Characterization of XML Functional Dependencies and their Interaction with DTDs

Łucja Kot and Walker White

Department of Computer Science
Cornell University
{lucja,wmwhite}@cs.cornell.edu

Abstract. With the rise of XML as a standard model of data exchange, XML functional dependencies (XFDs) have become important to areas such as key analysis, document normalization, and data integrity. XFDs are more complicated than relational functional dependencies because the set of XFDs satisfied by an XML document depends not only on the document values, but also the tree structure and corresponding DTD. In particular, constraints imposed by DTDs may alter the implications from a base set of XFDs, and may even be inconsistent with a set of XFDs. In this paper we examine the interaction between XFDs and DTDs. We present a sound and complete axiomatization for XFDs, both alone and in the presence of certain classes of DTDs. We show that these DTD classes form an axiomatic hierarchy, with the axioms at each level a proper superset of the previous. Furthermore, we show that consistency checking with respect to a set of XFDs is feasible for these same classes.

1 Introduction

Functional dependencies have proved to be a very useful class of integrity constraints for traditional database systems. They are integral to key analysis, normalization, and query optimization [1]. As XML is increasingly becoming the standard model of data exchange, there is much interest in formulating a definition of XML functional dependency. In addition to the benefits found in relational databases, a proper XFD definition would also aid in many new areas, such as verifying data consistency, preserving semantics during data exchange, and XML-SQL translation [10].

Several different XFD definitions have been suggested [2, 20, 14, 19, 15, 13]. The major XFD definitions are similar to the relational definition except that, instead of attributes and table rows, they use path identifiers and subtrees. Informally, these definitions say that an XFD $A \to B$ is satisfied in a document if, for any two subtrees, whenever they agree on the paths in A, they also agree on the paths in B. The definitions differ primarily in how they choose subtrees, specify path identifiers, or test equality between XML nodes.

XFDs differ from their relational counterparts in that they must take into account the tree structure of XML documents. For example, a language for defining XFDs must allow us to specify when one path is a prefix of another. Another issue is the definition of equality: nodes can be compared by identity or value equality. In the relational model, no duplicate tuples are allowed, and so value equality is sufficient in FDs. However, in an XML document, we can have two different subtrees that are isomorphic and have exactly the same values. This is a clear instance of data duplication, but one allowed by the data model. Therefore, if XFDs are to be used for keys or normalization, they must be able to detect identity (in)equality between nodes. Finally, as XML represents semi-structured data that is often incomplete, XFDs must properly handle null values.

The implication problem is fundamental to all of the applications above [1]; hence this has been the focus of much of the work on XFDs. There are two important approaches to the implication problem. One approach is that of efficient decision algorithms, which allow us to determine whether an XFD is implied by a set of XFDs; some feasible decision algorithms have been discovered already [2]. The other approach is axiomatization, which often gives us slower decision algorithms, but which is important for understanding the underlying theory of XFDs [1]. For example, every child XML node has a unique parent node. Thus for any two path identifiers q and p, where p is an identifier for the parent of q, every XML document satisfies the XFD $q \rightarrow p$. A decision algorithm would allow us to check for *each specific* instance of parent-child p, q that $q \rightarrow p$ holds. However, an axiomatization would allow us to prove this entire general class of XFDs.

The implication problem becomes more complicated in the presence of a DTD. Consider the XML document illustrated in Figure 1. This document represents admissions at a special charity hospital and has the following DTD.

```
<!DOCTYPE admissions [
    <!ELEMENT admissions (patient*)>
    <!ELEMENT patient    (name,DOB,insurance?,doctor,doctor)>
    <!ELEMENT insurance  (company,policy)>
    <!ELEMENT name       (#CDATA)>
    <!ELEMENT DOB        (#CDATA)>
    <!ELEMENT company    (#CDATA)>
    <!ELEMENT policy     (#CDATA)>
    <!ELEMENT doctor     (#CDATA)>
    <!ATTLIST doctor license CDATA #REQUIRED>
]>
```

In particular, each patient must have a recommendation from exactly two doctors (who are unordered), and may or may not have insurance. Note that in this DTD, every patient has exactly one `name` node. So every document conforming to this DTD must satisfy the XFD $p \rightarrow q$, where p is a path identifier for a patient node, and q is a path identifier for the name node of that patient. This suggests that we should be able to use the structure of a DTD to make deductions about classes of XFDs satisfied by conforming documents. Again, while there are several decision algorithms for XFDs conforming to certain classes of DTDs, to our knowledge there is no existing sound and complete axiomatization for XFDs with identity equality in the presence of a DTD.

As we show in this paper, in order to axiomatize XFDs, we must also address the problem of consistency. In the example in Figure 1, every patient must have two recommending doctors. We give no preference to either doctor, and we may want to assert an XFD from the `license` number of a `doctor` to its text content, so we do not wish to give the doctors different tags. However, consider the XFD $p \rightarrow q$, where p is a path identifier for a patient, and q is a path identifier for a recommending doctor; this XFD

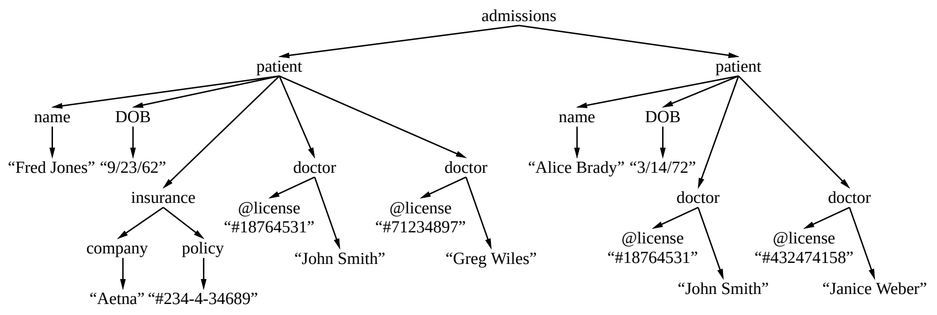


Fig. 1. XML Document for Medical Admissions

is inconsistent with our DTD. While consistency has been studied for key constraints [11], these types of constraints are properly weaker than XFDs and are not sufficient for many of the applications mentioned above. To the best of our knowledge, the problem of the consistency of XFDs with a DTD has not been studied before.

1.1 Contributions

This paper is a thorough study of the theory of XFDs and their interaction with various classes of DTDs, focusing on the implication and consistency problems. In this paper, we make the following contributions.

- We adapt the definition of XFDs presented in [2] to include documents without a DTD so that we can identify the base theory of XFDs.
- We make explicit the natural mapping between XFDs and relational FDs implicit in [2], and use it to leverage existing relational FD theory to reason about XFDs.
- We adapt the chase algorithm to XFDs, and use it to improve existing bounds on the implication problem for XFDs, in some cases to linear time.
- We use the chase to formulate the first sound and complete axiomatization for XFD implication (using identity equality), in the absence of a DTD.
- We show that derivations produced from these axioms can be encoded as logical inferences with Horn clauses.
- We expand these techniques to explore the interactions between XFDs and several classes of DTDs (refer to the matrix at the end of the paper for a summary).
  - We present both the chase and a consistency checking algorithm (where applicable) for each of the various classes of DTDs, showing that deciding implication and consistency is feasible for these classes.
  - We present a sound and complete axiomatization for each class of DTDs. We show that these classes form a natural hierarchy, each class adding a new set of axioms to those of the previous class.
  - We show that our Horn clause encoding of the derivations generalizes to these axioms.
- We present intractability results for the remaining classes of DTDs, giving a complete characterization of where consistency and implication are feasible.

2 Preliminaries

2.1 The document model

Throughout this paper, our notation is similar to that in the literature [2], though with some noticeable differences. These differences are necessary because this existing notation requires that an XML document have a corresponding DTD. In order to study the theory of XFDs, we need to decouple the definition of an XFD from a DTD.

Our model for an XML document is a tree representing the underlying DOM structure. We associate each part of the document, including text and attribute data, with a labeled node in the model. For these labels, we have two disjoint sets $\textsc{El}$ and $\textsc{Val}$. The set $\textsc{El}$ is the label alphabet for XML nodes and attribute names, while $\textsc{Val}$ is the alphabet of attribute values. The two sets $\textsc{El}$ and $\textsc{Val}$ are countably infinite, as we are not restricting ourself to a single DTD. We define a special null symbol null $\notin \textsc{El} \cup \textsc{Val}$.Furthermore, so that we can encode document ordering of the nodes, we assume that $\mathbb{N} \cap \textsc{El} = \mathbb{N} \cap \textsc{Val} = \text{null}$.

Our model is the same in as Arenas and Libkin [2] with only two minor modifications. First, our alphabet $\textsc{El}$ contains two distinguished elements ρ and α, for identifying special nodes in the tree. The label ρ is used to identify the unique root element of each XML tree. This corresponds to the `<?xml>` tag in an XML document and is necessary because documents without a DTD are not constrainted with respect to root label. The label α, on the other hand, is intended for attribute *values* and character data; the second set $\textsc{Val}$ represents these data values.

Formally, we model an XML document T as the tuple $(V, v_{\text{root}}, \text{child}, \text{lab}, \text{attval}, \text{num})$. The initial triple $(V, v_{\text{root}}, \text{child})$ represents an unranked (i.e. not n-ary for any finite n) finite tree with no sibling ordering on the nodes. The set V is the set of nodes in T, which we often refer to as nodes(T), while v_{root} is the root node. The function child : $V \rightarrow 2^V$ takes a node and returns the children of that node; the definition of child is restricted so that it does encode a proper tree.

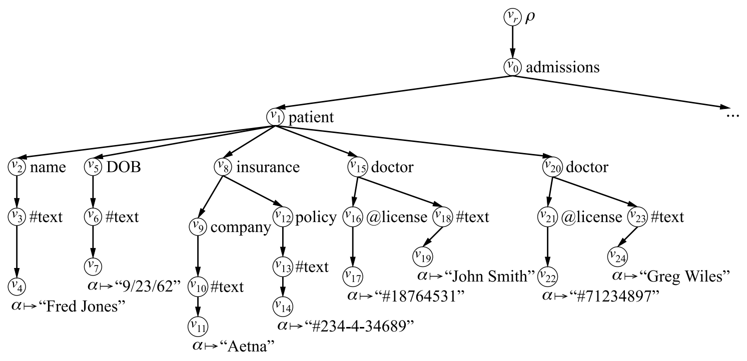
The remaining elements of the XML document are labeling functions to encode the data and document order. The function lab : $V \rightarrow \textsc{El}$ labels all of the XML attributes and entities. It is subject to the following restrictions for the special elements of $\textsc{El}$:

- lab$(v) = \rho$ if and only if v is the root of T
- lab$(v) = \alpha$ implies v is a leaf of T with no siblings in T

Those special elements labeled with α are data elements; for them we have the labeling function attval : $\{\, v \in V \mid \text{lab}(v) = \alpha \,\} \rightarrow \textsc{Val}$ which assigns attribute values to those leaf nodes of T. The function num : $V \rightarrow \mathbb{N}$ orders the nodes in T in some arbitrary order, which we can assume to be the document order.

To illustrate this model, consider the XML document from Figure 1. Our model would represented this document by the labeled tree in Figure 2.

