Propagating XML Keys to Relations

Susan Davidson*
University of Pennsylvania
susan@cis.upenn.edu

Wenfei Fan†
Bell Laboratories
wenfei@research.bell-labs.com

Carmem Hara
Universidade Federal do Parana, Brazil
carmem@inf.ufpr.br

Abstract

Although much work has been done on specifying relational storage strategies for XML [30, 31, 6, 29, 27, 28], this paper is a first attempt to study the problem of how XML constraints are propagated to a relational view of XML data. We consider an important class of constraints, XML keys [8], and investigate their associated propagation problem: whether a functional dependency must hold on the relational view provided that a set of XML keys holds on the XML data. To define the relational view, we adopt a simple specification language for transforming XML data to relations and demonstrate that it is capable of expressing transformations commonly found in practice. For this language we provide a polynomial time algorithm to determine propagation of XML keys. In contrast, we show that for an extension of our specification language the key propagation problem is undecidable.

These results are important in designing consistent and efficient relational storage for XML data (e.g. normal forms), and in formulating and optimizing XML queries and updates via its relational view. In addition, our algorithm for key propagation can be generalized and incorporated into relational storage techniques published in the literature (e.g. [30, 31, 28]). These results are also useful in understanding XML to XML transformations, which subsume XML to relational transformations. In particular, as an immediate consequence, the key propagation problem is undecidable when XQuery [11] is used to specify transformations since it subsumes the undecidable extension of our specification language; and a fragment of XQuery [11] and other popular query languages possesses a PTIME algorithm for XML key propagation.

Keywords: XML, constraint propagation, keys, functional dependency, transformation.

*Supported by NSF DBI99-75206
†Currently on leave from Temple University. Supported in part by NSF IIS 00-93168.

1 Introduction

Over the past five years, XML has become enormously popular as a data exchange format. Within bioinformatics, for example, it has been widely embraced since it captures almost automatically the complex, deeply nested data found in databases such as SwissProt [3] and Genbank [5], as well as the often incomplete (semistructured) data being generated by experimental techniques such as gene expression arrays [18]. A common paradigm here as well as in many other application areas is for a data provider to store its data using database technology and export it using XML; on the other end, the data consumer imports the XML data and stores it using database technology.

Since the XML data being transmitted is often large in size and fairly regular in structure (termed "data-oriented" in [17]), the consumer database technology is frequently relational. Often, the consumer database has already been designed and the XML data must be mapped or transformed into an existing schema. For example, the GUS database at the Penn Center for Bioinformatics imports XML gene expression data and maps portions of it into their relational schema using a Perl object layer. In other cases, the consumer database will be created to store the XML data using automatic translations or ones that involve transformations specified by humans.

Although useful as a data exchange language, a recognized problem with XML is that it is only syntax and does not carry the semantics of the data. In particular, if the provider of the XML data is using database technology, the semantics of the data – captured in relational schemas as domain constraints, keys, and foreign keys – is completely lost outside of the surrounding documentation. However, it is clear that information about constraints can help generate a good design, or determine when an existing design is incompatible with the data being imported.

For example, suppose the XML data of figure 1 is being exchanged and that the consumer database has a single table `Chapter` with fields `bookTitle`, `chapterNum` and `chapterName` (written `Chapter(bookTitle, chapterNum, chapterName)`). The table is populated from the XML data as follows: For each `book` element, the value of the `title` subelement is extracted. A tuple is then created in the `Chapter` relation for each `chapter` subelement containing the `title` value for `bookTitle`, the `number` value for `chapterNum`, and the `name` value for `chapterName` (see figure 2(a) for the resulting relational instance.) The key of the `Chapter` table has been specified as `bookTitle` and `chapterNum`. While importing this XML data, violations of the key constraint are detected because two different books have the same title ("XML") and disagree on the name of chapter one ("Introduction" versus "Getting Acquainted"). After digging through the documentation accompanying the XML data, the database designers decide to change the schema to `Chapter(isbn, chapterNum, chapterName)` with a key of `isbn` and `chapterNum` (populated in the obvious way

```
⟨db⟩
  ⟨book  isbn=123⟩
    ⟨title⟩ XML ⟨/title⟩
    ⟨author⟩ Tim Bray ⟨/author⟩
    ⟨chapter  number=1⟩ ⟨name⟩ Introduction ⟨/name⟩ ⟨/chapter⟩
    .  .  .
    ⟨chapter  number=10⟩ ⟨name⟩ Conclusion ⟨/name⟩ ⟨/chapter⟩
  ⟨/book⟩
  ⟨book  isbn=234⟩
  ⟨title⟩ XML ⟨/title⟩
    ⟨chapter  number=1⟩ ⟨name⟩ Getting Acquainted ⟨/name⟩
      ⟨section  number=1⟩ ⟨name⟩ Fundamentals ⟨/name⟩
        ⟨section  number=1⟩ ⟨name⟩ Attributes ⟨/name⟩ ⟨/section⟩
      ⟨/section⟩
    ⟨/chapter⟩
    .  .  .
  ⟨/book⟩
⟨/db⟩
```

Figure 1: Sample XML data

Chapter	bookTitle	chapterNum	chapterName
	XML	1	Introduction
	XML	10	Conclusion
	XML	1	Getting Acquainted

(a)

Chapter	isbn	chapterNum	chapterName
	123	1	Introduction
	123	10	Conclusion
	234	1	Getting Acquainted

(b)

Figure 2: Sample relational instances

from the XML data). The resulting relational instance is shown in figure 2(b). While importing the XML data, no violations of the key constraint are detected. However, the database designers are not sure whether they were lucky with this particular XML data set, or whether such violations will never occur.

It turns out that given the following constraint information about the XML data, using the techniques in this paper the designers of the consumer database could prove that the key of `Chapter` in their modified design is correct:

1. isbn uniquely identifies a book element.

2. Within each book, number is a key for chapter.

3. Within each book, each chapter has a unique name.

Although this example is very simple and the designers could have intuited the correct design from the names of attributes, it is typical of the types of problems encountered when importing biomedical data (for example, into the GUS database). In this domain, the semantics of attributes are often difficult to understand and fields which appear to be keys (e.g. those labeled "ID") may not be keys. Explicitly stating the semantics of data would help tremendously.

In response to the criticism that XML is only syntax, a number of proposals for capturing constraints in XML have recently emerged [32, 10]. Several definitions of keys for XML have been introduced [7, 23, 8], and aspects of these proposals are finding their way into XML Schema [32]. In particular, the proposal of [8] presents a very general definition in which keys are specified in terms of path expressions (i.e. one or more attributes, subelements or more general structures) and can be scoped within an arbitrary substructure of the document. For example, the constraint that a book element is identified by its isbn subelement is scoped at the top level of the document (an *absolute* key), while the constraint that number is a key for chapter is scoped within the context of a book element (a *relative* key). In this proposal, keys are independent of the type (or DTD) of the document and can be reasoned about efficiently [9].

However, none of these proposals has considered the problem of mapping constraints through a transformation of the XML data as illustrated for the consumer database. Note that the consumer database can be thought of as a view, and that the problem of mapping constraints through views has been well-studied in the context of relational data [21, 22, 25, 4] and more recently object and nested relational data models [15]. Being able to reason about constraints on views is not only important for data integration, but also plays an important role in query optimization.

In this paper, we address the problem of mapping key constraints through relational views for XML, referred to as the *XML key propagation problem*. Specifically, using the proposal in [8] and inference techniques of [9], we present an algorithm for determining whether a functional dependency on a relational database necessarily holds given a set of XML keys. An important component of this framework is the ability to specify the mapping from XML to the relational database rather than using one which is automatically generated from a DTD (as in, for example, [30]). This allows us to deal with consumer databases that are pre-existing or independently defined.

The main contributions of the paper are the following:

- We propose a simple language that is capable of specifying transformations from XML data to relational databases of any pre-defined schema, and is independent of DTDs and other schema for XML.
- We develop a PTIME algorithm for determining the propagation of XML keys into relational functional dependencies. Given a transformation σ expressed in our language and a set Σ of XML keys, the algorithm determines whether a relational functional dependency is propagated from Σ through σ. This algorithm can be naturally generalized to derive a complete set of functional dependencies from Σ.
- We show that in general, the XML key propagation problem is undecidable. Specifically, it is undecidable when the transformation language is powerful enough to express all relational algebra operators. Since XML to relational transformations are subsumed by XML to XML transformations expressible in query languages such as XQuery [11], this negative result applies to most popular XML query languages.

Related Work. Several approaches have been explored for using a relational database to store XML documents, and can be classified into two categories: those that represent the document as a graph [26] or set of paths [29], and those that capture the structure of the data by grouping together certain subelements [30, 6]. For those in the first category, very little can be said about how constraints in the XML document are propagated to the database, since the connection between subelements is lost when mapped to the relational database. In contrast, approaches in the second category take advantage of DTDs [30] or XML Schema [6] to generate relational storage mappings that maintain related data in the same relation, and hence dependencies may be preserved. Our framework allows the grouping of subelements by means of variable scoping rules, and does not require a DTD or any other type of schema. The results of this paper on deriving functional dependencies from XML keys could be combined with techniques in the second category to automatically generate mappings through which constraints can be reasoned about (see section 3 for an example). It should be mentioned that since our approach preserves the structure of the data, one can simply use the technique of [31] to query XML data with our transformations.

The mapping language developed in this paper is similar to that proposed by the Stored project [14]. While our path language for transformation specifications is more expressive, Stored has features that we do not support, such as label variables. The new release of Oracle (9i) [28] also supports a new datatype called XML Type with member functions to traverse it using XPath. With this functionality, a relation can be created and populated using a Table function in a way similar to our mapping language.

To the best of our knowledge, work closely related to XML constraint propagation only includes the query rewriting method of CRPQ [16] and the CPI algorithm [24]. In [16], a chase/backchase method for query rewriting of [15] has been extended for a semistructured data model. The method can be used for determining constraint propagation through views when views are expressed in CRPQ (conjunctive regular path queries) and dependencies are DERPDs (disjunctive embedded regular path dependencies). Despite its generality, DERPDs do not capture the notion of tree equality used in the definition of XML keys in [9]. Therefore it could not be directly applied to XML key propagation problem.

