Synthesizing transformations from XML schema mappings

Claire David, Piotr Hofman, Filip Murlak, Michal Pilipczuk

To cite this version:

Claire David, Piotr Hofman, Filip Murlak, Michal Pilipczuk. Synthesizing transformations from XML schema mappings. 17th International Conference on Database Theory (ICDT 2014), Mar 2014, Athènes, Greece. pp.61-71, 10.5441/002/icdt.2014.10 . hal-01854501

HAL Id: hal-01854501
https://hal.science/hal-01854501

Submitted on 6 Aug 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - NoDerivatives 4.0 International License

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

(c) 2014, Copyright is with the authors. Published in Proc. 17th International Conference on Database Theory (ICDT), March 24-28, 2014, Athens, Greece: ISBN 978-3-89318066-1, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

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.

10.5441/002/icdt.2014.10

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). Missing arguments can be found in the full version of this article.

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.

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 $T_1 T_2 \dots T_p$ 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 $\operatorname{Rg} \xi_\pi$, is the set of variables used by π; the *arity* of π is $|\operatorname{Rg} \xi_\pi|$; and $\|\pi\|$ is the size of the underlying DAG. For a set Δ of patterns, $\|\Delta\|$ is the sum of $\|\pi\|$ for $\pi \in \Delta$.

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

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

We write $\pi(\bar{x})$ to express that $\operatorname{Rg} \xi_\pi \subseteq \bar{x}$. 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.

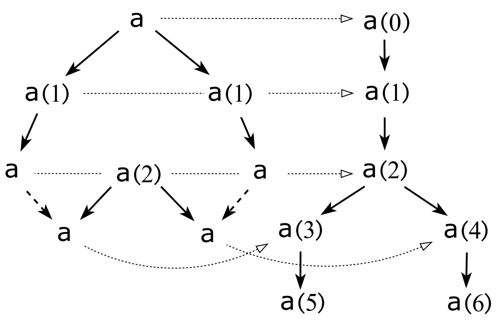

Figure 1: Homomorphisms witness satisfaction (solid and dashed arrows are child and descendant relations).

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.

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.

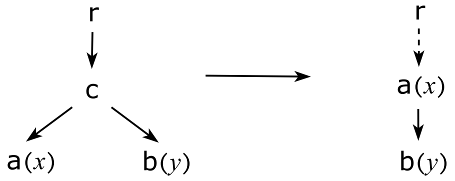

Figure 2: Dependencies are expressed with patterns.

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:

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

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 values (or nodes, identified with subtrees), and variables can store all of these as well. The expression $\sigma(x_i)\big[q'(\bar{x})\big]$ returns the tree obtained by plugging the forest returned by q' below a root node 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 $q''(\bar{x},y)$ 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 freshnull() returning a fresh null value at each call is available. An implementing query can be written as

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

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}:

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

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 $\mathsf{r} \to \mathsf{a}; \mathsf{a} \to \mathsf{ab} \mid \mathsf{db}$ 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**

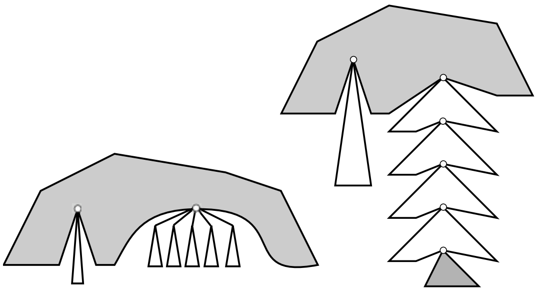

Figure 3: Combining trees horizontally and vertically.

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

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

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

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:*

(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* for $\mathcal{K}$. 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 the context obtained by replacing each tree Z_u by a port 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, the idea is to find, for each dependency $\delta \in \Delta$, a target tree T_δ consistent with a single king $\mathcal{K}$. The 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 $F_1, F_2, \ldots, F_n$ there are forests $I_1, I_2, \ldots, I_{n-1}$ such that $F_1 + I_1 + F_2 + I_2 + \ldots + F_n$ is compatible; for compatible trees $S_1, S_2, \ldots, S_n$ there are multicontexts $I_1, I_2, \ldots, I_{n-1}$ 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 $C_1, C_2, \ldots, C_n$ there are contexts $I_1, I_2, \ldots, I_{n-1}$ 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 $T^1, T^2, \ldots, T^n \in L(\mathcal{K}(\bar{c}))$ 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 $T^1_u, T^2_u, \ldots, T^n_u$. 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.

$\mathcal{K}(\bar{c})$

$$L_{u_1} = L(\langle \mathsf{b}, \{\mathsf{b} \to (\mathsf{b}|\circ)\,\mathsf{c};\ \mathsf{c} \to \varepsilon\}\rangle)$$
$$L_{u_2} = L(\langle \mathsf{c}^*, \{\mathsf{c} \to \varepsilon\}\rangle)$$

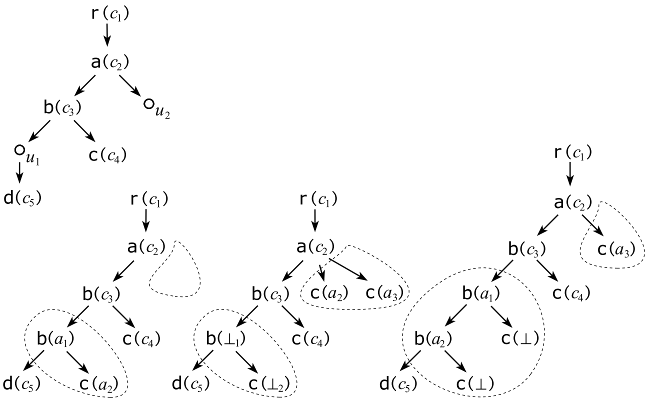

Figure 4: A data kind $\mathcal{K}(\bar{c})$ and three trees in $L(\mathcal{K}(\bar{c}))$.