A path identifier is a finite list of labels in $\textsc{El}$. For clarity, we separate the elements in a path identifier by periods, such as ρ.admissions.patient. We say that a path identifier p *occurs* in a tree T if there is a path $v_1, v_2, \cdots v_n$ in T such that the labels for the v_i form a string equal to p. For the rest of the paper, we will abuse terminology slightly by using the term *paths* both for path identifiers and actual paths, with the understanding



v_r ρ
v_0 admissions
v_1 patient
v_2 name
v_3 #text
v_4 $\alpha\mapsto$ "Fred Jones"
v_5 DOB
v_6 #text
v_7 $\alpha\mapsto$ "9/23/62"
v_8 insurance
v_9 company
v_{10} #text
v_{11} $\alpha\mapsto$ "Aetna"
v_{12} policy
v_{13} #text
v_{14} $\alpha\mapsto$ "#234-4-34689"
v_{15} doctor
v_{16} @license
v_{17} $\alpha\mapsto$ "#18764531"
v_{18} #text
v_{19} $\alpha\mapsto$ "John Smith"
v_{20} doctor
v_{21} @license
v_{22} $\alpha\mapsto$ "#71234897"
v_{23} #text
v_{24} $\alpha\mapsto$ "Greg Wiles"
...

Fig. 2. Encoding of Medical XML Document

that a single path identifier may represent more than one actual path in the tree. We use the letters p, q, s, t, u to denote path identifiers, and x, y, z, w for symbols in EL.

We say that a path is *rooted* if its first identifier is ρ. We denote the set of all rooted paths that occur in T as paths(T); note that this is a prefix-closed set. For any paths p and q, by $p \preceq q$ we mean that p is a prefix of q; we use $p \prec q$ we denote that p is a proper prefix of q.

Observe that for all T, paths(T) must be a subset of the set of all possible allowed rooted paths over EL. Those can be described using the regular expression

$$\text{PATHS} = \rho\ (\text{EL}\backslash\{\rho, \alpha\})^*\ \alpha?$$

Because we are often be concerned with regular expressions, given an alphabet A, we write rexp(A) to denote the set of all regular expressions over A.

As with the definition in Arenas and Libkin [2], we work with two kinds of equality on the nodes of T. We compare internal tree nodes by identity; on the other hand, we compare α leaf nodes by their value in VAL. Identity equality is denoted $=_{id}$. Given two nodes $v, w \in V$, $v =_{id} w$ iff num(v) = num(w). Similarly, value equality is denoted $=_{val}$ and defined only on attribute nodes. That is, for $v, w \in V$, v and w both leaf nodes, $v =_{val} w$ iff if lab(v) = lab$(w) = \alpha$ and attval(v) = attval(w).

Finally, we find it useful to talk about embeddings and structural embeddings between trees. A structural embedding from a tree T to a tree T' is a structure-preserving and label-preserving one-to-one map f from the nodes of T to the nodes of T'. Two trees are structurally isomorphic if they structurally embed into each other. An embedding is a structural embedding which also respects equality on attribute values; formally, we require that for every two nodes v_1, v_2 in T whose label is α, we have $f(v_1) =_{val} f(b_2)$ if and only if $v_1 =_{val} v_2$.

2.2 DTDs

Our definition of an XFD does not depend on any features of sibling order in documents. However, DTDs do impose a sibling order. Therefore, in order to study the interaction between XFDs and DTDs, we need to define an equivalence relation on DTDs which allows us to disregard sibling ordering constraints.

A DTD is defined as $D = (E, P)$, where E is a finite subset of EL and P is a mapping from the elements E to element type definitions where

$$P \subseteq E \times (\text{rexp}(E \setminus \{\alpha, \rho\})\ \cup\ \{\alpha\})$$

It is standard notation, when working with DTDs, to use arrows in the notation for productions (i.e. $a \rightarrow bc^*$). In this paper, we need to avoid a notational conflict with another sense of $\rightarrow$, which is used for XFDs. Therefore, we use $\rightsquigarrow$ instead for DTD productions.

A document T satisfies D, written $T \vDash D$, if for every node $v \in \text{nodes}(T)$ whose children are $v_1, \cdots v_n$, the string $\text{lab}(v_1)\text{lab}(v_2)\cdots\text{lab}(v_n)$ is in the language of P. We assume that all DTDs considered from now on are consistent, in that there is at least one finite tree satisfying the DTD. Furthermore, we denote by $\text{paths}(D)$ all the possible rooted paths that may occur in any T such that $T \vDash D$.

We now define the equivalence relation on DTDs, $\cong$, which we use to ignore sibling order. Let D and D' be two DTDs. Let $\mathcal{T}_D = \{\, T \mid T \vDash D \,\}$ and $\mathcal{T}'_D = \{\, T \mid T \vDash D' \,\}$. Then $D \cong D'$ if and only if the following two conditions are satisfied.

- There is a bijection $f : \mathcal{T}_D \rightarrow \mathcal{T}'_D$.
- Whenever $f(T) = T'$ for some T', then T' can be obtained from T by perfoming a finite number of sibling subtree swaps in T. That is, T' and T are structurally isomorphic.

We assume that all DTDs considered from now on are consistent (i.e. there is at least one finite tree satisfying the DTD). Furthermore, we denote by $\text{paths}(D)$ all the possible rooted paths that may occur in any T such that $T \vDash D$.

We now define several classes of DTDs, according to the complexity of the regular expressions present in the productions. Bear in mind that we disregard order in defining these classes. Our DTD classes form a hierarchy according to the following containment:

$$\text{Simple DTD} \subseteq \begin{matrix}\text{\#-DTD} \\ \vee\text{-DTD}\end{matrix} \subseteq \text{Arbitrary DTD}$$

Given a DTD, deciding which class it belongs to may be nontrivial in the worst case. However, we do not expect this to be an issue in practice, as humans do not typically write DTDs with complex nested regular expressions.

Simple DTDs Our definition for a simple DTD is closely related to that given in Arenas and Libkin [2]. Given an alphabet A, a regular expression over A is called *trivial* if it is of the form $s_1 \cdots s_n$, where for each s_i there is a letter $a_i \in A$ such that s_i is either $a_i, a_i?, a_i^+$ or a_i^*, and for $i \neq j$, $a_i \neq a_j$. A *simple* DTD is any DTD equivalent (under $\cong$) to some DTD where all the regular expressions appearing on the right-hand side of the productions are trivial. Note that every basic DTD is simple, but not vice versa. An example of a simple DTD is

$$\rho \rightsquigarrow ab^*, \;\; a \rightsquigarrow c^*, \;\; b \rightsquigarrow d^+e^*$$

#-DTDs #-DTDs are a proper extension of simple DTDs. This extension allows productions having more than one occurrence of the same alphabet symbol (the # is intended to represent the concept of number or counting). In other words, a #-DTD is an arbitrary DTD which does not require the use of disjunction. Formally, #-DTDs are exactly all DTDs equivalent (under $\cong$) to some DTD where all the the regular expressions appearing on the right-hand side of the productions contain no disjunctions. An example of a non-simple #-DTD is

$$\rho \rightsquigarrow aaab^*, \;\; a \rightsquigarrow c^*, \;\; b \rightsquigarrow ddeee^*$$

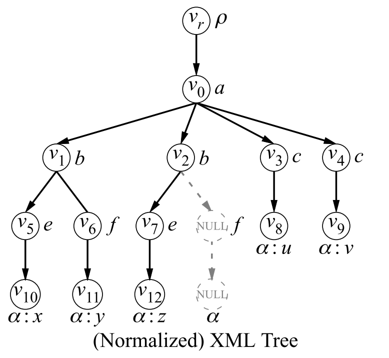
$\vee$-DTDs $\vee$-DTDs allow disjunction, but are otherwise a minor extension of simple DTDs. Our $\vee$-DTDs are exactly those called "disjunctive DTDs" in Arenas and Libkin [2]. We define a regular expression over an alphabet A to be a *simple disjunction* if $s = \varepsilon$, $s = a, a \in A$, or $s = s_1 \mid s_2$, where s_1 and s_2 are simple disjunctions over A_1 and A_2 respectively, with $A_1, A_2 \subseteq A$ and $A_1 \cap A_2 = \emptyset$. A $\vee$-DTD is one equivalent (under $\cong$) to one where every production is of the form $p \to s_1 s_2 \cdots s_k$, where every s_k is either a simple regular expression or a simple disjunction over a subalphabet A_k, and all the A_k are disjoint. $\vee$-DTDs do not allow multiple occurrences of a symbol in a production, nor do they allow nested disjunction. An example of a non-simple $\vee$-DTD is

$$\rho \rightsquigarrow (c|d)b^*, \;\; c \rightsquigarrow a^*, \;\; b \rightsquigarrow (e|f|g)(h|i)$$

Arbitrary DTDs Arbitrary DTDs represent the most general class of DTDs. They allow all features, including arbitrary disjunction.

2.3 Mapping an XML tree to a nested relation

Many existing definitions of XFDs implicitly rely on the nested relational structure of XML documents. In order to use the existing theory of relational functional dependencies, in this section we make this connection explicit. To each tree T, we associate a nested relation $R(T)$, which we unnest to a flat relation $U(R(T))$. First we give an intuitive, high-level illustration of $U(R(T))$, and then we give a more formal construction. Our illustration makes use of the example in Figure 3. In this illustration, we have separated the tree elements v_i from their labels in EL to make clear the difference between nodes and their labels.



(Normalized) XML Tree

ρ	$\rho.a$	$\rho.a.c$	$\rho.a.c.\alpha$
v_r	v_0	v_3	u
v_r	v_0	v_4	v

$$\bowtie$$

ρ	$\rho.a$	$\rho.a.b$	$\rho.a.b.e$	$\rho.a.b.f$	$\rho.a.b.e.\alpha$	$\rho.a.b.f.\alpha$
v_r	v_0	v_1	v_5	v_6	x	y
v_r	v_0	v_2	v_7	NULL	z	NULL

Unnested Relation

Fig. 3. Mapping an XML Document to an Unnested Relation

We start with a tree T for an XML document. First, we must normalize the tree to make it suitably homogeneous. For each path p that occurs in T, we take each prefix q of p. For each occurrence of q in T, we guarantee that q can be extended to a path matching p by adding special null nodes as necessary. In Figure 3, $\rho.a.b.f.\alpha$ matches a path in the tree, so we have to extend the second path matching $\rho.a.b$ with null nodes to extend it to $\rho.a.b.f.\alpha$.

After normalizing the tree, we take each maximal rooted path p occurring in the tree. We make a table with an attribute for each non-empty prefix of p. For each match to p in the normalized tree, we construct a row in this table. In this case, each attribute corresponds to a unique node v and we assign this attribute v if v is not an attribute (i.e. the label is not α) or is null; otherwise we assign it the attribute value for α in $\textsc{Val}$. The top table on the right hand side of Figure 3 represents the table for the path $\rho.a.c.\alpha$.

We then join all of these tables together. For each pair of tables, we use the common prefix of the pair as our join key. The bottom table on the right hand side of Figure 3 is the result of joining the tables for $\rho.a.b.e.\alpha$ and $\rho.a.b.f.\alpha$.

When $T \vDash D$, the resulting relation is essentially identical to $tuples_D(T)$ from Arenas and Libkin [2]. The only difference is that we have an additional root node in the tree and each XML attribute corresponds to two columns: one for its node address and the other for its value.

Nested Relations For the formal correspondence, we first need to introduce nested relations. Our definition of nested relations is very close to the standard one found in the literature [1, 12]. We assume that we have a countable set $\mathbf{C}$ of constants and a finite set B of attribute names. Complex values are any values of type τ, where τ is defined recursively, using tuple and set constructors, as

$$\tau = \mathbf{C} \mid \langle B_1 : \tau_1, \cdots B_k : \tau_k \rangle \mid \text{Set}(\tau)$$

where $k \geq 0$ and $B_1, \cdots B_k$ are finite strings formed over the alphabet B. Note here that our use of strings for attribute names is our only departure from the standard definition of nested relations. As with paths, we separate the elements of B with periods.