CPI is orthogonal to our work because it derives constraints from DTDs (the so-called type constraints), while we propose a transformation language and key propagation analysis that is independent of a DTD. If both DTDs and XML keys are present, we could use both approaches for determining constraint propagation.

Organization. The rest of the paper is organized as follows. Section 2 reviews the relational model and functional dependencies, as well as the XML model and XML keys. Section 3 presents our transformation language. For this language, section 4 provides a PTIME algorithm for determining XML key propagation. Section 5 identifies directions for future work.

2 XML keys and constraint propagation

This section formally defines transformations from XML data to relational databases and the associated key constraint propagation problem. To do so, we start by reviewing the data models and integrity constraints involved in transformations, namely, the relational data model and functional dependencies, and the XML tree model and keys for XML [8]. We also present a path language that is used to define both XML keys and our transformation language.

2.1 The relational model and XML tree model

A *relation schema* R is specified with a set of fields (attributes), denoted by $att(R)$. A *relational schema* $\mathbf{R}$ is a collection of relation schemas $(R_1, \ldots, R_k)$. We write $I(R)$ for the set of instances of R, and $\mathbf{I(R)}$ for the set of relational instances of $\mathbf{R}$.

A *functional dependency (FD)* ϕ on a relation schema R is expressed as $X \to Y$, where X and Y are subsets of $att(R)$. A relation I of R *satisfies* ϕ, denoted by $I \models \phi$, iff for any two tuples t_1, t_2 in I, if $\pi_X(t_1) = \pi_X(t_2)$ then $\pi_Y(t_1) = \pi_Y(t_2)$, where $\pi_X(t)$ means the projection of tuple t on the set X of its attributes. That is, if two tuples agree on their X attributes (i.e., they have the same X-attribute values), then they must agree on their Y attributes. Since an attribute has an atomic

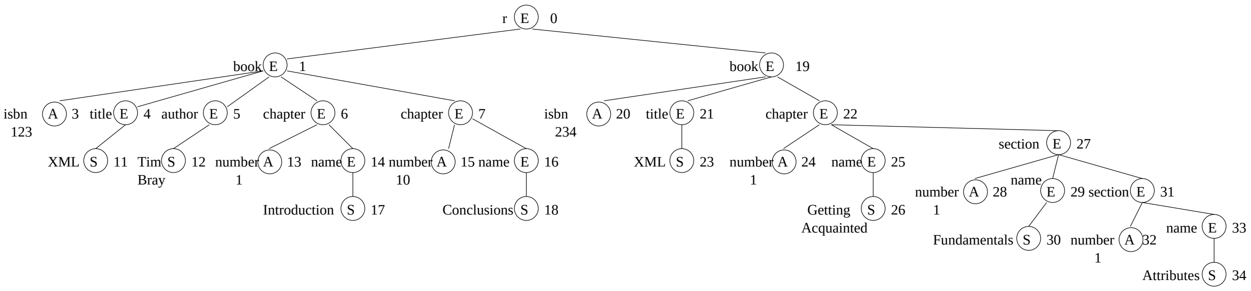

Figure 3: Tree representation of XML data

value (e.g., integer, string), the definition of FDs involves equality on atomic values[1].

In contrast to the flat (First Normal Form) structure of relational data, XML data is hierarchically structured and is modeled as a node-labeled tree. For example, figure 3 shows the tree representation of part of the XML data of figure 1. In this figure, nodes are classified into three types by their labels: element nodes (E), attribute nodes (A), and text nodes (S). To the left of each node is its name and/or value; to the right of each node is its identity. Text nodes have no name but carry text, attributes both have a name and carry text, and elements have a name but do not carry text. To model these we assume three pairwise disjoint sets of labels: $\mathbf{E}$ of element tags, $\mathbf{A}$ of attribute names, and a singleton set $\{\mathtt{S}\}$ denoting text (PCDATA). These can then be formalized as follows (cf. [8]).

Definition 2.1: An *XML (document) tree* is specified as $T = (V, lab, ele, att, val, id, r)$, where

1. V is a finite set of nodes;
2. lab is a mapping $V \to \mathbf{E} \cup \mathbf{A} \cup \{\mathtt{S}\}$ which assigns a label to each node in V; a node v in V is called an *element* (*E node*) if $lab(v) \in \mathbf{E}$, an *attribute* (*A node*) if $lab(v) \in \mathbf{A}$, and a text node (*S node*) if $lab(v) = \mathtt{S}$;
3. ele and att are partial mappings that define the edge relation of T: for any node v in V
   - if v is an element then $ele(v)$ is a *sequence* of elements and text nodes in V and $att(v)$ is a *set* of attributes in V; for each v' in $ele(v)$ or $att(v)$, v' is called a *child* of v and we say that there is a *parent-child* edge from v to v';
   - if v is an attribute or a text node then $ele(v)$ and $att(v)$ are undefined;
4. val is a partial mapping that assigns a string to each attribute and text node: for any node v in V, if v is an A or S node then $val(v)$ is a string, and $val(v)$ is undefined otherwise;

[1] Comparisons of a `null` value with any value do not evaluate to `true` (it is treated as `unknown`).

5. id is a mapping that assigns a unique identity to each node in the tree;

6. r is the unique and distinguished root node.

An XML tree has a tree structure, i.e., for each $v \in V$, there is a unique path of parent-child edges from root r to v. □

In contrast to value equality on atomic values, we need a general notion of equality on subtrees when comparing the "values" of two nodes. Two nodes n_1, n_2 are *value equal* if they have the same tag (label) and in addition, either they have the same (string) value (when they are `S` or `A` nodes) or their children are pairwise value equal (when they are `E` nodes). Observe that for any nodes x, y, if $x = y$, i.e., if they are the same node, then $x =_v y$. However, $x =_v y$ does not necessarily mean $x = y$. For example, in figure 3, nodes 13 and 24 are value equal but are distinct nodes; the same holds for nodes 4 and 21.

2.2 Path expressions

Before we move to the definition of XML keys, we first describe two path languages PL_s and PL. These languages are not only used in key specifications but also in the transformation language.

Path language	Syntax
PL_s	$\rho \ ::= \ \epsilon \mid l.\rho$
PL	$P \ ::= \ \epsilon \mid l \mid P.P \mid \_*$

Here ϵ represents the empty path, node label $l \in \mathbf{E} \cup \mathbf{A} \cup \{\mathtt{S}\}$, "." denotes concatenation of two path expressions. "_*" is a combination of wildcard (which matches any label) and Kleene closure. A path of PL_s is a sequence of labels, called a *simple path*, which represents a path in an XML tree. An expression P of PL defines a set of simple paths, as "_*" can match any path. For example, $book.author.name$ is a simple path while $\_*.name$ is an expression of PL. Observe that these path languages are common fragments of regular path expressions [20] and XPath [12].

Let ρ be a path in PL_s, P be a path expression in PL and T be an XML tree. We use $\rho \in P$ to denote that path ρ is in the regular language defined by path expression P. For example, $book.author.name \in book.author.name$ and $book.author.name \in \_*.name$.

We say that a node n_2 in T is *reachable* from node n_1 by following the path ρ, denoted by $T \models \rho(n_1, n_2)$, iff (1) $n_1 = n_2$ if $\rho = \epsilon$, and (2) if $\rho = \rho'.l$ then there is node n such that $T \models \rho'(n_1, n)$ and n_2 is a child of n with label l. More generally, we say that node n_2 is *reachable* from n_1 by following path expression P, denoted by $T \models P(n_1, n_2)$, iff there is a path $\rho \in P$ such that $T \models \rho(n_1, n_2)$. Given a node n in T, we write $n[\![P]\!]$ for the set of nodes in T that can be reached by following P from n. In particular, if n is the root r, we write $[\![P]\!]$ for $n[\![P]\!]$.

2.3 XML Keys

The definition of keys given in [8] has several salient features: First, keys are defined in terms of one or more path expressions, i.e. they may involve one or more attributes, subelements or more general structures. Equality is defined on tree structures instead of on simple text, referred to as *value equality*. Second, keys can be scoped within the context of the entire document (an *absolute* key), or within the context of a particular part of the document (a *relative* key). An absolute key is a special case of a relative key. Third, the specification of keys is orthogonal to the typing specification for the document (e.g. DTD or XML Schema). The type of documents will therefore be ignored throughout this paper.

Definition 2.2: A *key constraint* φ for XML is an expression of the form

$$(Q,\ (Q',\ \{P_1, \ldots, P_k\})),$$

where Q, Q' are PL expressions and P_i's are paths of PL_s. The path Q is called the *context path*, Q' the *target path*, and P_1, ..., P_k the *key paths* of φ. When $Q = \epsilon$, we call φ an *absolute key*; otherwise, it is a *relative key*. We write $(Q',\ \{P_1, \ldots, P_k\})$ for an absolute key. We use $\mathcal{K}$ to denote the language of keys. □

A key φ specifies the following: (1) the context path Q, starting from the root of an XML tree T, identifies a set of nodes $[\![Q]\!]$; (2) for each node $n \in [\![Q]\!]$, φ defines an absolute key $(Q',\ \{P_1, \ldots, P_k\})$ that is to hold in the subtree rooted at n; specifically,

- the target path Q' identifies a set of nodes $n[\![Q']\!]$ in the subtree, referred to as the *target set*,
- the key paths $P_1, \ldots, P_k$ provide an identification for nodes in the target set, i.e., for each $n' \in n[\![Q']\!]$ the values of the nodes reached by following these key paths from n' uniquely identify n' in the target set. In other words, if two nodes in $n[\![Q']\!]$ are distinct, then the two sets of nodes reached on some P_i must be disjoint by value equality.

Drawing an analogy with keys in relational databases, the target path Q' corresponds to a relation name in the relational model, and the set of key paths P_1, ..., P_k corresponds to the set of attributes in the key for a relation. There is no relational counterpart to the context path Q since relations are flat (not hierarchical).

Using this syntax, the first two sample constraints from section 1 can be written as follows:

- KS_1 : $(\epsilon, (\_ * .book, \{isbn\}))$: within the context of the entire document (an absolute key), any book element is identified by its *isbn* attribute. Note that the book element can occur anywhere in the tree.

- KS_2 : $(\_*.book, (chapter, \{number\}))$: within the context of any subtree rooted at book node, a chapter is identified by its number attribute. Note that the chapter must be immediately under the book node.

