i = \tau_i$ or $\hat{\tau}_i = \tau_i^+$.

For each pattern π_i' and each port u in $\mathcal{K}$, the query $subst_{i,u}(\bar{x}_i, \bar{y}_i, \bar{z})$ outputs a forest to be substituted at u in order to obtain a tree in $L(\mathcal{K}(\bar{z}))$ satisfying $\pi_i'(\bar{x}_i, \bar{y}_i, \bar{z})$. Let $(\pi_i')_u$ be the sequence of subpatterns of π_i' rooted at vertices of π_i' that must be matched to the roots of the forest substituted at u (they are determined by the path leading to port u in $\mathcal{K}$). Query $subst_{i,u}(\bar{x}_i, \bar{y}_i, \bar{z})$ is obtained from $(\pi_i')_u$ by adding the nodes that are missing with respect to $\mathcal{D}_t$: for each node v with label σ, where $\sigma \to \hat{\tau}_1\hat{\tau}_2 \dots \hat{\tau}_k$ is the production in $\mathcal{D}_t$, include among children of v subqueries q_{τ_i} for each i such that $\hat{\tau}_i = \tau_i$ or $\hat{\tau}_i = \tau_i^+$ and no child of v is labelled with τ_i (make sure that the ordering required by the production $\sigma \to \hat{\tau}_1\hat{\tau}_2 \dots \hat{\tau}_k$ in $\mathcal{D}_t$ is respected). Additionally, if u is a forest port with root expression of the form τ^+, include q_τ in $subst_{i,u}(\bar{x}_i, \bar{y}_i, \bar{z})$ to make sure that the returned forest is not empty. By the second preprocessing, the forests returned by the obtained query are compatible with the ports.

The query $sol_{\mathcal{K}}(\bar{z})$ is essentially obtained by plugging in at each port u the query $subst_u(\bar{y}, \bar{z})$, where $\bar{y} = \bar{y}_1, \bar{y}_2, \dots, \bar{y}_k$, obtained as the concatenation of queries

$$\text{for } \bar{x}_i \text{ in } q_{\pi_i} \text{ return } subst_{i,u}(\bar{x}_i, \bar{y}_i, \bar{z})$$

for $i = 1, 2, \dots, k$. We would like to define $sol_{\mathcal{K}}(\bar{z})$ as

$$\begin{aligned} &\text{let } \bar{y} := (\text{freshnull}(), \dots, \text{freshnull}()) \text{ return} \\ &\qquad \mathcal{K}(subst_{u_1}(\bar{y}, \bar{z}), subst_{u_2}(\bar{y}, \bar{z}), \dots, subst_{u_m}(\bar{y}, \bar{z})) \end{aligned}$$

where $u_1, u_2, \dots, u_{m'}$ are all ports of $\mathcal{K}$. By the definition of $\mathcal{K}$, all ports are accessible, so there is at most M of them, which is fine. The problem is that $\mathcal{K}$ may be exponential, so again we need to use queries q_σ: when $\mathcal{K}$ is built, whenever an inaccessible τ-node is reached, we immediately substitute query q_τ. Note that this does not interfere with the previous steps of the construction, as we always work only with the accessible nodes of $\mathcal{K}$.