Next we define the notion of a complex value. Given $\tau \in \mathcal{T}$ the set of values of type τ, denoted $[\![\tau]\!]$, is defined by

1. $[\![\mathbf{C}]\!] = C$
2. $[\![\text{Set}(\tau)]\!] = \{\{v_1, \cdots v_k\} \mid v_i \in [\![\tau]\!], 1 \leq i \leq k\}$
3. $[\![\langle B_1 : \tau_1, \cdots B_k : \tau_k \rangle]\!] = \{\langle v_1, \cdots v_k \rangle \mid v_i \in [\![\tau_i]\!], 1 \leq i \leq k\}$

A complex value of type $\langle B_1 : \tau_1, \cdots B_k : \tau_k \rangle$ is called a tuple, whereas a complex value of type Set(τ) is called a set or a *nested relation*. A *flat relation* is a nested relation of type

$$\text{Set}(\langle B_1 : \mathbf{C}, \cdots B_k : \mathbf{C} \rangle)$$

for some $B_1, \cdots B_k$, meaning that it has no further complex types. For flat tuples, we often use the shorthand $\langle B_i : \mathbf{C} \rangle, B_i \in \mathcal{B}$, where $\mathcal{B}$ is a finite set of strings over B. If $\mathcal{B} = \{B_1, B_2, \cdots B_l\}$, then this represents the l-tuple type $\langle B_1 : \mathbf{C}, \cdots B_l : \mathbf{C} \rangle$.

Finally, we define a function attnames that extracts the attribute names from a flat relation. Given $\tau = \text{Set}(\langle B_1 : \mathbf{C}, \cdots B_k : \mathbf{C} \rangle)$, $\text{attnames}(\tau) = \{B_1, \cdots B_k\}$. We also define $\text{attnames}(\mathbf{C}) = \{\varepsilon\}$, where ε is the empty string.

Normalization The normalization described above is performed so a well-typed nested relation can be produced. Formally, it involves repeating the following procedure until no more new nodes can be added.

– Find a node v in T such that the path from the root to v is p, and $\exists x \in \textsc{El} \cup \textsc{Val}$ such that $p.x$ occurs in T, but v itself has no children labeled x

– Add to v a child node v' with label x.

We now extend the function num_T to a function num'_T to account for the nodes we added in the normalization process. Given a tree T' produced from T by normalization and a node $v \in T'$,

$$\text{num}'(v) = \begin{cases} \text{num}(v) & \text{if } v \in T, \\ \text{null} & \text{otherwise} \end{cases}$$

We also extend attval to attval$'$ such that for any new nodes v added through normalization with label a, $\text{attval}(v) = \text{null}$.

Producing the nested relation From the normalized T we produce a nested relation $R(T)$ by inductively performing the following steps, starting at the leaves.

Base case: T consists of a single node v. Then, $R(T)$ is a single-tuple relation where

– $R(T)$ has type $\text{Set}(\langle \text{lab}(v) : \mathbb{N} \cup \{\text{null}\}\rangle)$ and contains $\text{num}'(v)$ if $\text{lab}(v) \neq \alpha$
– $R(T)$ has type $\text{Set}(\langle \text{lab}(v) : \text{VAL} \cup \{\text{null}\}\rangle)$ and contains $\text{attval}'(v)$ if $\text{lab}(v) = \alpha$

Inductive case: T has a root node v_{root} and various subtrees, with the labels on subtrees forming a set $\{l_1, \cdots l_k\} \subseteq \text{EL}$. We partition the subtrees into k sets $S_1, \cdots S_k$ based on the root labels. By induction, we know that for a set S_m, all subtrees $T_{m1}, \cdots T_{ml}$ in the set can be represented as nested relations $R(T_{m1}), \cdots R(T_{ml})$. Due to the normalization process, all these relations have the same type; we call this type τ_m.

$R(T)$ itself is then a single-tuple relation of type

$$\text{Set}(\langle l(v) : \mathbb{N} \cup \{\text{null}\}, \varepsilon : \tau_1, \cdots, \varepsilon : \tau_k\rangle)$$

where ε is the empty string. The value in the first element is $\text{num}'(v)$, and the value in element $j, 2 \leq j \leq k+1$, is $\bigcup_{T_{jl} \in S_j} R(T_{jl})$.

Unnesting Our unnesting operation U on a relation R is also very close to the standard one, and is defined inductively. We define $F : \mathcal{T} \to \mathcal{T}$ to be the unnesting map on types, such that

– $F(\mathbf{C}) = \mathbf{C}$
– $F(\langle B_1 : \tau_1, \cdots, B_k : \tau_k\rangle) = \langle B : \mathbf{C}\rangle, B \in \{B_i.B_{ij} \,|\, B_{ij} \in \text{attnames}(F(\tau_i))\}$
– $F(\text{Set}(\tau)) = \text{Set}(F(\tau))$

Inductively, a complex value $v : \tau$, $U(v)$ has type $F(\tau)$ and the following value

– $U(c)$, where $c \in C$, is a unary, single-element relation with element c, attribute name ε and type $\mathbf{C}$.
– For $\langle v_1, \cdots v_k\rangle$ of type $\langle B_1 : \tau_1, \cdots B_k : \tau_k\rangle$, we define $U(\langle v_1, \cdots v_k\rangle) = \prod_i U(v_i)$. (The product operator used is the ordinary cross product).
– For $\{v_1, \cdots v_k\}$ of type $\text{Set}(\tau)$, we define $U(\{v_1, \cdots v_k\}) = \bigcup_i U(v_i)$.

Note that U is the identity on flat relations, as might be expected.

The following proposition is clear from our construction.

Proposition 1. *The three steps of the above mapping take a tree T to a flat relation $U(R(T))$ such that*
– *The schema of $U(R(T))$ is exactly $paths(T)$, and*
– *If p is any path in T, and we do a projection on $U(R(T))$ to retain only prefixes of p, each tuple in the projected relation corresponds to exactly one specific occurrence of p in T.*

3 XML Functional Dependencies

Given the basic notation, we can now define XFDs, and describe the implication and consistency problems. As we mentioned before, our definition of XFDs must account explicitly for the possibility of null values. Several means of handling null values have been suggested for the relational model [3, 16]. We adopt the definition in [3], which is also the one used in Arenas and Libkin [2]: given a relation R over a set of attributes U and $A, B \subseteq U$, $A \to B$ holds if for any two tuples $t_1, t_2 \in R$ that agree and are nonnull on all of A, the t_i are equal (though possibly both null) for each attribute in B.

3.1 Tree patterns and functional dependencies

Relational functional dependencies are defined on tuples in the relation. The corresponding notion for an XML tree is a match for a *tree pattern*. Syntactically, a tree pattern φ is a document in which none of the attribute nodes (i.e. nodes labeled α) are yet mapped to a value in VAL. A *simple* pattern is a tree pattern where no node has two children with the same label $l \in$ EL. In a simple pattern, every rooted path occurs at most once; from now, we assume that all patterns are simple.

In order to use tree patterns to define XML functional dependencies, we must first define what it means to *match* a pattern in a document T. Intuitively, we want to match the pattern in the document "as far as possible", allowing incomplete matches only when the document does not allow a full match. Formally, given a tree T and a pattern φ, a match for φ in T is a function μ : nodes(φ) $\to$ nodes(T) $\cup$ {null} where

- μ maps the root of φ to the root of T.
- For all nodes v, $\mu(v)$ is either null, or it preserves the label of v.
- If v' is a child of v and $\mu(v')$ is not null, then $\mu(v)$ is also not null and $\mu(v')$ is a child of $\mu(v)$.
- If v' is a child of v and $\mu(v')$ is null while $\mu(v)$ is not, then $\mu(v)$ could not have had any child with the same label as v' (i.e. no "premature" null values are allowed).

For any path $p \in \varphi$, we use $\mu(p)$ to represent the image under μ for the node at the end of this path. We let $\mathcal{M}_{\varphi,T} = \{\, \mu \mid \mu$ is a match for φ in $T \,\}$.

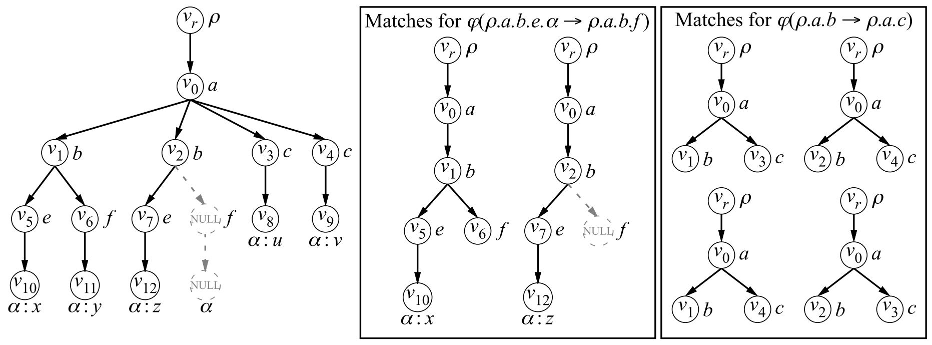


Fig. 4. Sample Tree Pattern Matches

Definition 1. *A* functional dependency *$\sigma = A \to B$ consists of two subsets $A, B \subseteq$* PATHS. *We let $\varphi(\sigma)$ be the smallest tree pattern (with respect to number of nodes) in which all paths in A and B occur. Furthermore, we let $=_{fd}$ be an equality relation that compares nodes by identity equality if they are not attribute nodes (i.e. labeled by α), and by value equality otherwise. The dependency holds if, for any two matches $\mu_1, \mu_2 \in \mathcal{M}_{\varphi(\sigma),T}$, whenever we have $\mu_1(p), \mu_2(p) \neq$ null and $\mu_1(p) =_{fd} \mu_2(p)$ for all $p \in A$, then for all $q \in B$, $\mu_1(q) =_{fd} \mu_2(q)$.*

We illustrate this definition in Figure 4. Recall that our patterns are simple, and so each path in the pattern must occur exactly once. This means that there are only two matches for the pattern $\varphi(\rho.a.b.e.\alpha \to \rho.a.b.f)$ within this tree; no match for a simple pattern can contain both v_7 and v_6. If $x \neq z$ then the corresponding XFD is satisfied, otherwise it is not, as the f node is null in one match and non-null in another. Similarly, there are four matches for the pattern $\varphi(\rho.a.b \to \rho.a.c)$ in Figure 4; this XFD can never be satisfied in this tree.

Theorem 1.

1. *A tree pattern XFD $A \to B$ holds on T if and only if the relational FD $A \to B$ holds on $U(R(T))$.*
2. *Tree tuple XFDs [2] are expressible as tree pattern XFDs. For any DTD D and tree tuple $A \to B$, there is a tree pattern $A' \to B'$ such that for any document $T \vDash D$, $A \to B$ holds on T if and only if $A' \to B'$ does.*

Proof. For the first claim, we assume that all of the paths in A, B have at least one non-null match in a document T. Otherwise $A \to B$ trivially holds in both T and $U(R(T))$. Then we need only show that each match of a simple tree pattern in T is the prefix-preserving projection of a single row in $U(R(T))$, and vice versa. Consider first a match to a simple tree pattern in T. The range of this match is a subtree of the normalized version of T. Each path in this tree is a row in one of the pre-joined components of $U(R(T))$. As the pattern is simple, each path occurs exactly once, and so these components join to form a single row in $U(R(T))$.