Definition 2.3: An XML tree T *satisfies* φ, denoted by $T \models \varphi$, iff for any n in $[\![Q]\!]$ and any n_1, n_2 in $n[\![Q']\!]$, if for all $i \in [1, k]$ there are nodes $x \in n_1[\![P_i]\!]$, $y \in n_2[\![P_i]\!]$ such that $x =_v y$, then $n_1 = n_2$. That is, $\forall n \in [\![Q]\!] \; \forall n_1 \, n_2 \in n[\![Q']\!] \;\; ((\bigwedge_{1 \le i \le k} \exists \, x \in n_1[\![P_i]\!] \; \exists \, y \in n_2[\![P_i]\!] \; (x =_v y)) \rightarrow n_1 = n_2)$. □

Note that the XML tree of figure 3 satisfies our sample constraints. For example, the first constraint is satisfied since $[\![\_ * .book]\!] = \{1, 19\}$ and $1[\![isbn]\!] \cap 19[\![isbn]\!] = \{123\} \cap \{234\} = \emptyset$. The second constraint can be checked by verifying the absolute key $(\epsilon, (chapter, \{number\}))$ in the context of each of the subtree rooted at 1 and the subtree rooted at 19.

Since keys will be used to uniquely specify components of a document, we need to know whether or not a set of keys is *transitive* [8]. For example, $\{KS_2\}$ is *not* transitive since with it alone there is no way to uniquely identify a book in the document, which is necessary before identifying a chapter of that book.

To formalize this, we start with the notion of precedes: $(Q_1, (Q'_1, S_1))$ *immediately precedes* $(Q_2, (Q'_2, S_2))$ if $Q_2 = Q_1.Q'_1$. The *precedes* relation is the transitive closure of the immediately precedes relation.

A set Σ of keys is *transitive* if for any relative key $(Q_1, (Q'_1, S_1)) \in \Sigma$ there is an absolute key $(\epsilon, (Q'_2, S_2)) \in \Sigma$ which precedes $(Q_1, (Q'_1, S_1))$. A set of transitive keys can uniquely identify a component within the entire document. As an example, $\{KS_1, KS_2\}$ is transitive since any chapter in the document can be identified by providing the `isbn` of a book and the `number` of a chapter.

A special case of keys is worth mentioning. Consider $\varphi = (Q, (Q', \emptyset))$, in which the set of key paths of φ is empty. Observe that for any XML tree T, $T \models \varphi$ iff for any $n \in [\![Q]\!]$, if there exists a node $n' \in n[\![Q']\!]$ then n' is unique. In other words, there is at most one node in $n[\![Q']\!]$. The last constraint from section 1 is an example of such a key:

- KS_3 : $(\_ * .book, (chapter.name, \{\}))$: within the context of any subtree rooted at a book node, there can be at most one *chapter.name* path (i.e. each chapter has a unique name).

We shall use this special case in the study of key constraint propagation.

2.4 Problem statement

A *transformation* from XML data to relational databases of schema $\mathbf{R}$ is a mapping σ such that for any XML tree T, $\sigma(T)$ is an instance of $\mathbf{R}$. A transformation defines a (materialized) view for XML data in some transformation specification language.

The question of *key constraint propagation* asks whether a relational view of XML data preserves certain semantic information specified by XML key constraints. It is to determine, given a transformation σ from XML data to relational databases of schema $\mathbf{R}$, and an XML tree T satisfying a set Σ of XML keys, whether $\sigma(T)$ satisfies a functional dependency φ (on a schema R in $\mathbf{R}$). We write $\Sigma \models_\sigma R : \varphi$ if the implication holds for all XML trees satisfying Σ, and refer to φ as a FD *propagated from* Σ. More specifically, with respect to a transformation specification language, the *key constraint propagation problem* can be stated as:

INPUT:	A transformation σ from XML data to relational databases of schema $\mathbf{R}$, a set Σ of XML key constraints, and a FD φ over a schema R in $\mathbf{R}$.
QUESTION:	Is it the case that $\Sigma \models_\sigma R : \varphi$?

Observe that the key constraint propagation problem is general as it does not require that the XML data conforms to any DTD or other type specification.

3 Transformation language

We now present our language for specifying transformations from XML data to relations. Although the language is quite simple, it is powerful enough to specify many common transformations found throughout the literature, in particular those of [30]. Since it is declarative, it is also easy to understand (cf. [13]). Most importantly, it possesses a polynomial time algorithm for the key constraint propagation problem, which is the subject of section 4. In contrast, we show that for some extensions to this language the key constraint propagation problem becomes undecidable.

3.1 Transformation specifications

A transformation is specified with a set of rules. Given a relational schema $\mathbf{R} = (R_1, \ldots, R_n)$, we specify a mapping σ from XML data to $\mathbf{I(R)}$ by defining a mapping $\mathtt{Rule}(R_i)$ for each R_i

in $\mathbf{R}$, referred to as the *table rule* for R_i, to generate an instance of R_i from XML data. The transformation σ is defined to be

$$\sigma = (\texttt{Rule}(R_1), \ldots, \texttt{Rule}(R_n)).$$

Specifically, a table rule is defined in terms of a set of rules that populate the fields (columns) of the table. That is, for each field $l \in att(R_i)$, we also define a mapping $\texttt{rule}(R_i, l)$, referred to as the *field rule* for the l field of R_i. For each R_i, assume a set of variables X_i in which x_r is a distinguished variable. Using these variables, $\texttt{rule}(R_i, l)$ is expressed as one of the following:

$$l : value(x) \quad | \quad l : id(x),$$

where $x \in X_i$ that is to range over nodes in an XML tree, $value(x)$ is the serialized value of the subtree rooted at x (to be explained later), and $id(x)$ is the identity of node x.

The table rule $\texttt{Rule}(R_i)$ for R_i can then be specified with a set of field rules

$$\texttt{Rule}(R_i) = \{\texttt{rule}(R_i, l) \mid l \in att(R_i)\}$$

along with a set of variable mapping rules of the form:

$$x \leftarrow y.P,$$

where $x, y \in X_i$ and P is a path expression of PL. Intuitively, referring to an XML tree T, if the variable y denotes a node in T, then the variable x ranges over nodes in $y[\![P]\!]$. That is, these variable mappings select nodes of the tree with PL expressions, and the relation (table) is populated with the identities or values of these nodes. In particular, the variable x_r denotes the root of the tree. We shall write $\texttt{rule}(l)$ for $\texttt{rule}(R_i, l)$ when R_i is clear from the context.

For example, a transformation of the XML data of figure 1 to a relational schema could be specified as follows:

$\sigma = (\texttt{Rule}(\text{book}), \texttt{Rule}(\text{chapter}), \texttt{Rule}(\text{chapSec}), \texttt{Rule}(\text{section}))$

$\texttt{Rule}(\text{book}) = \{$ isbn: value(x_1), title: value(x_2), author: value(x_3)$\}$,
$x_b \leftarrow x_r.\_^*$.book, $x_1 \leftarrow x_b$.isbn, $x_2 \leftarrow x_b$.title, $x_3 \leftarrow x_b$.author;

$\texttt{Rule}(\text{chapter}) = \{$ inBook: value(y_1), number: value(y_2), name: value(y_3)$\}$,
$y_b \leftarrow x_r.\_^*$.book, $y_1 \leftarrow y_b$.isbn, $y_c \leftarrow y_b$.chapter, $y_2 \leftarrow y_c$.number, $y_3 \leftarrow y_c$.name;

$\texttt{Rule}(\text{chapSec}) = \{$ inChapt: value(z_1), sid: id(z_2), number: value(z_3), name: value(z_4)$\}$,
$z_c \leftarrow x_r.\_^*$.book.chapter, $z_1 \leftarrow z_c$.number, $z_2 \leftarrow z_c$.section, $z_3 \leftarrow z_2$.number, $z_4 \leftarrow z_2$.name;

$\texttt{Rule}(\text{section}) = \{$ inSec: id(x_s), sid: id(x'_s), number: value(x_1), name: value(x_2)$\}$,
$x_s \leftarrow x_r.\_*$.section, $x'_s \leftarrow x_s$.section, $x_1 \leftarrow x'_s$.number, $x_2 \leftarrow x'_s$.name.

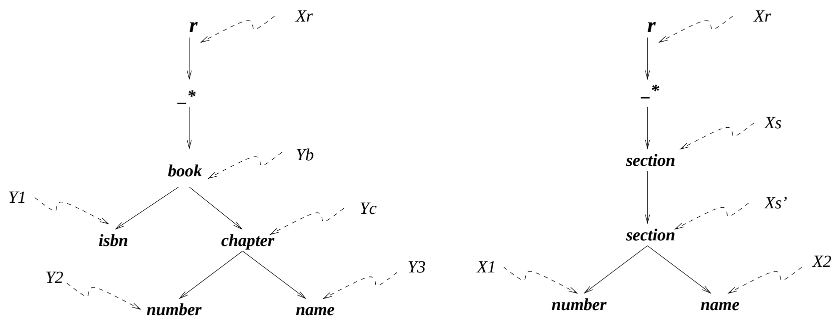

Figure 4: The table trees representing `Rule`(chapter) and `Rule`(section)

We say that a table rule `Rule`(R_i) is *well-formed* iff

1. each variable $x \in X_i$ is *connected* to the root r; that is, x is specified with either $x \leftarrow x_r.P$ in the rule, or $x \leftarrow y.P$ and y is connected to the root r;
2. for any $x_1 \leftarrow y.P_1$ and $x_2 \leftarrow y.P_2$ in the rule, P_1 and P_2 do not share a common prefix, i.e., there exists no PL expressions P, P_1', P_2' such that $P_1 = P.P_1'$ and $P_2 = P.P_2'$ and $P \neq \epsilon$; moreover, for any $x \leftarrow y.P$, P is a simple path (i.e., without "$\_*$") unless y is x_r;
3. for any field rules $l_1 : f_1(x_1)$ and $l_2 : f_2(x_2)$, if l_1 and l_2 are different labels, then either x_1 and x_2 are different variables or f_1 and f_2 are different functions.

A transformation mapping σ is *well-formed* iff each table rule in σ is well formed. We consider well-formed transformations only.