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

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 the full version . Here we give an example for a relatively generic mapping.

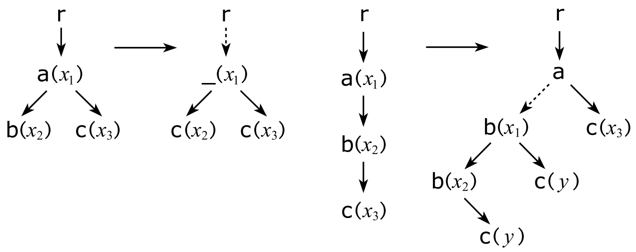

Figure 5: A generic mapping.

EXAMPLE 2. Let $\mathcal{M}$ be a mapping with source DTD $\mathcal{D}_s$: $\mathsf{r} \to \mathsf{a}^*$; $\mathsf{a} \to \mathsf{b\,c}$; $\mathsf{b} \to \mathsf{c}$, target DTD $\mathcal{D}_t$: $\mathsf{r} \to \mathsf{a}$; $\mathsf{a} \to \mathsf{b\,c}^*$; $\mathsf{b} \to (\mathsf{b} \,|\, \mathsf{d})\mathsf{c}$, 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 $\bar{c} = c_1, c_2, c_3, c_4, c_5$, 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 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 $y_1 := \mathsf{freshnull}()$ return
  let $y_2 := \mathsf{freshnull}()$ return
    let $y_3 := \mathsf{freshnull}()$ 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,

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

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 $sol_{\mathcal{K}}(\bar{z})$.

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:

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

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. 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 $|\mathcal{K}_i| \le K$, $\|\mathcal{K}_i\| \le \|D_t\|$, 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 $\neg\mathsf{empty}(const_{\mathcal{K}_1})$ then let $\bar{z} := const_{\mathcal{K}_1}$ return $sol_{\mathcal{K}_1}(\bar{z})$ else
          $\vdots$
if $\neg\mathsf{empty}(const_{\mathcal{K}_k})$ 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 $x_\ell = x_{\ell'}$ or $x_\ell \neq x_{\ell'}$ 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 $z_\ell = z_{\ell'}$ or $z_\ell \neq z_{\ell'}$ 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. The resulting query can be plugged in instead of $const_{\mathcal{K}}$ in 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

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

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. 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. 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 the mapping $\mathcal{M}$.

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 $\mathcal{K}^s_1, \ldots, \mathcal{K}^s_k$ covering $L(\mathcal{D}_s)$. Our idea is based on the following theorem.

THEOREM 5 ([8]). *For absolutely consistent mapping $\mathcal{M}$, target kinds $\mathcal{K}^t_1, \ldots, \mathcal{K}^t_k$ 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 $L(\mathcal{K}^t_i(\bar{d}))$. Moreover, for each i, such $\bar{d}$ can be found in time $K^{K \cdot poly(\|\mathcal{M}\|)}$, where $K = \max(|\mathcal{K}^s|, |\mathcal{K}^t_i|)$, assuming $\|\mathcal{K}^s\| + \|\mathcal{K}^t\| = 2^{poly(\|\mathcal{M}\|)}$.*

Thus if we want to determine the right data kind for $T \in L(\mathcal{K}^s_i)$, the only interesting data values in T are those in the nodes corresponding to the ordinary nodes of $\mathcal{K}^s_i$. 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 $v_{i_{m-1}} \xleftarrow{}^{p} v_\ell$ 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 $P_u = \{\sigma_1, \sigma_2, \ldots, \sigma_q\}$. □

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 $values_{|\bar{z}|}$

$$\mathsf{let}\ \bar{c} := const'_{\mathcal{K}^s}\ \mathsf{return}$$
$$\mathsf{if}\ \mathcal{E}_1(\bar{c})\ \mathsf{then}\ \bar{d}_1\ \mathsf{else}\ \ldots\ \mathsf{if}\ \mathcal{E}_k(\bar{c})\ \mathsf{then}\ \bar{d}_k\ .$$

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.

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 $\mathcal{K}^s_1, \mathcal{K}^s_2, \ldots, \mathcal{K}^s_n$ covering $L(\mathcal{D}_s)$, such that $|\mathcal{K}^s_i| \le K$, $\|\mathcal{K}^s_i\| = \mathcal{O}(\|\mathcal{D}_s\| \cdot |\Gamma|^p)$, 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 $\mathcal{M}$.*

In the notation of Lemma 9, $q_{\mathcal{M}}$ can be defined as

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

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 $\sigma \to \hat{\tau}_1\hat{\tau}_2\ldots\hat{\tau}_n$, where $\tau_1, \tau_2, \ldots, \tau_n$ 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.*

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

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

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, 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?

Acknowledgments. This work was partially supported by the Polish Ministry of Science grant nr N N206 567840. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement n. 267959.

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.