Now consider a row in $U(R(T))$, and project out attributes in such a way that prefixes are preserved. This row was the result of a join from the individual path tables. As the projection is prefix-preserving, we can construct this projection as the join of prefix-preserving projections on the component tables. Take the maximal path for each table and construct the simple tree pattern from these tables. The row is the subtree in the normalized version of T isomorphic to this pattern, and hence the range of a match.

For the second claim, take any tree-tuple XFD [2] $A \to B$. Construct A' by taking each path in A and prefixing it by ρ, and suffixing those paths that end in an attribute with α. Construct B' similarly. Now take any document $T \vDash D$. $A \to B$ holds in T if and only if it holds as a relational functional dependency on $tuples_D(T)$. $tuples_D(T)$ differs from $U(R(T))$ only in that

– it lacks an attribute for ρ and an attribute representing the node (not just the value) for each XML attribute, and
– it has attributes for all paths enforced by D, even if they do not occur in T (e.g. $\rho \rightsquigarrow ab^*$ is in D and T has no b nodes).

Our construction of A', B' ensures that the first set of attributes do not appear in A', B'. We can remove any paths from A and B that do not occur in T, as any match must be $(null)$ (and thus equivalent) on those values. Thus the restriction of $tuples_D(T)$ to $A \cup B$ is identical to the restriction of $U(R(T))$ to $A' \cup B'$ and our proof follows from the first claim.

3.2 The Implication Problem

Our primary area of focus is the implication problem. Given an XFD σ and document T, we write $T \models \sigma$ to mean σ holds in T. Given a set Σ, we write $T \models \Sigma$ if every $\tau \in \Sigma$ holds in T. Finally, we write $\Sigma \models \sigma$ to mean that, for all $T \models \Sigma$, $T \models \sigma$. Similarly, for a DTD D, $\Sigma \models_D \sigma$ means that, for any $T \models \Sigma$ and $T \models D$, we have $T \models \sigma$. Given (Σ, σ) [or in the case of a DTD, (Σ, σ, D)], the implication problem is to decide whether $\Sigma \models \sigma)$ [respectively, $\Sigma \models_D \sigma$].

Obviously in the practical case we are most interested in finite implication. However, in standard database theory, most reasoning is done with unrestricted implication, as the results are often cleaner and much simpler. The propositions below outline the relationship between finite and unrestricted implication.

Proposition 2. *In the absence of a DTD, finite and unrestricted implication for XFDs coincide.*

Proof. It suffices to show that if $\Sigma \not\models^{unr} \sigma$ for some $\sigma = A \to B$ then there is a finite document T such that $T \models \Sigma$, $T \not\models \sigma$. Given that $\Sigma \not\models^{unr} \sigma$, there must exist some (potentially infinite) tree T' such that $T' \models \Sigma$, $T' \not\models \sigma$.

Let m be the maximum length of any path in any $\varphi(\sigma')$, $\sigma' \in \Sigma \cup \{\sigma\}$. We obtain T'' from T' by removing all nodes whose distance from the root is more than m. Clearly we still have $T'' \models \Sigma$ and $T'' \not\models \sigma$. Obtain $U(R(T''))$. It must contain two tuples witnessing $T'' \not\models \sigma$; retain only the nodes of T'' directly involved in those tuples. It is clear that we have obtained a finite tree with the property desired.

Proposition 3. *In the presence of a nonrecursive DTD, finite and unrestricted implication for FDs coincide.*

Proof. This is immediate, as any tree satisfying a nonrecursive DTD must be finite.

However, we have the following negative result for recursive DTDs.

Proposition 4. *In the presence of an arbitrary DTD, finite and unrestricted implication for FDs do not coincide.*

Proof. Consider the DTD with the single production $\rho \rightsquigarrow c^*, c \rightsquigarrow c$, and the dependency $\sigma = \rho \to \rho.c$. The only finite tree satisfying D is the tree consisting just of the root; thus $\models_D^{fin} \sigma$. On the other hand, clearly $\not\models_D^{unr} \sigma$.

Thus, the mathematics behind our techniques and algorithms generally assume that trees are unrestricted (i.e. potentially infinite). Infinite branching in a tree can be handled in the relational model - it produces infinite numbers of tuples, and such infinite relations are known in the theoretical literature. The existence of infinite length *paths*, however, is more difficult to deal with. Our relational mapping requires/induces a relational schema of finite arity, and is therefore not directly suited for working with trees that have infinite paths.

Nevertheless, we observe that for any given instance of an implication or closure problem with a corresponding set of XFDs Σ, there is a finite cutoff to the depth at which the tree need ever be considered. If we let k be the maximal depth of any tree pattern for any $\sigma \in \Sigma$, it is clear that no dependency in Σ can affect any part of the tree which is more than k edges from the root (DTDs, of course, can and do affect nodes at arbitrary depth, but XFDs cannot).

We can use this fact to our advantage. Our reasoning usually occurs in the relational model, using our mapping method. As a first step to solving any problem instance, we build a separate mapping with a separate relational schema R for every problem instance; this mapping is a function of the set Σ of all XFDs appearing in the instance. When reasoning in the absence of a DTD, the attributes of R are exactly the subset of PATHS of length k or less, which we write as PATHS $\upharpoonright k$. With a DTD D, R need only contain all the attributes in paths$(D) \upharpoonright k$.

A similar restriction is possible with regards to our alphabet. Recall that EL is countable, but need not be finite. In the presence of a DTD, the label alphabet becomes finite automatically. However, in the absence of a DTD,we still only need a finite subset of EL as label alphabet. Again this is because any alphabet symbols which are not on any paths mentioned in the problem instance can be ignored; in the absence of a DTD, no dependency a set Σ can have any effect on dependencies whose paths contain label symbols not used in Σ.

It follows from our axiomatization in Section 4.2 that we can assume that each instance (Σ, σ) is given so that σ has the form $A \to b$, with b a single path, and all $C \to d \in \Sigma$ have a single path on the right-hand side as well. For our chase algorithm, we assume that the dependencies are in this form; that allows us to give a clearer presentation and properly compare our complexity results with existing work, which makes the same assumption [2]. If our input is not in this form, conversion is polynomial, so tractability is not affected.

3.3 The Consistency Problem

As we saw in Section 1, it is possible to define sets of XFDs which are inconsistent with certain DTDs. Formally, given a set of XFDs Σ and a DTD D, we say that (Σ, D) is consistent if and only if there exists at least one finite document T such that $T \models D$ and $T \models \Sigma$. Given (Σ, D), the consistency problem is to decide whether it is consistent. As with the implication problem, our algorithm descriptions assume that Σ is given such that all XFDs have exactly one path on the right-hand side.

4 Implication without a DTD

Before we understand how DTDs affect the XFDs satisfied by a document, we must first understand the theory of XFDs alone. In this case, consistency is a non-issue and so we only concern ourselves with the implication problem.

4.1 Chase algorithm for FD implication

This section presents a fast algorithm for deciding XFD implication. This algorithm is essentially the standard chase algorithm adapted to XML documents. Suppose $\sigma = A \rightarrow b$. We set up a tableau $\mathcal{T}$ as follows: there are two rows, and one column for every path (and prefix of a path) in A and b. In the ρ column and in each column corresponding to a path in A itself, we insert two identical variables from some indexed set $\{v_i\}_{i \in I}$; however, no variables are repeated between columns. For all other entries in the tableau, we assign unique variables.

Consider, as an example, the dependency $\{\rho.a.b, \rho.a.c\} \rightarrow \rho.a.d.e$. This corresponds to the following tableau.

ρ	$\rho.a$	$\rho.a.b$	$\rho.a.c$	$\rho.a.d$	$\rho.a.d.e$
v_1	v_2	v_3	v_4	v_5	v_7
v_1	v_9	v_3	v_4	v_6	v_8

Next, we define the set $\widehat{\Sigma}$ of functional dependencies that are used to chase on $\mathcal{T}$. We let X be the set of attributes of $\mathcal{T}$ (equal to paths$(\varphi(\sigma))$); $\hat{\Sigma}$ consists exactly of Σ plus the following additional dependencies:

- $p.d \rightarrow p$ for every $p.x \in X, x \in$ EL (non-attribute parents are unique).
- $p \rightarrow p.\alpha$ for every $p.\alpha \in X$ (attributes have unique values).

The algorithm now involves chasing with $\widehat{\Sigma}$. At each step of the chase we have a dependency $C \rightarrow d$ and attempt to unify the two variables in the d column. This is a legal move if either (or both) the following hold:

1. All C column values of both rows are equal.
2. Let q be the longest prefix of d on which the two rows agree. The rows agree on all paths in C of the form $q.x.t$, $x \in$ EL, where $q.x$ is a prefix of d.

The correctness of the first chase rule is clear, as it is the standard one. To understand the second rule, consider the tableau $\mathcal{T}$ represented by the first two rows in Figure 5, and suppose we are chasing with the dependency $\{\rho.a.b, \rho.a.c\} \rightarrow \rho.a.c.f$. This tableau corresponds to the tree on the right hand side of Figure 5. However, when we convert this tree T back to its relation $U(R(T))$, the tree structure gives us two additional rows to the tableau. And from these two new rows we see that we need to unify v_8 and v_9 to satisfy our dependency.

	ρ	$\rho.a$	$\rho.a.b$	$\rho.a.c$	$\rho.a.b.d$	$\rho.a.b.e$	$\rho.a.c.f$
$\mathcal{T}$	v_r	v_0	v_1	v_3	v_4	v_5	v_8
$\mathcal{T}$	v_r	v_0	v_2	v_3	v_6	v_7	v_9
$U(R(T))$	v_r	v_0	v_2	v_3	v_6	v_7	v_8
$U(R(T))$	v_r	v_0	v_1	v_3	v_4	v_5	v_9

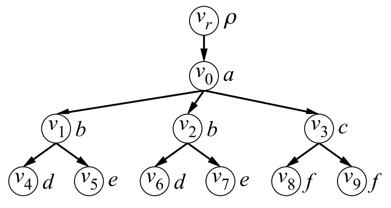


Fig. 5. Chasing with $\{\rho.a.b, \rho.a.c\} \rightarrow \rho.a.c.f$

One way to solve this problem is to construct our tableau so that it is closed under transformation to a tree and back to $U(R(T))$. However, the size of this tableau could be exponential in the size of our XFD σ. Fortunately, the regularity of the tree structure makes this unnecessary. We see that the two rows in the tableau for Figure 5 have the same value of $\rho.a.c$. As $\rho.a.b$ is a branch independent of $\rho.a.c$, the tree structure ensures that the equality between the values for $\rho.a.c$ is enough to unify $\rho.a.c.f$. The second chase rule generalizes this idea, allowing us to restrict our chase to just two rows.

Correctness of the Chase When the chase terminates (as it clearly must), we can construct a tree T_{chase} from the tableau.

Lemma 1. *After the chase terminates, $\mathcal{T}$ corresponds to $U(R(T_{chase}))$ for some document T_{chase}; moreover, $T_{chase} \models \Sigma$.*

Proof. We can obtain T_{chase} by the following process.

- For the subset of paths in $\mathcal{T}$ that does not end in an attribute node, generate a unique node for each variable v_i in $\mathcal{T}$. Connect and label these nodes as suggested by the structure of $\mathcal{T}$.
- For those paths in $\mathcal{T}$ ending in attribute nodes, generate nodes as appropriate. If the two variables in the appropriate columns of $\mathcal{T}$ are equal, add to the nodes equal values from VAL. Otherwise add different values, which should be unique in the entire tree.