The first condition ensures that there is some XML tree T for which all mapping rules make sense, i.e. for any $x \leftarrow y.P$ in the rule, $y[\![P]\!]$ is indeed a set of nodes reachable from the root of T. The second condition concerns scoping: it groups two nodes under a common ancestor in T into the same tuple in a relation; and moreover, it does not group "unrelated" data in the same relation to preserve the structure of the data. For this purpose "$\_*$" is allowed and typically used in specifications of the form $x \leftarrow x_r.P$, i.e., from the root; but it is not used in other places to connect nodes that are "far away" from each other. The third condition prevents populating different columns of a relation with the same values, which does not make much sense in practice.

By treating "$\_*$" as a special node label, each well-formed table rule can be represented as a node-labeled tree. For example, the transformation given above is well-formed and the table rules `Rule`(chapter) and `Rule`(section) can be represented as the trees shown in figure 4.

A tree representing a table rule is referred to as a *table tree*. In a table tree T_R representing `Rule`(R), each variable in `Rule`(R) corresponds to a unique node, and each node corresponds to at most one variable (assume that no redundant variables are defined).

We next formally define the semantics of a transformation mapping σ. Given an XML tree $T = (V, lab, ele, att, val, id, r)$, each table rule $\texttt{Rule}(R_i)$ maps T to an instance I_i of R_i. More specifically, given a variable specification $x \leftarrow y.P$, x ranges over $y[\![P]\!]$; x_r is always interpreted as the root r. Given a field rule $\texttt{rule}(l) = l : f(x)$, the l field is populated with values in the set $\{f(x) \mid x \in y[\![P]\!]\}$, where f is either id or $value$. Without loss of generality, let $att(R_i) = \{l_1, \ldots, l_k\}$, $\texttt{rule}(l_i)$ be $l_i : f_i(x_i)$, the set of variables associated with $\texttt{Rule}(R_i)$ be X_i, and each variable x be specified with $x \leftarrow x'.P_x$. Then the instance I_i is generated by the following:

$$I_i = \{(l_1 : f_1(x_1),\ \ldots,\ l_k : f_k(x_k)) \mid x_r = r,\ \forall x \in X_i,\ x \in x'[\![P_x]\!]\}.$$

For example, $\texttt{Rule}(section)$ is interpreted as:

$$\{(inSec : id(x_s),\ number : value(x_1),\ name : value(x_2)) \mid$$
$$x_s \in r[\![\_ * .section]\!],\ x'_s \in x_s[\![section]\!],\ x_1 \in x'_s[\![number]\!],\ x_2 \in x'_s[\![name]\!]\}.$$

We next describe the function f in a field rule. If f is id, then $f(x)$ returns the node identity of x, using the function id of T. If f is $value$, then $f(x)$ returns a string representing the pre-order traversal of the subtree rooted at x. More precisely, the $value$ function is defined with the val function in T as follows:

- if $lab(x) = \mathbf{S}$ or $lab(x) \in \mathbf{A}$, then $value(x)$ is $val(x)$;
- if $lab(x) \in \mathbf{E}$, then $value(x)$ is a string

  $$\text{“(”} \mid lab(x_1) \mid \text{“:”} \mid value(x_1) \mid \text{“,”} \mid \ldots \mid lab(x_k) \mid \text{“:”} \mid value(x_k) \mid \text{“,”} \mid$$
  $$lab(a_1) \mid \text{“:”} \mid value(a_1) \mid \text{“,”} \mid \ldots \mid lab(a_m) \mid \text{“:”} \mid value(a_m) \mid \text{“)”},$$

  where $ele(x) = [x_1, \ldots, x_k]$ and $att(x) = \{a_1, \ldots, a_m\}$, and ‘|’ denotes string concatenation.

For example, $value(6) = (name : (S : Introduction),\ number : 1)$ in figure 3.

Assuming that attributes are alphabetically ordered by their label, it is easy to verify that for any nodes x and y, $x =_v y$ iff $value(x) = value(y)$. Note that the $value$ function is rather general since it covers, but is not restricted to nodes with simple text (`PCDATA`). To see the need for the general notion, suppose that one wants to extract `author` of a book. Some author nodes may have a `S` child carrying their full name, as indicated by node 5 in figure 3, whereas others may have subelements `first-name` and `last-name`. Both cases are handled by our $value$ function uniformly.

As an example, the interpretation of the table rules of σ over the XML tree of figure 3 generates the relational instance of figure 5.

Several subtleties are worth mentioning. First, since XML data is semistructured in nature it is likely that for $x \leftarrow y.P$, $y[\![P]\!]$ is empty. In this case $f(x)$ is defined to be `null`. Second, if $y[\![P]\!]$

book	isbn	title	author
	123	XML	Tim Bray
	234	XML	`null`

chapter	inBook	number	name
	123	1	Introduction
	123	10	Conclusions
	234	1	Getting Acquainted

chapSec	inChapt	sid	number	name
	1	27	1	Fundamentals

section	inSec	sid	number	name
	27	31	1	Attributes
	31	`null`	`null`	`null`

Figure 5: Instances generated by σ for XML tree data.

has multiple elements, then to generate the relation with the table rule containing x, an implicit Cartesian product is computed so that all nodes in $y[\![P]\!]$ are covered in the relation. That is, if $l : f(x)$ is a field rule then for each $x \in y[\![P]\!]$, there is a tuple t whose l field has the value $f(x)$.

Observe that our language is capable of expressing projection (π), Cartesian product ($\times$) and set union ($\cup$; limited in the sense that it is implemented with "_*"). One might be tempted to develop a more powerful language to express all relational algebra operators: projection, selection (σ), Cartesian product, set union and difference ($-$). Indeed, these operators can be generalized to XML trees (e.g., with our *value* function). However, when it comes to the analysis of key propagation, this extension leads to a negative result:

Theorem 3.1: *The key constraint propagation problem from XML to relational data is undecidable when the transformation specification language can express all relational algebra operators.* $\square$

Proof sketch: This can be shown by reduction from the equivalence problem for relational algebra queries: given any relation schema S, R and relational algebra queries Q_1, Q_2 from S to R, determine whether for any I in $I(S)$, $Q_1(I) = Q_2(I)$ (denoted by $Q_1 \equiv Q_2$). The problem is well-known to be undecidable [33]. The idea of our reduction is borrowed from [21].

We give a reduction from the equivalence problem for relational algebra queries. Let Q_1, Q_2 be arbitrary relational algebra queries from S to R, and R' be an extension of R with the addition of a new attribute l, i.e., $att(R') = att(R) \cup \{l\}$ where l is not in $att(R)$ and $att(S)$. We define a mapping σ from XML data to relational databases of schema $\mathbf{R} = \{R'\}$, a set Σ of XML keys, and a relational key φ on R', such that $\Sigma \models_\sigma R' : \varphi$ iff $Q_1 \equiv Q_2$. More specifically, let Σ be the set consisting of a single absolute key: $(S.t,\ att(S))$. That is, in an XML tree T satisfying Σ, the values of $att(S)$ subelements of any $S.t$ node uniquely determine the node. We define the relational key φ to be $\emptyset \to R'$, i.e., there is a unique tuple in the R' relation. Finally, we define the mapping

σ to be

$$\{t\} \cup \text{"2"} \times ((Q_1(\pi_{att(S)}(S.t)) - Q_2(\pi_{att(S)}(S.t))) \cup (Q_2(\pi_{att(S)}(S.t)) - Q_1(\pi_{att(S)}(S.t)))),$$

where t is a R' tuple such that all of its attributes have constant value "1", while in the rest of the tuples in the relation, the l attributes have a constant value "2"; and $\pi_{att(S)}(S.t)$ denotes the values of $att(S)$ subelements of $S.t$ nodes. For any attribute B in $att(S)$, if a $S.t$ node does not have any B subelement, then $\pi_B(S.t)$ is defined to be $null$. If a $S.t$ node has more than one B subelement, then $\pi_B(S.t)$ is a set consisting of the values of all these B subelements, and $\pi_{att(S)}(S.t)$ is the Cartesian product of these sets.

Observe that there is an one-one mapping between $I(S)$ and the $\pi_{att(S)}(S.t)$ values of all XML trees satisfying Σ. Thus $E = (Q_1(\pi_{att(S)}(S.t)) - Q_2(\pi_{att(S)}(S.t))) \cup (Q_2(\pi_{att(S)}(S.t)) - Q_1(\pi_{att(S)}(S.t)))$ equals the empty set $\emptyset$ for all XML trees satisfying Σ iff for all instances I of S, the difference between $Q_1(I)$ and $Q_2(I)$ is $\emptyset$, i.e., when $Q_1 \equiv Q_2$. Note that $\sigma(T) \models \varphi$ iff $\sigma(T)$ consists of a single tuple. However the tuple t is already in $\sigma(T)$, which is distinguished from the others in $\sigma(T)$ by the value of its l attribute. Hence for each XML tree T satisfying Σ, $\sigma(T) \models \varphi$ iff the set E is $\emptyset$. Thus $\Sigma \models_\sigma R' : \varphi$ iff $Q_1 \equiv Q_2$. Thus the encoding is indeed a reduction from the equivalence problem for relational algebra queries to the key constraint propagation problem. Since the former is undecidable, so is the later. This completes the proof of Theorem 3.1. □

It is worth mentioning that popular query languages for XML, such as XQuery [11], subsume our specification language and in fact, an undecidable extension as well. Key propagation problem can be similarly defined for XML to XML transformations, and is equally important in that context. As an immediate result of Theorem 3.1, when these query languages are (naturally) used to specify XML to XML transformations, the key constraint propagation problem is undecidable.

3.2 A case study

Despite its simplicity, our transformation language is capable of expressing transformations specified with other techniques. In particular, we demonstrate that our language can specify the "hybrid inlining" transformation of [30] via an example.

The hybrid algorithm requires a DTD. Figure 6 presents a possible DTD for the book example given in section 2, and a graph representing its structure (called a DTD graph [30]). Assuming that the root element of the document is known, the hybrid algorithm derives a relational schema in the following way. (1) A relation is created for the root element. (2) A relation is created for all element types under a "*" node and for one of the mutually recursive element types; these relations contain $parentID$ and $parentCODE$ attributes to encode the parent-child relationship.

```
⟨!DOCTYPE book [
   ⟨<!ELEMENT book (title, author, chapter*)⟩
   ⟨<!ATTLIST book isbn CDATA #IMPLIED⟩
   ⟨<!ELEMENT chapter (name, section*)⟩
   ⟨<!ATTLIST chapter number CDATA #IMPLIED⟩
   ⟨<!ELEMENT section (name, section*)⟩
   ⟨<!ATTLIST section number CDATA #IMPLIED⟩
   ⟨<!ELEMENT title (#PCDATA)⟩
   ⟨<!ELEMENT author (#PCDATA)⟩
   ⟨<!ELEMENT name (#PCDATA)⟩
]⟩
```

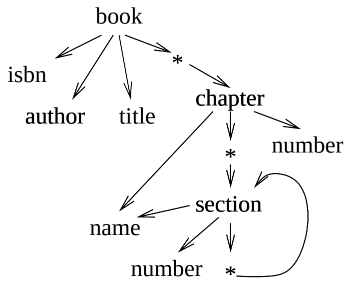

Figure 6: Example of a DTD and its graph

As an example, the schema created for the DTD in figure 6 is the following[2]:

book:	(bookID, book.isbn, book.title, book.author)
chapter:	(chapterID, chapter.parentID, chapter.parentCODE, chapter.number, chapter.name)
section:	(sectionID, section.parentID, section.parentCODE, section.number, section.name)

We can specify the transformation in our language as follows:

`Rule`(book) = { bookID: id(x_b), book.isbn: value(x_1), book.title: value(x_2), book.author: value(x_3)},
    $x_b \leftarrow x_r$.book, $x_1 \leftarrow x_b$.isbn, $x_2 \leftarrow x_b$.title, $x_3 \leftarrow x_b$.author;

`Rule`(chapter) = { chapterID: id(y_c), chapter.parentID: id(y_b), chapter.number: value(y_1), chapter.name: value(y_2)},
    $y_b \leftarrow x_r$.book, $y_c \leftarrow y_b$.chapter, $y_1 \leftarrow y_c$.number, $y_2 \leftarrow y_c$.name;

`Rule`(chapSec) = { sectionID: id(z_s), section.parentID: id(z_c), section.number: value(z_1), section.name: value(z_2)},
    $z_c \leftarrow x_r$.book.chapter, $z_s \leftarrow z_c$.section, $z_1 \leftarrow z_s$.number, $z_2 \leftarrow z_s$.name;

`Rule`(section) = { sectionID: id(x'_s), section.parentID: id(x_s), section.number: value(x_1), section.name: value(x_2)},
    $x_s \leftarrow x_r.\_*$.section, $x'_s \leftarrow x_s$.section, $x_1 \leftarrow x'_s$.number, $x_2 \leftarrow x'_s$.name.

Instead of using the `parentCODE` attribute in `section` to distinguish different types of parent elements, we separate `section` into two relations. With a slight change to our specification language by allowing (1) constants in field-rules and (2) multiple table-rules for the same relation, one can encode the hybrid algorithm exactly using our language. We do not support these changes since,

[2]We omit the `isroot` attribute that the algorithm creates for each subelement, assuming that the root is known.

as demonstrated by the example above, one can always specify transformations in our language that are "equivalent" to those generated by the hybrid algorithm. Note that the support of "$\_*$" in our language allows us to handle XML documents of a recursive DTD.

A major difference between the two approaches is that ours does not require the presence of a DTD or any schema. In practice, many XML documents do not come with a DTD. Moreover, ours allows one to specify transformations to a relational predefined schema, instead of to a fixed one.

4 Propagation of XML keys to functional dependencies

In this section we investigate the key constraint propagation problem and show the following:

Theorem 4.1: *The key constraint propagation problem is decidable in PTIME for XML keys defined in $\mathcal{K}$ and transformations expressed in our language.* □

The proof is given by developing an algorithm that, given a set Σ of XML keys of $\mathcal{K}$, a FD ϕ over a relation schema R and a mapping σ from XML to relational databases expressed in our transformation language, determines whether $\Sigma \models_\sigma R : \phi$ in polynomial time. Recall that $\Sigma \models_\sigma R : \phi$ means that for any XML tree T, if $T \models \Sigma$ then $\sigma(T) \models \phi$.

Before we present our algorithm, we first introduce some notation. Assume that the table rule $\texttt{Rule}(R)$ for R is $\{l_i : f_i(x_i) \mid i \in [1, m]\}$ with a set X of variables and mappings $x \leftarrow y.P$ for each $x \in X$; and the FD ϕ is $Y \to l$ with $l \in att(R)$ and $Y \subseteq att(R)$. Recall that $\texttt{Rule}(R)$ can be represented by a table tree, denoted by T_R; and each variable x in X corresponds to a node in T_R, referred to as the x-node; In particular, the x_r-node is the root of T_R. Observe that for any $x, y \in X$, if the x-node is a descendant of the y-node in T_R, then there is a unique path in T_R from the y-node to x-node, which is a PL expression, denoted by $P(y, x)$. It is easy to verify that this only happens if there are variables $x_1, \ldots, x_k$ in X such that $x_1 = x$, $x_k = y$ and for each $i \in [1, k-1]$, $x_{i+1} \leftarrow x_i.P_i$ is a variable mapping in $\texttt{Rule}(R)$. We use $descendants(y)$ to denote the set of all the variables that are descendants of y; similarly we define $ancestors(x)$. In particular, if x is specified with $x \leftarrow y.P$ then we say that y is the *parent* of x, denoted by $parent(x)$.

It is nontrivial to determine whether the FD $Y \to l$ can be derived from Σ via σ. We now give some intuition behind the algorithm. A simple case is when $l \in Y$, i.e., ϕ is a *trivial* FD, which always holds. If $l \notin Y$, we need a sufficient and necessary condition on the specifications of the field rules for Y and l to decide whether ϕ can be derived. This condition needs to accommodate the semantics of the transformation and its variable scoping rules, as well as to cope with the mismatch

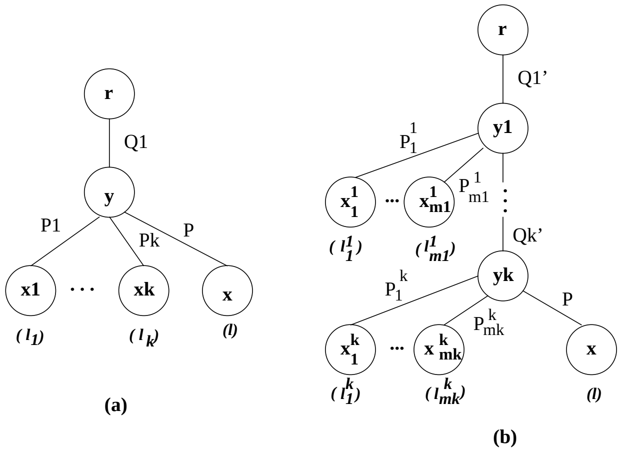

Figure 7: Connection between keys and functional dependencies

between the XML tree model and the relational model. We characterize the condition with a set Θ of XML keys defined in terms of PL expressions in T_R, and reduce the problem of deciding whether $\Sigma \models_\sigma R : \phi$ to deciding whether $\Sigma \models \Theta$. The latter is to determine whether for any XML tree T, if $T \models \Sigma$ then $T \models \Theta$, i.e., implication of XML keys. We have developed an algorithm in [9] for checking implication of XML keys, which is in PTIME, i.e., polynomial in the size of Θ and Σ. Thus the central task is to find Θ. We then check whether $\Sigma \models \Theta$ using the algorithm of [9].

Next we consider the generation of the set Θ of XML keys. The FD says that for any tuples t_1, t_2 in a relation of R, if t_1 and t_2 agree on their Y fields, then they also agree on their l fields. These fields are populated via the table rule `Rule`(R) in σ by extracting the id's or $values$ of XML nodes from all over an XML document, with certain variable scoping rules. To capture these in terms of keys on the original XML data, let us first consider a basic case depicted in figure 7 (a), where l is specified with $l : f(x)$ in `Rule`(R), Y with $l_i : f_i(x_i)$ for $i \in [1, k]$, and f and f_i are either id or $value$. Here x and x_i have a common parent y, i.e., $x \leftarrow y.P$ and $x_i \leftarrow y.P_i$. Moreover, y is specified with $y \leftarrow x_r.Q_1$ with the PL expression Q_1 from the root of T_R. The FD can be captured with an absolute key in Θ:

$$\phi_1 = (Q_1,\ \{P_1, \ldots, P_k\}),$$

which asserts that the values of the nodes reached by following P_i's from a node y in $[\![Q_1]\!]$ uniquely identify y. If this is the case then for any two tuples t_1, t_2 generated from nodes in $[\![Q_1]\!]$, if they agree on their Y fields then they are value equal, and thus (ideally) so are their l fields which are extracted from the nodes reached by following P from the same nodes. This is because, as mentioned earlier, node identity implies value equality. However, ϕ_1 alone is not enough: due to the semistructured nature of XML data, it is possible that for any $y \in [\![Q_1]\!]$, the set $y[\![P]\!]$ may not be a singleton set. Since all the nodes in $y[\![P]\!]$ need to be extracted and grouped together with each value of the Y-fields extracted by following P_i's from y, if $y[\![P]\!]$ has multiple nodes, it literally violates the FD ϕ. Thus we need another key to assure that $y[\![P]\!]$ has a single node:

$$\phi_2 = (Q_1,\ (P,\ \emptyset)).$$

Having looked at the basic case, let us consider the general case, which is depicted in figure 7 (b). In contrast to the basic case, here Y includes, but is not limited to, $\{l_i^k, \ldots, l_{m_k}^k\}$, and x is specified with $x \leftarrow y_k.P$ and $y_k \leftarrow y_{k-1}.Q'_k$, where y_{k-1} is not the root. Thus instead of the absolute key ϕ_1, we need a relative key in Θ:

$$\phi_1 = (Q,\ (Q'_k,\ \{P_1^k, \ldots, P_{m_k}^k\})),$$

where Q is the PL expression $P(x_r, y_{k-1})$, i.e., from the root to the parent of y_k. That is, $P_1^k, \ldots P_{m_k}^k$ are a key for nodes in $y_{k-1}[\![Q'_k]\!]$ relative to the parent y_{k-1} of y_k. As observed earlier, a relative key alone cannot identify a node in an XML document. To do so, we need a *transitive* set of relative keys. Thus to identify y_k we need to generate a key for y_{k-1}, and so on. We increment Θ by generating these keys until either we obtain a transitive set of relative keys up to the root of T_R, or we reach some y_i specified with $l' : id(y_i)$ such that l' is in Y. In the later case we no longer need a key for y_i as its id is unique in the entire document. Thus given its id as the value of a field in Y, we can consequently identify $y_i, y_{i+1}, \ldots, y_k$ using the relative keys we have generated so far, by the variable scoping rules and the semantics of keys. We use this process to generate keys of Θ.