It is clear from the description of the chase algorithm that T_{chase} is be a valid tree, as we put the tree "bookkeeping" constraints in as explicit XFDs. To see that $T_{chase} \models \Sigma$, suppose not; let $(C \to d) \in \Sigma$ not hold for T_{chase}. Then we have a witness pair μ_1, μ_2 of matches for this fact. Consequently the two tableau rows for d cannot be equal, because T_{chase} contains at most two occurrences of d paths, and we need both of them for the witness pair.

If the tableau rows agree on all the C columns, clearly the algorithm terminated too early, as the first rule for unification should have fired. If the two rows don't agree on all of C, they must agree on at least some prefix q of d, in the worst case the root itself. Letting the next character in d after p to be x, we split C into two parts — those attributes that are extensions of $q.x$ and those that are not. If for any $q.x.t \in C$, we have the two columns not equal, $C \to d$ holds after all, as any matches that pick up the two different d branches cannot agree on C. And if all the $q.x.t$ columns are equal, again the algorithm terminated too soon, because we should have applied the second rule to unify the d values.

By the following theorem, the values in the two columns for this tableau d are equal if and only if $\Sigma \models \sigma$. Hence our chase is correct.

Theorem 2. *After the chase terminates, $T_{chase} \models \sigma$ if and only if $\Sigma \models \sigma$.*

Proof. Clearly if T_{chase} does not satisfy $\sigma = (A \to b)$, we have that $\Sigma \not\models \sigma$, as $T_{chase} \models \Sigma$ by the previous lemma. On the other hand, suppose $T_{chase} \models \sigma$. Consider any tree T' such that $T' \models \Sigma$. Suppose for a contradiction that $T' \not\models \sigma$. Then there are witnesses μ_1, μ_2 for this fact. These witnesses must agree and be non-null on all the path identifiers in A. Furthermore, one of the matches must be non-null on b and thus on all of $\varphi(\sigma)$. Without loss of generality suppose $\mu_1(b) \neq$ null.

As the pattern $\varphi(\sigma)$ is simple, it is clear from the proof of Theorem 1 that each match corresponds to a row in $U(R(T'))$. Let $U(R(T'), \mu_1, \mu_2)$ be the table with just those two rows. Also, let $\mathcal{T}_0$ be our tableau at the start of the chase. We show that at each stage of the chase, $U(R(T'), \mu_1, \mu_2)$ must be at least as unified as the tableau so far (though it may have more unifications). As the b column is unified at the end of the chase, this gives us our contradiction.

To formalize this idea, we say that a relation R *covers* a relation R' if they have the same schema and there is a one-to-one mapping from tuples in R' to tuples in R such that if two tuples have the same value for a column in R, then they have the same value for that column in R'. As the initial tableau $\mathcal{T}_0$ only unifies those paths in A, it clearly covers $U(R(T'), \mu_1, \mu_2)$. We need to show that $U(R(T'), \mu_1, \mu_2)$ is covered by the tableau at each step of the chase. Suppose we are stage n of the chase, and that $\mathcal{T}_n$ covers $U(R(T'), \mu_1, \mu_2)$. We attempt to unify with some $C \to d \in \widehat{\Sigma}$. If we cannot unify or if d is already unified, we are done. If we do unify, we either unify with rule 1 or rule 2. Suppose first that we unify with rule 1. Then all of the columns corresponding to C must be the same in both rows. As $\mathcal{T}_n$ covers $U(R(T'), \mu_1, \mu_2)$ and μ_1 is not-null, these columns must be the same and not null in $U(R(T'), \mu_1, \mu_2)$. As $T' \vDash \Sigma$, $T' \vDash \widehat{\Sigma}$ (our additional XFDs hold because of the tree structure of T'). Hence μ_1, μ_2 must agree on d if it exists, so when we unify the d column $\mathcal{T}_{n+1}$ still covers $U(R(T'), \mu_1, \mu_2)$.

Suppose we are stage n of the chase, and that $\mathcal{T}_n$ covers $U(R(T'), \mu_1, \mu_2)$. At this stage we attempt to unify with some $C \to d \in \widehat{\Sigma}$. If we cannot unify, or if d is already unified, we are done. If we do unify, we either unify with either the first or second chase rule. Suppose that we unify with the first row. Then all of the columns corresponding to C must be the same in both rows. As $\mathcal{T}_n$ covers $U(R(T'), \mu_1, \mu_2)$ and μ_1 is non-null, these columns must be the same and not null in $U(R(T'), \mu_1, \mu_2)$. Recall that $\widehat{\Sigma}$ is just Σ plus some additional XFDs representing the tree structure of a document. Hence $T' \vDash \widehat{\Sigma}$. Thus μ_1, μ_2 must agree on d if it exists, so when we unify the d column $\mathcal{T}_{n+1}$ still covers $U(R(T'), \mu_1, \mu_2)$.

Now suppose that we unify with respect to the second chase rule. Expand $\mathcal{T}_n$ of $\overline{\mathcal{T}_n}$ by converting it to a tree and then back to $U(R(T))$ as demonstrated in Figure 5. Expand $U(R(T'), \mu_1, \mu_2)$ to $\overline{U(R(T'), \mu_1, \mu_2)}$ in the same way. The rows in these expansions are determined entirely by the tree structure of the attribute columns, and so $\overline{\mathcal{T}_n}$ must cover $\overline{U(R(T'), \mu_1, \mu_2)}$, and this covering can be constructed to extend the initial covering.

Let $r_1 \in \mathcal{T}_n$ be the row which corresponds to μ_1 in covering. We see from the description of the second rule in Figure 5 that there must be some row in $\overline{\mathcal{T}_n}$ that agrees with r_1 on all the values in C but not d. As $\overline{\mathcal{T}_n}$ covers $\overline{U(R(T'), \mu_1, \mu_2)}$ the second row corresponds to some row in $\overline{U(R(T'), \mu_1, \mu_2)}$ which agrees with the μ_1 row on C. As these values are completely non-null and $T' \vDash C \to d$, these two rows agree on d. There are only two different values for d in $\overline{\mathcal{T}_n}$ and these are different in these two rows. Thus as $\overline{\mathcal{T}_n}$ covers $\overline{U(R(T'), \mu_1, \mu_2)}$ we have $\mu_1(d) = \mu_2(d)$. Hence when we unify the d column $\mathcal{T}_{n+1}$ still covers $U(R(T'), \mu_1, \mu_2)$.

Complexity of the Chase Computing the chase naively is $O(|\Sigma||\sigma|)$. We have at most $|\sigma|$ iterations of the outer loop, and finding a dependency that satsifies either rule 1 or rule 2 requires a scan of Σ. There is the issue that we are chasing with $\widehat{\Sigma}$ and not Σ. However, the functional dependencies in $\widehat{\Sigma} \setminus \Sigma$ depend on a single column and can be attached to that column for constant time evaluation.

It is easy to create a data structure that tracks, for each $C \to d \in \Sigma$, the greatest prefix of d unified in the chase so far. This allows us to adapt rule 2 to the linear time chase presented in Beeri and Bernstein [4]. Our algorithm consists of several parts.

Step 1: Construct a reference tree for $\varphi(\sigma)$. We introduce a data structure which we call `ReferenceTreeNode` with the following fields.

```
struct ReferenceTreeNode {
    int column;
    ReferenceTreeNode⋆ internalParent;
    ReferenceTreeNode⋆ attributeChild;
    Hashtable<Label,ReferenceTreeNode⋆> children
}
```

By constructing a tree of these nodes, we use this data structure to represent $\varphi(\sigma)$. Each node correponds to a path prefix in $\varphi(\sigma)$. The field `column` represents the column of the tableau for that prefix. The hashtable `children` links this prefix to its extensions according to the value of the next label. As a result, once this structure is built, given a path p, it requires $|p|$ time to find the corresponding `ReferenceTreeNode` and hence the correct tableau column.

The two additional fields in `ReferenceTreeNode` model the additional XFDs of $\widehat{\Sigma}$. If the `ReferenceTreeNode` represents a path ending in an internal node, `internalParent` is a reference to a parent of this node; otherwise it is null. This corresponds to the additional XFD $p.d \to p$ for every $p.x \in X$ of the chase. Similarly, the `ReferenceTreeNode` represents a path ending in a node with an attribute child, we let `attributeChild` refer to that child. For all others, this value is null.

To construct this tree, we allocate an array `Tree[0:`n`]`, where n is the number of nodes in $\varphi(\sigma)$. We initialize this array with Algorithm 1. In this algorithm, the function INITIALIZE_NODE$(node, k)$ initializes a `ReferenceTreeNode` with column $k$, assigns `internalParent` and `attributeChild` to null, and creates an empty hashtable. It is clear that this algorithm is simply a single scan of σ and hence runs in $O(|\sigma|)$ time. Similarly, it is clear that it produces a tree with the properties stated above.

Step 2: Construct the data structures for Σ. To represent our XFDs $C \to d$, we need three different types of data structures. The first is a summary structure for the XFD.

```
struct Summary {
    PrefixNode[1:$k$] prefixes;
    int dColumn;
    int unifiedDepth;
    int leftToUnify;
}
```

The field `prefixes` is an array of special data structures for the prefixes of d; $k$ is the depth of d from the root. The value `dColumn` represents the position of d in the array `Tree`. The field `unifiedDepth` keeps track of the depth from the root of the greatest prefix of d that is unified so far in the chase; as the elements of `prefixes` are ordered by depth, it acts as an index into this array. Finally, the counter `leftToUnify` keeps track of the remaining paths left to unify in C before we must unify d.

Algorithm 1 INITIALIZE_TREE($A \rightarrow b$)

```
Require: $A \rightarrow b$ is an XFD.
Ensure: Tree represents the tree pattern $\varphi(\sigma)$
 1: free = 1
 2: INITIALIZE_NODE(&Tree[0],0)
 3: for each path $p$ in $A \rightarrow b$ do
 4:   let $\rho.p_1.p2.\ldots.p_k$ be the labels of $p$.
 5:   label = 1
 6:   current = &Tree[0], previous = null
 7:   while label $\leq k$ do
 8:     previous = current
 9:     if $p_{\texttt{label}}$ is in current.children then
10:       current = current.children.get($p_{\texttt{label}}$)
11:     else
12:       current = INITIALIZE_NODE(&Tree[free],free)
13:       free = free+1
14:       previous.children.put($p_{\texttt{label}}$,current)
15:       if $p_{\texttt{label}}$ is an internal node then
16:         current.internalParent = previous
17:       end if
18:       if $p_{\texttt{label}}$ is an attribute node then
19:         previous.attributeChild = current
20:       end if
21:     end if
22:     label = label+1
23:   end while
24: end for
```

For each XFD $C \rightarrow d$, a `PrefixNode` represents the prefixes of the path d.

```
struct PrefixNode {
    Summary* summary;
    int column;
    int depth;
    int cNodeToUnify;
}
```

Each `PrefixNode` corresponds to a single XFD and so has a single `Summary` structure. The value `column` represents the position of this element in the array `Tree`, while `depth` is its depth from the root, and thus is its index in `summary.prefixes`. Finally, the value `cNodesToUnify` counts the number of $c \in C$ which are not yet unified, and for which this node is the *greatest* common prefix between c and d. This node does not keep track of those elements of C for which it is a prefix, but not the greatest common prefix.