The notions of equality in the XML context, i.e. node identity and value equality, complicate FD derivation and needs special treatments. For this purpose, we associate a distinguished attribute $\#id$ with each element in an XML tree, and enforce it by expanding Σ with:

$$(\_*,\ \{\#id\}), \qquad (\_*,\ (\#id,\ \emptyset)).$$

The first key asserts that $\#id$ models the id value of each XML node, and the second asserts that each node has at most one $\#id$ attribute. As an immediate result, if x is a descendant of y and $P(y, x)$ does not contain "$\_*$", then $(P(x_r, y), \{P(y, x).\#id\})$ follows from the expanded Σ, which can be proved using the inference rules of [9]. The intuition behind this is that each node in an XML tree has a unique ancestor at a certain level above it.

It is worth mentioning that we need to expand Σ to cope with the semantics of the transformation algorithm. This concerns a specific rule in the inference system for XML keys, the subnode rule of [9]: if $(Q, (Q'.Q'', \{P\}))$ then $(Q, (Q', \{Q''.P\}))$. In general, this rule requires that the set of key paths is a singleton, as it may not be sound if multiple key paths do not emit from the same node in $[\![Q.Q'.Q'']\!]$. In the context of transformations, however, this will not happen since according to the definition of well-formed transformations, from each node $n \in [\![Q.Q']\!]$, if $n[\![Q''.P_1]\!], \ldots, n[\![Q''.P_k]\!]$

are used to populate attributes in R then $P_1, \ldots, P_k$ are children of a single node $n' \in n[\![Q'']\!]$. Formally, one can show that for any transformation σ and any XML tree T satisfying a key $\varphi = (Q, (Q'.Q'', \{P_1, \ldots, P_k\}))$ in Σ, there is another XML tree T' such that T' satisfies both φ and $\varphi_1 = (Q, (Q', \{Q''.P_1, \ldots, Q''.P_k\}))$ and moreover, $\sigma(T) = \sigma(T')$. To capture these, we expand Σ with φ_1. This expansion does not leads to derivation of FDs that are not implied by Σ. To see this, let Σ' be Σ plus φ_1, and ϕ be a FD over a relation R. Obviously, for any transformation σ, if $\Sigma \models_\sigma R : \phi$, then $\Sigma' \models_\sigma R : \phi$. Conversely, if $\Sigma \not\models_\sigma R : \phi$, i.e., there is an XML tree T such that $T \models \Sigma$ but $\sigma(T) \not\models \phi$, then from the discussion above follows that there exists another XML tree T' such that $T' \models \Sigma'$ and $\sigma(T) = \sigma(T')$, and thus $\Sigma' \not\models_\sigma R : \phi$. Hence this expansion preserves the FD derivable from the original key set.

Putting everything together, we develop the algorithm `propagation` in figure 8. Given as input a set Σ of XML keys, a FD $\phi = Y \to l$ over R and a table rule `Rule`(R), it returns the value `true` if and only if $\Sigma \models_\sigma R : \phi$. The algorithm first expands Σ as illustrated above, and then uses a recursive function `keys` to compute Θ. More specifically, given a set of XML keys Θ (initially $\emptyset$), a variable x in the table rule, the set Y of fields and the table rule `Rule`(R), the function `keys` increments Θ and returns the new value of the set. Finally, the algorithm calls the implication checking algorithm developed in [9] to determine whether $\Sigma \models \Theta$, which holds if and only if $\Sigma \models_\sigma R : \phi$.

As an example, consider a transformation σ specified in section 3, which defines `Rule`(chapter), represented by the table tree depicted in figure 4. Let Σ be the set of XML keys consisting of KS_1, KS_2, KS_3 given in section 2, and ϕ be a FD over `chapter`: $(inBook, number) \to name$. Given Σ, ϕ and `Rule`(chapter), the algorithm `propagation` first computes Θ, which consists of:

$$(\_*.book.chapter,\ (number,\ \emptyset)), \qquad (\_*.book,\ (chapter,\ \{number\})),$$
$$(\_*.book,\ \{isbn,\ chapter.number\})).$$

It then uses the implication checking algorithm of [9] to determine whether $\Sigma' \models \Theta$, where $\Sigma' = \Sigma \cup \{(\_*, \{\#id\}), (\_*, (\#id, \{\}))\}$. It concludes that $\Sigma' \models_\sigma$ `chapter` : ϕ and returns `true` since $\Sigma' \models \Theta$. Indeed, ϕ is a FD derived from Σ as for any XML tree T, if $T \models \Sigma$ then $\sigma(T) \models \phi$.

For the complexity of the algorithm, a careful analysis reveals that the set Θ can be computed in $O(n^2)$ time, where n is the size of the table tree T_R, which is in turn linear in the size of `Rule`(R) (and is larger than the size of the FD ϕ). This can be achieved with a proper preprocessing of the table tree T_R (in $O(n)$ time). With this the function `keys` takes no more time than traversing T_R starting from each node encountered. The size of Θ is thus at most $O(n^2)$, and the size of the expanded Σ is $m + 2$, where m is the size of the input Σ. Thus the implication checking algorithm of [9] takes $O(p(n^2 + m))$ time for a polynomial p. Hence algorithm `propagation` is in PTIME.

Algorithm `propagation`

Input: XML keys Σ, FD $\phi = Y \rightarrow l$ over R, and $\texttt{Rule}(R)$ in a transformation σ, in which $l : f(x)$.
Output: `true` iff $\Sigma \models_\sigma R : \phi$.

1. if $l \in Y$
   then return `true`; /* trivial FDs */
2. for each $(Q, (Q'.Q'', \{P_1, \ldots, P_k\})) \in \Sigma$ do
   $\Sigma := \Sigma \cup \{(Q, (Q', \{Q''.P_1, \ldots, Q''.P_k\}))\}$; /* captures subnodes rule applied to $k > 1$ */
3. $\Sigma := \Sigma \ \cup \ \{(\_*, \ \{\#id\}), \ \ (\_*, \ (\#id, \ \emptyset))\}$; /* captures the semantics of id */
4. $\Theta :=$ $\texttt{keys}(\emptyset, x, Y, \texttt{Rule}(\mathrm{R}))$; /* generates Θ by using the function `keys` */
5. if $\Sigma \models \Theta$ /* uses the implication algorithm of [9] */
   then returns `true`
   else returns `false`;

function `keys` $(\Theta, y, Y, \texttt{Rule}(\mathrm{R}))$

Input: Θ: a set of XML keys, y: a variable in $\texttt{Rule}(R)$, Y: the set of fields in $\phi = Y \rightarrow l$, and $\texttt{Rule}(R)$.
Output: expanded Θ.

1. if (there is $l' \in Y$ such that $l' : id(y)$ is in $\texttt{Rule}(\mathrm{R})$) or ($y$ is x_r, i.e., the root)
   then return Θ; /* Θ remains unchanged in this case */
2. $w := parent(y)$;
3. $S := \emptyset$;
4. for each $z \in descendants(y)$ such that there is $l' \in Y$ and $l' : f(z)$ is in $\texttt{Rule}(\mathrm{R})$ do
   if f is id /* distinguishing value equality from node identity */
   then $S := S \ \cup \ \{P(y, z).\#id\}$; /* collecting the key paths of a new key */
   else $S := S \ \cup \ \{P(y, z)\}$;
5. $\Theta := \Theta \ \cup \ \{(P(x_r, w), \ (P(w, y), \ S))\}$; /* expanding Θ */
6. return $\texttt{keys}(\Theta, \ w, \ Y, \ \texttt{Rule}(R))$; /* recursively processing the parent w of y */

Figure 8: An algorithm for determining XML key propagation

Next, we give a formal proof of Theorem 4.1.

Proof: It suffices to show that the algorithm `propagation` returns `true` if and only if $\Sigma \models_\sigma R : \phi$. That is, $\Sigma' \models \Theta$ if and only if $\Sigma \models_\sigma R : \phi$, where Σ' is the expansion of Σ and Θ is the set of XML keys computed by the algorithm.

Before we give the proof, we need to introduce some notations. First, assume FD $\phi = Y \rightarrow l$, where l is specified with $l : f(x)$, and each $l_a \in Y$ is specified with $l_a : f_a(a)$. Let Y-var be the set of variables z such that $l : f(z)$ is in $\texttt{Rule}(\mathrm{R})$ and $l \in Y$. Then, given a variable y, we define $Ydesc(y)$ to be the set of variables z such that $z \in descendents(y)$ and $z \in Y$-var. Second, recall that in a table tree T_R, each variable z corresponds to a unique node in T_R, called the z-node, and if z_1 is an ancestor of z_2 then there exists a unique path from the z_1-node to the z_2-node denoted by $P(z_1, z_2)$. We denote by z-path the path from the root to z-node, that is, $P(x_r, z)$; also, if for some $l_z \in Y$, l_z is specified with $l_z : f_z(z)$, then we call l_z the z-label. The next notation concerns when two nodes in an XML tree *agree on* Y. Let y be a variable used in a transformation, T be an XML tree and u, u' be two nodes in $[\![P(x_r, y)]\!]$ in T. Assume that $ancestors(y) = \{y_1, \ldots, y_k\}$, where $k \geq 0$ and for all $i \in [2, k]$, $y_{i-1} = parent(y_i)$. Let $v_1, \ldots, v_k, v_{k+1}$ and $v'_1, \ldots, v'_k, v'_{k+1}$ be nodes in T along the path from the root to u and u', respectively. That is, $v_1 = v'_1 = r$, where r is the root node, $v_{k+1} = u$, $v'_{k+1} = u'$, and for $i \in [2, k+1]$, $v_i \in v_{i-1}[\![P(y_{i-1}, y_i)]\!]$, and $v'_i \in v'_{i-1}[\![P(y_{i-1}, y_i)]\!]$. We say that u and u' *agree on* Y if for every i in $[1, k+1]$ and every z in $Ydesc(y_i)$, there exists $e \in v_i[\![P(y_i, z)]\!]$ such that $e' \in v'_i[\![P(y_i, z)]\!]$, and $e =_v e'$ if $f_z = value$ or $e = e'$ if $f_z = id$.

Given these notations, we are ready to show that if $\Sigma' \models \Theta$ then $\Sigma \models_\sigma R : \phi$. It suffices to show that given $\Sigma' \models \Theta$, for any tree T, if $T \models \Sigma$ then there are no two distinct nodes u, u' in $[\![P(x_r, x)]\!]$ that agree on Y. For if it holds then we can prove $\Sigma \models_\sigma R : \phi$ as follows. Suppose by contradiction $\Sigma \not\models_\sigma R : \phi$. That is, there exists an XML tree T such that $T \models \Sigma$ but $\sigma(T) \not\models \phi$, i.e., there are two tuples $t_1, t_2 \in \sigma(T)$ such that $t_1.l' = t_2.l'$ for all $l' \in Y$ but $t_1.l \neq t_2.l$. Let $ancestors(x) = \{y_1, \ldots, y_k\}$. By the semantics of the transformation σ, if a tuple t is in $\sigma(T)$ then there are nodes $v_1, \ldots, v_k, v_{k+1}$ on a path $P(x_r, x)$ in T such that y_i is mapped to v_i, v_1 is the root r, $v_{k+1} \in [\![P(x_r, x)]\!]$, and $v_i \in v_{i-1}[\![P(y_{i-1}, y_i)]\!]$ for $i \in [2, k+1]$. Moreover, $t.l$ is determined by the value or node identity of v_{k+1} by the definition of f, and $t.l'$ is determined by the value or node identity of a node $e \in v_i[\![P(y_i, z)]\!]$ for some $z \in Ydesc(y_i)$. Thus one can easily verify that $\Sigma \not\models_\sigma R : \phi$ implies the existence of two nodes $u, u' \in [\![P(x_r, x)]\!]$ agreeing on Y such that either $u \neq_v u'$ or $u \neq u'$. As node equality implies value equality, in either case we have $u \neq u'$. That is, there are two distinct nodes agreeing on Y, which contradicts the claim. Thus $\Sigma \models_\sigma R : \phi$.

We now show the claim, i.e., if $\Sigma' \models \Theta$ then for any XML tree T satisfying Σ, there exist no two distinct nodes u, u' in $[\![P(x_r, x)]\!]$ agreeing on Y. We prove the claim by induction on $|ancestor(x)|$, the cardinality of $ancestor(x)$. The base case, $|ancestor(x)| = 0$, is immediate because x must be the root. For the inductive step, assume that $|ancestor(x)| \geq 1$, and the statement holds for any ancestor of x. Suppose, by contradiction, that there exists an XML tree T such that $T \models \Sigma$ and there are two distinct nodes in $[\![P(x_r, x)]\!]$ agreeing on Y. By the definition of Σ' and the discussion given earlier, it can be shown that there is an XML tree T' such that $T' \models \Sigma'$ and in T' there exist two distinct nodes $u, u' \in [\![P(x_r, x)]\!]$ agreeing on Y. Assume that in $\texttt{Rule}(\mathrm{R})$, x is specified with $x \leftarrow x'.\rho$. Let w, w' be nodes in $[\![(P(x_r, x')]\!]$ such that $u \in w[\![\rho]\!]$ and $u' \in w'[\![\rho]\!]$.

Since x' has one less ancestor than x, by the induction hypothesis, $w = w'$. That is, u, u' are both descendants of the same node w. By the function `keys` given in the algorithm, Θ contains a key $(P(x_r, x'), (P(x', x), S))$, where S consists of $P(x, z)$ or $P(x, z).\#id$ for all z in $Ydesc(x)$. By the assumption, $\Sigma' \models \Theta$; hence $T' \models \Theta$. Therefore, since u and u' agree on the key paths in S, they cannot be distinct nodes in T'. This leads to a contradiction. Thus the claim holds in all cases.

For the other direction, we show that if $\Sigma \models_\sigma R : \phi$, then $\Sigma' \models \Theta$. It suffices to show that if $\Sigma' \not\models \Theta$ then $\Sigma \not\models_\sigma R : \phi$. More specifically, given $\Sigma' \not\models \Theta$, there exists an XML tree T such that $T \models \Sigma'$ but $T \not\models \Theta$. We show $\sigma(T) \not\models \phi$. Note that Σ' subsumes Σ and thus for any XML tree T, if $T \models \Sigma'$ then $T \models \Sigma$.

Recall that $\phi = Y \to l$, where l is specified with $l : f(x)$. We assume $x \neq x_r$, since otherwise $\Theta = \{\}$, $\Sigma' \models \Theta$, and $\Sigma \models_\sigma R : \phi$ given that in any XML tree there exists a unique root node. We also assume that $l \notin Y$, since otherwise ϕ is a trivial FD and the propagation algorithm returns `true`. Let $ancestors(x) = \{y_1, \ldots, y_k\}$, where $k \geq 0$ and for any $i \in [2, k]$, $y_{i-1} = parent(y_i)$. Observe that the propagation algorithm computes $\Theta = \{\psi_1, \ldots, \psi_k\}$, where for each $i \in [1, k]$, $\psi_i = (Q_i, (Q'_i, S_i))$, $Q_i = P(x_r, y_i)$, $Q'_i = P(y_i, y_{i+1})$, $y_{k+1} = x$, and $S_i = \{P(x_{i+1}, z) \mid z \in Ydesc(y_{i+1})\}$.

Since $\Sigma' \not\models \Theta$, there exists at least one key $\psi_i \in \Theta$ such that $\Sigma' \not\models \psi_i$ for some $i \in [1, k]$. Let ψ_i be such a key with the shortest context path. That is, for all $j < i$, $\Sigma' \models \psi_j$, but $\Sigma' \not\models \psi_i$. We consider the following two cases.

(1) $i = 1$, that is, ψ_i is an absolute key (Q'_1, S_1), where $Q'_1 = P(y_1, y_2)$ and $y_1 = x_r$. We can partition Y into two sets: $Y_1 = \{l_i \mid i \in [1, m]\}$ such that $S_1 = \{P(y_2, z_i) \mid i \in [1, m],\ l_i \text{ is the } z_i\text{-label}\}$ and $Y_2 = Y \setminus Y_1 = \{y_j \mid j \in [m+1, n]\}$. Intuitively, if we use Y_{var1} and Y_{var2} to denote the variables specifying the Y_1-labels and Y_2-labels, rspectively, then variables in Y_{var1} are descendants of y_2, whereas variables in Y_{var2} are not below y_2, i.e., not in $Ydesc(y_2)$. Since $\Sigma' \not\models \psi_1$, there must exist an XML tree T such that $T \models \Sigma'$, but $T \not\models \psi_1$. The structure of T is depicted in Figure 9(a). By $T \not\models \psi_1$, there exists in T at least two distinct nodes v_1, v'_1 in $[\![P(x_r, y_2)]\!]$ that agree on Y. Thus for each $i \in [1, m]$, there are v_{x_i}, v'_{x_i} in $v_1[\![P(y_2, z_i)]\!]$ and $v'_1[\![P(y_2, z_i)]\!]$ respectively such that $v_{x_i} =_v v'_{x_i}$, where l_i is the z_i-label. In addition, assume that there are v_x, v'_x in $v_1[\![P(y_2, x)]\!]$ and $v'_1[\![P(y_2, x)]\!]$, respectively, such that $v_x \neq_v v'_x$. This assumption does not lose generality as one can always modify T to include such nodes without affecting the satisfaction of Σ' and Θ (cf. [9]).

Given these, we verify $\Sigma \not\models_\sigma R : \phi$ by showing that $\sigma(T)$ contains two tuples t_1, t_2 such that $t_1.l' = t_2.l'$ for all $l' \in Y$ but $t_1.l \neq t_2.l$. Indeed, by the semantics of the transformation σ, there are t_1, t_2 in $\sigma(T)$ such that: (a) $t_1.l$ and $t_2.l$ are the values or ids of v_x and v'_x, respectively, depending on the definition of l; thus in either case $t_1.l \neq t_2.l$; (b) for each $l' \in Y_2$, $t_1.l'$ and $t_2.l'$ are the value