Similarly, for each XFD $C \rightarrow d$, a `CNode` represents a path $c \in C$.

```
struct CNode {
    PrefixNode* prefix;
    int column;
};
```

The value `column` is the same as for `PrefixNode`, while `prefix` refers to the greatest common prefix of c and d. Note therefore that any $c \in C$ which is a prefix of d is represented by both a `CNode` and a `PrefixNode`.

We let n be as in step 1, and allocate `AttList[1:`n`]` and `PrefList[1:`n`]`. The elements of the first array are sets of `CNode` objects, while the elements of the second array are of type `PrefixNode`. As in Beeri and Bernstein [4], each element of `AttList` is the set of `CNode` objects that represent this attribute in the tableau. The array `PrefixNode` is what we use to integrate the second chase rule. These arrays start out with all sets empty; we initialize them by running Algorithm 2 for each $C \rightarrow d \in \Sigma$. The subroutine for INITIALIZE_SUMMARY is shown in Algorithm 3.

Algorithm 2 INITIALIZE_XFD($C \rightarrow d$)

```
Require: $C \rightarrow d$ is an XFD.
Ensure: AttList contains the set of CNode* in $C \rightarrow d$ that depend on each attribute
Ensure: PrefList contains the set of PrefixNode* in $C \rightarrow d$ that depend on each attribute
 1: summary = INITIALIZE_SUMMARY($b$)
 2: if summary == null then
 3:     return
 4: end if
 5: allocated = new List<CNode>
 6: for each $c \in C$ do
 7:     let $\rho.c_1.c_2.\ldots.c_m$ be the labels of $c$.
 8:     label = 1, position = 0, agree = 0
 9:     while label $\leq m$ do
10:         if $d_{\text{label}}$ is in Tree[position].children then
11:             tableau = Tree[position].children.get($c_{\text{label}}$)
12:             agree = ($c_{\text{label}}$ == $d_{\text{label}}$ ? agree+1 : agree)
13:             label = label+1, position = tableau.column
14:         else
15:             return
16:         end if
17:     end while
18:     node = new CNode, node.column = position
19:     node.prefix = &summary.prefixes[agree]
20:     node.prefix.cNodeToUnify++, summary.leftToUnify++
21:     Add node to allocated
22: end for
23: for index = 0 to $k$ do
24:     Add summary.prefixes[index] to PrefList[prefixes[index].column]
25: end for
26: for each node in allocated do
27:     Add node to AttList[node.column]
28: end for
```

The **return** statements at lines 3 and 15 represent the fact that part of $C \rightarrow d$ lies outside the tableau and so we can safely ignore this functional dependency. Hence we do not update `AttList` until we are sure that $C \rightarrow d$ lies in the tableau. Again, it is clear that using this algorithm to initialize `AttList` is a single scan of Σ, and hence time $O(|\Sigma|)$.

Algorithm 3 INITIALIZE_SUMMARY(d)

```
Ensure: Returns a Summary structure for $d$.
1: let $\rho.d_1.d_2.\dots.d_k$ be the labels of $d$.
2: summary = new Summary, summary.prefixes = new PrefixNode[1:$k$]
3: label = 1, position = 0
4: while label $\leq k$ do
5:   if $d_{\texttt{label}}$ is in Tree[position].children then
6:     tableau = Tree[tableauPos].children.get($d_{\texttt{label}}$)
7:     prefixes[label] = new PrefixNode
8:     prefixes[label].summary = &summary
9:     prefixes[label].column = tableau.column
10:    prefixes[label].depth = label
11:    prefixes[label].cNodeToUnify = 0
12:    label = label+1, position = tableau.column
13:  else
14:    return null
15:  end if
16: end while
17: summary.dColumn = summary.prefixes[$k$].column
18: summary.unifiedDepth = 0
19: return summary
```

Step 3: The Main Algorithm. Now that we have the structures `Tree` and `AttList`, we are ready to perform the chase. The chase is shown in Algorithm 5. It has two subroutines: POSITION_OF and RESET_DEPTH. The first is a simple scan of a path p to find the location of it in `Tree`, similar to lines 7-17 of Algorithm 2. The second subroutine is illustrated in Algorithm 4. This algorithm crawls down the list of `PrefixNode` objects to compute the number of attributes left to unify.

Algorithm 4 RESET_DEPTH(`prefix`,`summary`)

```
Require: prefix.depth < summary.unifiedNode
Ensure: summary.unifiedDepth = prefix.depth
Ensure: summary.leftToUnify = # of remaining $c \in C$ for which it is a prefix
1: for depth = summary.unifiedDepth to prefix.depth-1 do
2:   current = summary.prefixes[summary.unifiedDepth]
3:   summary.leftToUnify -= current.cNodeToUnify
4: end for
5: summary.unifiedDepth = prefix.depth
6: return summary
```

First, we claim that this algorithm runs in $O(|\sigma| + |\Sigma|)$ time. Using hashtables, the set lookups are constant time. Thus the subroutine subroutine POSITION_OF(p,&Tree[0]) takes $|p|$ time, and so lines 1 to 9 take $O(|\sigma|)$ time. Next, note that we never examine a

Algorithm 5 CHASE($A \rightarrow b, \Sigma$)

```
Require: Tree and AttList are initialized
Ensure: The chase is successful if and only if we return true.
 1: bPosition = POSITION_OF($b$,&Tree[0])
 2: pending = new Queue<int>
 3: visited = new Set<int>
 4: for each $a \in A$ do
 5:   position = POSITION_OF($a$,&Tree[0]);
 6:   if position is not in visited then
 7:     Add position to visited and pending
 8:   end if
 9: end for
10: while pending is not empty do
11:   next = pending.removeFromFront()
12:   if bPosition == next then
13:     return true
14:   end if
15:   for each cnode in AttList[next] do
16:     node.prefix.cNodeToUnify--
17:     if node.prefix.summary.unifiedDepth $\geq$ node.prefix.depth then
18:       node.prefix.summary.leftToUnify--
19:       if node.prefix.summary.leftToUnify == 0 then
20:         if node.prefix.summary.dColumn is not in visited then
21:           Add node.prefix.summary.dColumn to visited and pending
22:         end if
23:       end if
24:     end if
25:   end for
26:   for each PrefixNode prefix in AttList[next] do
27:     if prefix.depth > prefix.summary.unifiedDepth then
28:       RESET_DEPTH(prefix,summary)
29:       if prefix.summary.leftToUnify == 0 then
30:         if prefix.summary.dColumn is not in visited then
31:           Add prefix.summary.dColumn to visited and pending
32:         end if
33:       end if
34:     end if
35:   end for
36:   if Tree[next].internalParent is not null and not in visited then
37:     Add Tree[next].internalParent to visited and pending
38:   end if
39:   if Tree[next].attributeChild is not null and not in visited then
40:     Add Tree[next].attributeChild to visited and pending
41:   end if
42: end while
43: return false
```

set `AttList[`i`]` or `PrefList[`i`]` more than once. The total number of elements of all of the sets in `AttList` is $O(|\Sigma|)$ and similarly for `PrefList`. Therefore, the only thing that we need be concerned with is the cost of RESET_DEPTH. However, RESET_DEPTH can only move down a linked list of `PrefixNode` objects. Therefore, the total cost of all the RESET_DEPTH calls in lines 10 to 42 is $O(|\Sigma|)$.

Next we show that Algorithm 5 implements the chase correctly. Note that, because the root is always unified, the second chase rule is a proper generalization of the second, and so we can ignore the first chase rule. We can treat each addition to the queue `pending` as a unification of an attribute. We need only show that at the start of each loop on line 10, every `PrefixNode` and `Summary` object is correct. That is, for every `PrefixNode` object corresponding to $C \to d$

- `cNodeToUnify` is the number of $c \in C$ for which this is the greatest common prefix with d, and for which `cnode.column` is not in `visited`.

Similarly, for the corresponding `Summary` object `summary`

- `summary.prefixes[summary.unifiedDepth]` is the `PrefixNode` of the greatest prefix of d that has the above property.
- `summary.prefixes[summary.unifiedDepth].column` is in `visited`, but not in `pending`.
- `summary.leftToUnify` is the number of $c \in C$ that are extensions of the node corresponding to `summary.prefix`, and for which `cnode.column` is not in `visited`.

Algorithm 2 ensures that this is all true initially, so our claim follows by simple induction. We need only observe that the inductive hypothesis ensures that Algorithm 4 correctly computes `summary.leftToUnify` by subtracting off those $c \in C$ that are no longer extensions of `summary.unifiedNode`, and have not been unified (i.e. are not in `visited`). Therefore, we have the following result.

Theorem 3. *Given (Σ, σ), implication can be decided in $O(|\Sigma| + |\sigma|)$ time.*

4.2 Axiomatization

Nonnull constraints and fictitious functional dependencies Given the chase algorithm, we can now extract an axiomatization using traditional techniques [1]. Our axiomatization requires that our axiom language be able to express two additional types of constraints on XML documents.

The first are *nonnull constraints*; intuitively, certain nodes in a tree may not be null if we know that certain other nodes are not null. For example, the root may never be null, and every nonnull node must have a nonnull parent. Formally, for $A, B \subseteq$ PATHS, we say that $nn(A, B)$ if for all matches of the smallest tree pattern for A and B, whenever the matches are not null on the paths in A, they are also not null on the paths in B.

The second constraint is a variation on the FD concept. Following Atzeni and Morfuni [3], we refer to these constraints as fictitious functional dependencies (FFDs).

Definition 2. *A* fictitious functional dependency $\sigma = A \overset{C}{\to} B$ *consists of three subsets $A, B, C \subseteq$* PATHS. *We let $\varphi(\sigma)$ be the smallest tree pattern in which all paths in A, B, and C occur, and let $=_{fd}$ be as in a normal XFD. The dependency holds if, for any two matches $\mu_1, \mu_2 \in \mathcal{M}_{\varphi(\sigma),T}$, whenever we have $\mu_1(p) =_{fd} \mu_2(p)$ for all $p \in A$, and $\mu_1(s) \neq null$ for all $s \in C$, then for all $q \in B$, $\mu_1(q) =_{fd} \mu_2(q)$.*

It is absolutely essential to note that the condition $\mu_1(p)$ and $\mu_2(p) \neq$ null for all $p \in A$, present for ordinary XFDs, is missing from the above definition. Thus the matches are now no longer required to be nonnull on A, only equal. For example, suppose we add an additional node v_{13} to the tree in Figure 3; we add this node as a child of v_2 and label it e (the label of v_7). Then this tree satisfies the XFD $\rho.a.b.f \rightarrow \rho.a.e$ because there is a unique occurrence of each path below v_1, but it does not satisfy the FFD $\rho.a.b.f \overset{\rho.a.b}{\rightarrow} \rho.a.e$.

The Axiomatization To axiomatize XFDs, we start with the axioms that Atzeni and Morfuni [3] introduced for relational FDs in the presence of null values. We let $X, Y, Z, W \subseteq$ PATHS. The following set of 12 axioms are sound and complete for relational FDs in the presence of nulls.

1. If $Y \subseteq X$, then $nn(X, Y)$.
2. If $nn(X, Y)$ then $nn(XZ, YZ)$.
3. If $nn(X, Y)$ and $nn(Y, Z)$ then $nn(X, Z)$
4. If $Y \subseteq X$, then $X \rightarrow Y$
5. If $X \rightarrow Y$, then $XZ \rightarrow YZ$
6. If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
7. If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
8. If $Y \subseteq X \subseteq Z$ then $X \overset{Z}{\rightarrow} Y$ (reflexivity for FFDs)
9. If $X \overset{Z}{\rightarrow} Y$ and $W \subseteq Z$, then $XW \overset{Z}{\rightarrow} YW$ (augmentation for FFDs)
10. If $X \overset{W}{\rightarrow} Y$ and $Y \overset{W}{\rightarrow} Z$ then $X \overset{W}{\rightarrow} Z$ (transitivity for FFDs)
11. If $X \rightarrow Y$ and $X \subseteq Z$ then $X \overset{Z}{\rightarrow} Y$ (FDs to FFDs)
12. If $X \overset{Z}{\rightarrow} p$ and $nn(Xp, Z)$ then $X \rightarrow p$ (FFDs to FDs)

In addition to these axioms, we add a new one one to change the intermediate set of an FFD.

13. If $X \overset{Z}{\rightarrow} Y$ and $nn(W, Z)$ then $X \overset{W}{\rightarrow} Y$

Furthemore, we need several axioms to capture the tree structure of an XML document. We let $p, q \in$ PATHS, $q \preceq p, x, y \in$ EL, $x \neq \alpha$.

14. $nn(\rho)$ (root is never null)
15. $X \rightarrow \rho$ (root is unique)
16. $nn(p, q)$ (if a node is not null, neither are any of its ancestors).
17. $p.x \rightarrow p$ (every non-attribute child has a unique parent).
18. $p \rightarrow p.\alpha$ (unique attribute child)

Finally, we need one axiom to capture the ability to "splice" paths together as we saw with the second chase rule illustrated in Figure 5. For this axiom, if $s \in$ PATHS, let $s.Y$ denote a set of paths all having s as a prefix.

19. If $X, q.Y \overset{Z}{\rightarrow} q.y.s$, q not a prefix of any path in X, where $X, q.Y \subseteq Z$ and $q.y.W \subseteq q.Y$ includes all paths in $q.Y$ with $q.y$ as a prefix, then $q, q.y.W \overset{Z}{\rightarrow} q.y.s$.

Theorem 4. *In the absence of a DTD, axioms 1-19 are sound and complete for XFD implication.*

Proof. We treat each direction separately.

Soundness: Soundess for Axioms 1-13 largely follows from the correctness of our mapping in Theorem 1 and from Atzeni and Morfuni [3]. However, unlike their definition for FFDs, we do not require that $A \subseteq C$ whenever $A \xrightarrow{C} B$ holds. However, this assumption is part of the axiom for Axioms 8 and 11, so we need only check Axioms 9, 10, 12, and 13 from the first two sets of axioms.

For Axiom 9, suppose $X \xrightarrow{Z} Y$ and $W \subseteq Z$ hold in T. As $W \subseteq Z$, both $X \xrightarrow{Z} Y$ and $XW \xrightarrow{Z} YW$ have the same tree pattern $\varphi(\sigma)$. Let $\mu_1, \mu_2 \in \mathcal{M}_{\varphi(\sigma),T}$ and suppose that $\mu_1(p) =_{fd} \mu_2(p)$ for all $p \in XW$, and $\mu_1(s) \neq$ null for all $s \in Z$. Then $\mu_1(p) =_{fd} \mu_2(p)$ for all $p \in X$ and so $\mu_1(q) =_{fd} \mu_1(q)$ for all $q \in Y$. As $\mu_1(p) =_{fd} \mu_2(p) \neq$ null for all $p \in W$ by assumption, $XW \xrightarrow{Z} YW$ holds in T.

For Axiom 10, suppose $X \xrightarrow{W} Y$ and $Y \xrightarrow{W} Z$ hold in T. Let $\varphi(\sigma)$ be the tree pattern for $X \xrightarrow{W} Z$, and let $\mu_1, \mu_2 \in \mathcal{M}_{\varphi(\sigma),T}$. We know from the proof of Theorem 1 that μ_1 and μ_2 each correspond to a row in $U(R(T))$. Let μ_1', μ_2' and μ_1'', μ_2'' be the matches that correspond to these two rows for the tree patterns for $X \xrightarrow{W} Y$, $Y \xrightarrow{W} Z$, respectively. As they are the same row in $U(R(T))$, we know that μ_i, μ_i', μ_i'' agree on all paths that they have in common. Suppose that $\mu_1(p) =_{fd} \mu_2(p)$ for all $p \in X$, and $\mu_1(s) \neq$ null for all $s \in W$. Then $\mu_1'(p) =_{fd} \mu_2'(p)$ for all $p \in X$, and $\mu_1'(s) \neq$ null for all $s \in W$, and so $\mu_1'(q) =_{fd} \mu_2'(q)$ for all $p \in Y$. That means $\mu_1''(q) =_{fd} \mu_2''(q)$ for all $p \in Y$. As $\mu_1''(s) = \mu_1(s) \neq$ null for all $s \in W$, we have $\mu_1(r) = \mu_1''(r) = \mu_2''(r) = \mu_2(r)$ for all $r \in Z$. Hence $X \xrightarrow{W} Z$ holds in T.

To check Axiom 12, suppose $X \xrightarrow{Z} p$ and $nn(Xp, Z)$ holds in T. Let $\mu_1, \mu_2 \in \mathcal{M}_{\varphi(\sigma),T}$, where $\mu_1(q), \mu_2(q) \neq$ null and $\mu_1(q) =_{fd} \mu_2(q)$ for all $q \in X$. To show $X \to p$ holds in T, we need only show that $\mu_1(p) =_{fd} \mu_2(p)$. If $\mu_1(p) = \mu_2(p) =$ null, we are done. So suppose without loss of generality that $\mu_1(p) \neq$ null. Therefore, as $nn(Xp, Z)$, $\mu_1(s) \neq$ null for all $s \in Z$. As $X \xrightarrow{Z} p$ holds, we are done.

Finally, for Axiom 13, suppose $X \xrightarrow{W} Y$ and $nn(W, Z)$ hold in T. Let $\varphi(\sigma)$ be the tree pattern for $X \xrightarrow{Z} Y$, and let $\mu_1, \mu_2 \in \mathcal{M}_{\varphi(\sigma),T}$. Similarly, let $\varphi(\sigma')$ be the tree pattern for $X \xrightarrow{W} Y$ and let $\varphi(\sigma'')$ be the tree pattern for the paths in XZ. We know from the proof of Theorem 1 that μ_1 and μ_2 each correspond to a row in $U(R(T))$. Let μ_1', μ_2' and μ_1'', μ_2'' be the matches that correspond to these two rows for $\varphi(\sigma')$ and $\varphi(\sigma'')$, respectively. As they are the same row in $U(R(T))$, we know that μ_i, μ_i', μ_i'' agree on all paths that they have in common. Suppose that $\mu_1(p) =_{fd} \mu_2(p)$ for all $p \in X$, and $\mu_1''(s) = \mu_1(s) \neq$ null for all $s \in W$. As $nn(W, Z)$ holds in T, $\mu_2'(r) = \mu_2''(r) \neq$ null for all $r \in Z$. Therefore, since $X \xrightarrow{W} Y$ holds and $\mu_i(p) = \mu_i'(p)$ for all $p \in X$, $\mu_1(q) = \mu_1'(q) =_{fd} \mu_2'(q) = \mu_2(q)$ for all $q \in Y$.

Of the remaining axioms, Axioms 14-18 follow from the tree structure of XML documents; this only leaves Axiom 19. Suppose for a contradiction we have a witness against $q, q.y.W \xrightarrow{Z} q.y.s$, but the premise of the last axiom holds. Then we have two matches μ_1, μ_2 for $Z \cup \{q.y.s\}$, agreeing on q and $q.y.W$ but disagreeing on $q.y.s$. Furthermore, at least one of them — say μ_1 — is completely nonnull. However, consider the following match μ_3. For all $t \in Z \setminus (q.y.W \cup \{q.y.s\})$ and prefixes of such paths,

$\mu_3(t) = \mu_1(t)$ (and is consequently nonnull). For all other paths t, $\mu_3(t) = \mu_2(t)$. We need only show that μ_1, μ_3 is a witness pair against $X, q.Y \xrightarrow{Z} q.y.s$.

By our selection μ_1 is completely nonnull on Z. Furthermore, μ_1 and μ_3 agree on X, since μ_3 took its values on X directly from μ_1. Furthermore, they agree on $q.Y$, because any match for a path in $q.Y$ in μ_3 is either taken from μ_1, or from μ_2 on a section where μ_1 and μ_2 are known to be equal. Finally, μ_1 and μ_3 disagree on $q.y.s$, so we do indeed have the required witness.

Completeness: For completeness, suppose $\Sigma \models \sigma$, where $\sigma = X \rightarrow Y$, $X, Y \subseteq$ PATHS. We show how to build a derivation for $\Sigma \vdash X \rightarrow p$ for each $p \in Y$, and then a series of applications of Axiom 6 completes the derivation to obtain a proof for $X \rightarrow Y$.

We fix one of the dependencies $X \rightarrow p$ which are of interest. Observe that, by Axioms 11 and 12, we have that $X \xrightarrow{X,p} p$ and $X \rightarrow p$ are equivalent, it suffices therefore to provide a derivation for $X \xrightarrow{X,p} p$. For any set $Y \subseteq$ PATHS, let $pref(Y)$ denote the set of all prefixes of paths in Y (with $Y \subseteq pref(Y)$, so that we include non-proper prefixes). Clearly we have $nn(Y, pref(Y))$, so by Axiom 13 a derivation for $X \xrightarrow{pref(X,p)} p$ is all that is required. For notational clarity, we use U to denote $pref(X, p)$.

We now show how to obtain a derivation for $\Sigma \vdash X \xrightarrow{U} q$. We know from the correctness of our chase algorithm that if $\Sigma \models X \rightarrow p$, there is a chase run on a tableau whose columns are exactly U and that terminates with the values in the p column unified. Fix this run; we make use of it to obtain our derivation.

Define the U-closure of X, X^U, as the set of all columns in the tableau that were unified by this chase run (so that $p \in X^U$ in particular). First we show that $\Sigma \vdash X \xrightarrow{U} X^U$ by induction on the steps of the chase run. Having established that, we complete the derivation as follows.

- $X \xrightarrow{U} X^U$ (derivation obtained in previous step)
- $X^U \xrightarrow{U} p$ (as $p \in X^U$ by definition of X^U, and using Axiom 8)
- $X \xrightarrow{U} p$ (Axiom 10).

We now construct a derivation for $\Sigma \vdash X \xrightarrow{U} X^U$. We define a chase step as the application of a single FD that results in column unification, and assume for simplicity that all dependencies in $\widehat{\Sigma}$ have been decomposed into XFDs with a unique path on the right-hand side (as complexity is not an issue here, this assumption is completely legal). Denote as $Cl_i(X)$ all the columns of the tableau that have been unified after step i; thus $Cl_0(X) = X \cup \{\rho\}$.

We now show by induction that for every i, $\Sigma \vdash X \xrightarrow{U} Cl_i(X)$. If $i = 0$ we have $X \xrightarrow{U} \rho$ by Axioms 15 and 11, and an application of Axiom 9 allows us to derive $X \xrightarrow{U} X \cup \{\rho\}$.

Now consider the inductive case of step i, where the one new unification was due to the application of some XFD $C \rightarrow d \in \widehat{\Sigma}$. By our inductive hypothesis, we have a derivation for $X \xrightarrow{U} Cl_{i-1}(X)$, and we know that $Cl_i(X) = Cl_{i-1}(X) \cup \{d\}$. The dependency $C \rightarrow d$ was applied either using the first or the second chase rule.

If the dependency was applied using the first chase rule, we use the following standard procedure to build our derivation.