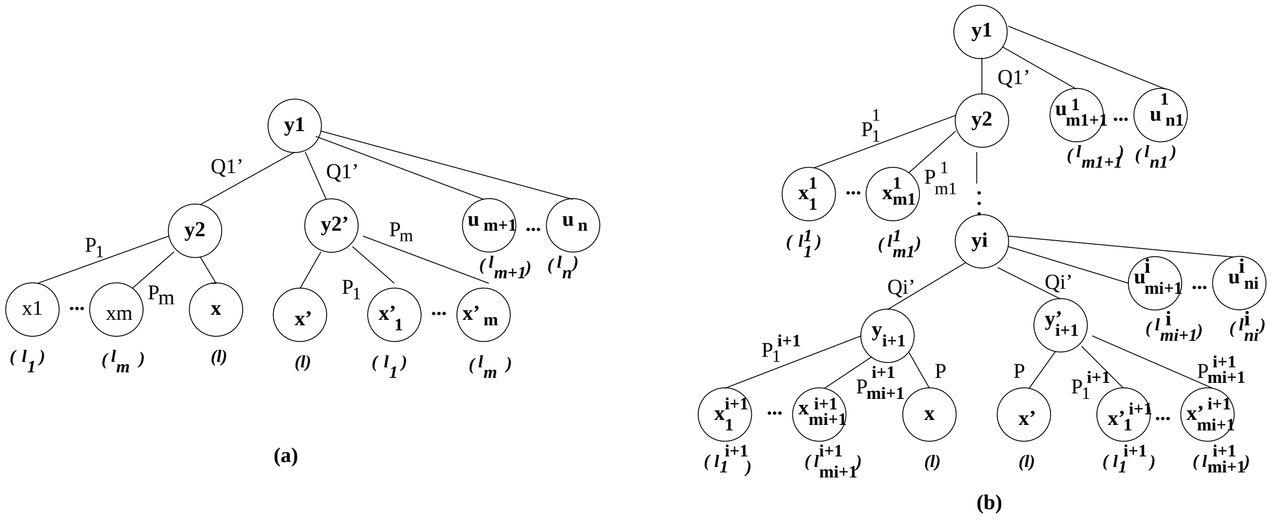

Figure 9: Trees used in the proof of Theorem 4.1

or id of the same node reached by following $P(y_1, z)$, where l' is the z-label; thus $t_1.l' = t_2.l'$; (c) for each $i \in [1, m]$ (i.e., $l_i \in Y_1$), $t_1.l_i$ and $t_2.l_i$ are the values of v_{x_i} and v'_{x_i}, respectively. Observe that l_i cannot be defined with id. Indeed, observe that Σ' includes $(\_*, \{\#id\})$ and $(\_*, (\#id, \emptyset))$. Thus if l_i is defined with id we would have had $\Sigma' \models \varphi_1$ by using the inference rules of [9], which contradicts the assumption. Hence again $t_1.l_i = t_2.l_i$. Finally, to see that these two tuples are well-defined, it suffices to show that for any $i, j \in [1, m]$, if $P(y_2, z_i)$ and $P(y_2, z_j)$ have a common prefix ρ, then v_{x_i} and v'_{x_i} emit from the same node e in $y_2[\![\rho]\!]$. If it is not the case one can slightly modify T to construct another T' that satisfies the condition. This does not lose generality. Indeed, observe that Σ' also includes $(Q, (Q', \{Q''.P_1, \ldots, Q''.P_k\}))$ as long as Σ contains $(Q, (Q'.Q'', \{P_1, \ldots, P_k\}))$. Given this, if the modification affected the satisfaction of Σ' and Θ we would again have had $\Sigma' \models \varphi_1$ by using the inference rules of [9], which contradicts the assumption. Thus t_1 and t_2 are indeed two tuples in $\sigma(T)$ that violate the FD ϕ. Therefore, $\Sigma \not\models_\sigma R : \phi$.

(2) $i > 1$, that is, ψ_i is a relative key. Let T be an XML tree such that $T \models \Sigma'$, but $T \not\models \psi_i$. By assumption, for all $j < i$, $\Sigma' \models \psi_j$, therefore for each ancestor y_j of x, there exist no two distinct nodes in $[\![P(x_r, y_j)]\!]$ that agree on Y. But since $T \not\models \psi_i$, there are at least two distinct nodes v_x, v'_x in $v_i[\![Q'_i]\!]$ agreeing on Y. The structure of T is depicted in Figure 9(b). Similarly to case (a), we define Y_{var2} to be the set of variables in Y-var that are not descendants of y_{i+1}. Then $\sigma(T)$ contains two tuples that agree on the attributes in Y, but disagree on l, where attributes in labels defined by variables of Y_2 are populated with values of the same nodes, and the remaining attributes in Y are populated from different nodes that agree on their values. Since v_x and v'_x are distinct nodes, we have $\Sigma \not\models_\sigma R : \phi$. $\square$

Algorithm `derivation`(Σ, `Rule`(R))

$\Gamma := \emptyset$;
for all $Y \subseteq att(R)$ do
  for all $l \in att(R)$ and $l \notin Y$ do
    if `propagation` ($\Sigma, Y \to l,$ `Rule`(R))
    then $\Gamma := \Gamma \cup \{Y \to l\}$;
return Γ;

Figure 10: An algorithm for deriving FDs

4.1 Derivation of functional dependencies

A more general problem is that of deriving a complete set of functional dependencies from a set of XML keys.

Roughly speaking, given a mapping σ from XML data to databases $\mathbf{I(R)}$ of a schema $\mathbf{R}$ and a set Σ of XML keys of $\mathcal{K}$, we want to find the set Γ of FDs such that $\Gamma = \{\phi \mid \Sigma \models_\sigma R : \phi,\ R \text{ in } \mathbf{R}\}$. A simple algorithm for computing the set of all non-trivial FDs propagated from a set Σ under a mapping σ is given in Figure 10.

In practice, however, it is impractical to compute Γ since Γ can be exponentially large in the size of $\mathbf{R}$ (cf. [1]) as it contains "redundant" FDs such as those of the form $X \to Y$ and $X' \to Y$ with $X' \subseteq X$. These redundant FDs can be derived from others using the simple inference rules for FDs [1]. But even with this restriction, an algorithm for computing a complete set of FDs over R has an EXPTIME complexity. There are practical restrictions to better complexity (PTIME), however. We defer to another publication a full treatment of these cases as they are a little involved.

5 Conclusion

We have developed a specification language for transforming XML documents to relational databases without requiring a DTD or schema on the XML data, and provide a PTIME algorithm for determining whether a functional dependency is propagated from XML keys through our transformations. These results can be naturally generalized to derive a complete set of functional dependencies from XML keys. We also show that the constraint propagation problem is undecidable for some extensions of our language.

These results are useful in refining the relational design and in optimizing queries, among other things. In addition, our algorithm for key propagation can be generalized and incorporated into relational storage techniques published in the literature (e.g. [30, 31, 28]). These results are also useful in understanding XML to XML transformations. In particular, the key propagation problem is undecidable when XQuery [11] is used to specify transformations since it subsumes the undecidable extension of our specification language; and a fragment of XQuery [11] and other popular query languages possesses a PTIME algorithm for XML key propagation. This work parallels that of [2], which investigates propagation of type constraints through queries.

One topic for future work is to extend our transformation language by incorporating predicates so that one can select parts of XML data that satisfy certain conditions. This will not make the key propagation problem undecidable, and we expect that our algorithm for XML key propagation can be generalized to accommodate the extension. Another topic is to study constraint propagation in the presence of DTDs and XML Schema. However, as indicated in [19], the interaction between DTDs and constraints may make the analysis of XML key propagation more intriguing. Finally, it is important to study propagation of other types of integrity constraints for XML, e.g., foreign keys and inverse constraints, to the relational format as well as to object-oriented constraints.

References

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

[2] N. Alon, T. Milo, F. Neven, and D. S. V. Vianu. XML with data values: Typechecking revisited. In *PODS*, 2001.

[3] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL. *Nucleic Acids Research*, 28:45–48, 2000.

[4] C. Beeri and M. Vardi. A proof procedure for data dependencies. *Journal of ACM*, 31(4):718–741, 1984.

[5] D. Benson et al. GenBank. *Nucleic Acids Research*, 28:15–18, 2000.

[6] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML schema to relations: A cost-based approach to XML storage. In *ICDE*, 2002.

[7] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation, Feb. 1998. `http://www.w3.org/TR/REC-xml/`.

[8] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *WWW'10*, 2001.

[9] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about keys for XML. In *Proc. of Int'l Workshop on Database Programming Languages (DBPL)*, 2001.

[10] P. Buneman, W. Fan, J. Siméon, and S. Weinstein. Constraints for semistructured data and XML. *SIGMOD Record*, 30(1), 2001.

[11] D. Chamberlin et al. XQuery 1.0: An XML Query Language. W3C Working Draft, June 2001. `http://www.w3.org/TR/xquery`.

[12] J. Clark and S. DeRose. XML Path Language (XPath). W3C Working Draft, Nov. 1999. `http://www.w3.org/TR/xpath`.

[13] S. Davidson and A. Kosky. WOL: A language for database transformations and constraints. In *Proc. of Int'l Conf. on Data Engineering (ICDE)*, 1997.

[14] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *SIGMOD'99*, pages 431–442, 1999.

[15] A. Deutsch, L. Popa, and V. Tannen. Physical data independence, constraints and optimization with universal plans. In *Proc. of Int'l Conf. on Very Large Databases (VLDB)*, 1999.

[16] A. Deutsch and V. Tannen. Optimization properties for classes of conjunctive regular path queries. In *Proc. of Int'l Workshop on Database Programming Languages (DBPL)*, 2001.

[17] L. Dodds. XML and Databases? Follow Your Nose, October 2001. `http://www.xml.com/lpt/a/2001/10/24/follow-yr-nose.html`.

[18] EMBL-EBI (European Bioinformatics Institute) and Rosetta Inpharmatics. Gene Expression RFP Response, August 2001. Available at `www.geml.org/omg.htm`.

[19] W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. In *PODS'01*, 2001.

[20] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addision Wesley, 1979.

[21] A. Klug. Calculating constraints on relational expressions. *ACM Trans. on Database Systems*, 5(3):260–290, 1980.

[22] A. Klug and R. Price. Determing view dependencies using tableaux. *ACM Trans. on Database Systems*, 7(3):361–380, 1982.

[23] A. Layman et al. XML-Data. W3C Note, Jan. 1998. `http://www.w3.org/TR/1998/NOTE-XML-data`.

[24] D. Lee and W. W. Chu. Constraints-preseving transformation from XML document type definition to relational schema. In *Proc. of Int'l Conf. on Conceptual Modeling (ER)*, 2000.

[25] D. Maier, A. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *ACM Trans. on Database Systems*, 4(4):455–469, 1979.

[26] I. Manolescu, D. Florescu, and D. Kossmann. Pushing XML queries inside relational databases. Tech. Report no. 4112, INRIA, 2001.

[27] I. Manolescu, D. Florescu, D. Kossmann, F. Xhumari, and D. Olteanu. Agora: Living with XML and relational. *VLDB Journal*, pages 623–626, 2000.

[28] Oracle Corporation. *Oracle9i Application Developer's Guide - XML, Release 1 (9.0.1)*, 2001. http://download-west.oracle.com/otndoc/oracle9i/901_doc/appdev.901/a88894/toc.htm.

[29] A. Schmidt, M. L. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *WebDB (Informal Proceedings)*, pages 47–52, 2000.

[30] J. Shanmugasundaram et al. Relational databases for querying XML documents: Limitations and opportunities. *VLDB Journal*, pages 302–314, 1999.

[31] J. Shanmugasundaram et al. A general techniques for querying XML documents using a relational database system. *SIGMOD Record*, 30(3):20–26, 2001.

[32] H. Thompson et al. XML Schema. W3C Working Draft, May 2001. `http://www.w3.org/XML/Schema`.

[33] M. Y. Vardi. The decision problem for database dependencies. *Inf. Proc. Letters*, 12(5):251–254, 1981.