- $Cl_{i-1}(X) \xrightarrow{U} C$ (definition of $Cl_{i-1}(X)$, the fact that we used rule 1 for the chase, so we must have $C \subseteq Cl_{i-1}(X)$, and Axiom 8)
- $C \rightarrow d$ (by assumption, this comes from $\widehat{\Sigma}$, so it is either in Σ or an instance of Axiom 17 or 18)
- $C \xrightarrow{U} d$ (Axiom 11)
- $Cl_{i-1}(X) \xrightarrow{U} d$ (Axiom 10)
- $Cl_{i-1}(X) \xrightarrow{U} Cl_i(X)$ (Axiom 9)
- $X \xrightarrow{U} Cl_{i-1}(X)$ (we have a derivation of this by inductive hypothesis)
- $X \xrightarrow{U} Cl_i(X)$ (Axiom 10)

If the dependency was applied using the second rule, we build the derivation as follows. Let q be the prefix involved in the unification, as per the second chase rule. We let D be C except for all paths having q as a prefix, thus $C = D \cup q.Y$ for some $q.Y$. We know d is $q.x.s$ for some $x \in$ EL and some s, perhaps empty. Let $q.x.W$ be the subset of paths in $q.Y$ that have $q.x$ as prefix. We build the derivation as follows.

- $Cl_{i-1}(X) \xrightarrow{U} q, q.x.W$ (definition of $Cl_{i-1}(X)$, the fact that we used the second chase rule, so we must have $(\{q\} \cup q.x.W) \subseteq Cl_{i-1}(X)$, and Axiom 8)
- $C \rightarrow d$ (by assumption, this comes from $\widehat{\Sigma}$, so it is either in Σ or an instance of Axiom 17 or 18)
- $C \xrightarrow{U} d$ (Axiom 11)
- $D, q.Y \xrightarrow{U} d$ (another way of writing the above)
- $q, q.x.W \xrightarrow{U} d$ (Axiom 19)
- $Cl_{i-1}(X) \xrightarrow{U} d$ (Axiom 10)
- $Cl_{i-1}(X) \xrightarrow{U} Cl_i(X)$ (Axiom 9)
- $X \xrightarrow{U} Cl_{i-1}(X)$ (we have a derivation of this by inductive hypothesis)
- $X \xrightarrow{U} Cl_i(X)$ (Axiom 10)

This completes the inductive case of the proof that gives the construction of the derivation for $\Sigma \vdash X \xrightarrow{U} X^U$, and also the entire completeness proof.

4.3 Reasoning about XFD implication with Horn clauses

One of the unsettling features of our axiomatization is that we had to add several new constraints to our language, making our axioms quite complicated. However, it turns out that derivations produced from these axioms can be encoded as logical inferences with Horn clauses. Thus they are closed under resolution, and so automated theorem provers can process them efficiently. Furthermore, we show that any interaction between the theory of trees and XFDs in this language is limited to the universal Horn fragment.

Our main theorem states that the XFD implication problem in the absence of a DTD is equivalent to an implication problem involving only Horn clauses in an appropriate signature. We observe that the electronic appendix to [2] also contains a use of Horn clauses to encode part of the XFD implication problem. However, our work differs from that encoding in the following ways.

- We do not deal with the complexity of inference; indeed, the full set of Horn clauses we generate has exponential size.
- We show how the *entire XFD implication problem* can be considered as an implication problem for a set of Horn clauses; in [2], only a computationally crucial subsection of the problem is encoded, and the relevant Horn clause problem used is not implication (the reduction given is to *satisfiability* of a set of Horn clauses).

We begin by introducing our coding of XFDs and nonnull constraints as Horn clauses. Here, we reason entirely within a flat relation R, using our standard mapping from XML FDs to relational ones. Consequently, we must begin by defining the schema for R. As observed in Section 3.2, we can define such a finite R easily from the instance of the implication problem we are given.

Having fixed R, we can express a (relational) functional dependency $X \rightarrow Y$ over R with a formula of the form

$$\forall_{i=1,2}\overline{x_i}\,\overline{y_i}\,\overline{z_i}\ \left(R(\overline{x_1},\overline{y_1},\overline{z_1}) \wedge R(\overline{x_2},\overline{y_2},\overline{z_2}) \bigwedge_{j=1,2} \overline{x_j} \neq \text{null} \wedge \overline{x_1} = \overline{x_2}\right) \rightarrow \overline{y_1} = \overline{y_2}$$

Note that we are using $\overline{x_i}, \overline{y_i}, \overline{z_i}$ to indicate that XFDs may involve arbitrary finite tuples, not just single variables. Also observe that writing $R(\overline{x_1},\overline{y_1},\overline{z_1})$ is notational shorthand; attributes within R are identified by their position, and the variables in the three tuples $\overline{x_1}, \overline{y_1}, \overline{z_1}$ may be arbitrarily "intermingled". Also, a statement such as $\overline{y_1} = \overline{y_2}$ is really shorthand for a conjunction of single-variable equality atoms, so the above technically represents a finite set of Horn clauses.

Observe that the formula above is almost a Horn clause. In fact, we can turn it into a real Horn clause by introducing a new unary relation symbol N, such that $N(x) \iff x \neq \text{null}$. The above now becomes

$$\forall_{i=1,2}\overline{x_i}\,\overline{y_i}\,\overline{z_i}\ \left(\bigwedge_{j=1,2} \big(R(\overline{x_j},\overline{y_j},\overline{z_j}) \wedge N(\overline{x_j})\big) \wedge \overline{x_1} = \overline{x_2}\right) \rightarrow \overline{y_1} = \overline{y_2}$$

We fix the signature $\{R, N\}$ for the remainder of the paper and do not mention it explicitly from now on.

This signature also lets us express nonnull constraints as Horn clauses. Relative constraints have the form

$$\forall \overline{x}\,\overline{y}\,\overline{a}\ (R(\overline{x},\overline{y},\overline{a}) \wedge N(\overline{x})) \rightarrow N(\overline{y})$$

For absolute constraints, such as "the root is never null", this becomes

$$\forall x, \overline{y}\ R(x,\overline{y}) \rightarrow N(x)$$

FFDs are also easy to encode. For an FFD $X \overset{Z}{\rightarrow} Y$ we have the encoding

$$\forall_{i=1,2}\overline{x_i}\,\overline{y_i}\,\overline{z_i}\ \left(\bigwedge_{j=1,2} R(\overline{x_j},\overline{y_j},\overline{z_j}) \wedge \overline{x_1} = \overline{x_2} \wedge N(\overline{z_2})\right) \rightarrow \overline{y_1} = \overline{y_2}$$

Note that the above is an example where all of X, Y, Z are disjoint,, and it is also clear that the encoding still works when this does not hold.

Given an XFD, FFD or nonnull constraint σ, we denote its Horn clause encoding over the signature $\{R, N\}$ as $H(\sigma)$. This gives us the following result.

Theorem 5. *Given an instance of an implication problem $\Sigma \models \sigma = A \rightarrow B$, $B = \{b_1, b_2, \cdots b_n\}$, we define a set Γ_σ of Horn clauses over $\{R, N\}$ to contain exactly the following.*

- *The Horn clause encodings of all dependencies in Σ,*
- *$\forall x\, \overline{y}\, R(x, \overline{y}) \rightarrow N(x)$ (i.e. the root is never null)*
- *$\forall_{i=1,2} x_i\, \overline{y_i} \left(\bigwedge_{j=1,2} \left(R(x_j, \overline{y_j}) \wedge N(\overline{y_j})\right) \wedge \overline{y_1} = \overline{y_2} \wedge N(x_1)\right) \rightarrow x_1 = x_2$ (i.e. the root is unique)*
- *For all attributes p, q of R such that $q = p.x$ for some $x \in \text{EL}$, $H(nn(q, p))$.*
- *For all attributes p, q of R such that $q = p.x$ for some $x \in \text{EL}, x \neq \alpha$, $H(q \rightarrow p)$*
- *$\forall p.\alpha$ attributes of R, $H(p \rightarrow p.\alpha)$*
- *For all $p.x \in paths(D)$ and in R, $x \in \text{EL}$, we add an extra constraint. Suppose for notational clarity that the attribute order of R is such that p is the first attribute, all the attributes that are extensions of $p.x$ are next, and all other attributes come last. Letting $\overline{px_1}$ and $\overline{px_2}$ be tuples of variables of arity equal to the number of extensions of $p.x$ in R, we have*

$$\forall_{i=1,2} p_i\, \overline{px_i}\, \overline{x_i} \left(\bigwedge_{j=1,2} R(p_j, \overline{px_j}, \overline{x_j}) \wedge p_1 = p_2 \right) \rightarrow R(p_1, \overline{px_2}, \overline{x_1})$$

Then $\Gamma \vdash H(A \overset{A,b_i}{\rightarrow} b_i)$ for all $b_i \in B$ if and only if $\Sigma \models A \rightarrow B$.

Observe that there is a slight technicality in the theorem statement above. Namely, reasoning with Horn clauses alone cannot get us all the way to $A \rightarrow B$ in general, only to $A \overset{A,b_i}{\rightarrow} b_i$ for all $b_i \in B$. However, by the soundness of our Axioms 11 and 12, these two sets of statements are logically equivalent, but our result is not enough to prove this equivalence using Horn clauses alone. Nevertheless, all the essential part of the reasoning can be performed with Horn clauses.

Proof. First, we must explain the last type of constraint we placed in Γ_σ; we call those *splicing constraints*. Intuitively, those constraints capture the fact that R is a flat relation produced by the unnesting of a tree. Suppose we have a node v that is a match for the path p and has two different matches below it for $p.x$. There are two different subtrees, rooted at $p.x$, below those matches. The splicing constraint tells us that there are two tuples involving v in R that are exactly the same except that one tuple contains the first match for $p.x$ and its corresponding subtree, and the other tuple contains the second match.

The forward direction is now clear, as all the clauses in Γ_σ are true in any tree in which Σ holds, and by Axioms 11 and 12 from our axiomatization, $\left\{ A \overset{A,b_i}{\rightarrow} b_i \mid b_i \in B \right\}$ and $A \rightarrow B$ are equivalent.

For the backward direction, we make use of our completeness proof for the axiomatization. We know that if $\Sigma \models \sigma$, then there is a proof of that fact using Axioms 1-19. Ideally, we would show that each of the axioms represents an inference that can be made using logic from premises that are only Horn clauses in Γ; in this case, any proof using the axioms would simply be "shorthand" for a Horn clause proof.

Indeed, it is not difficult to see that the above proposition holds for Axioms 1-10 and 13-18; we do not give the proofs for those axioms. The remaining Axioms are Axiom

11, 12 and 19. Our discussion must necessarily remain somewhat informal to maintain readability.

We first show Axiom 11 can be made as an inference with Horn clauses. Recall this axiom states that if $X \rightarrow Y$ and $X \subseteq Z$ then $X \overset{Z}{\rightarrow} Y$. Suppose we have $\overline{x_i}, \overline{y_i}, \overline{z_i}$, $i = 1, 2$, such that

$$\bigwedge_{j=1,2} R(\overline{x_j}, \overline{y_j}, \overline{z_j}) \wedge \overline{x_1} = \overline{x_2} \wedge N(z_1)$$

(i.e. the left-hand side of $H(X \overset{Z}{\rightarrow} Y)$). Then from the Horn clause encoding of Axiom 1, we have $N(\overline{x_1})$. As we have the premise $\overline{x_1} = \overline{x_2}$ we must have $N(\overline{x_2})$. However, we now have everything on the left-hand size of $H(X \rightarrow Y)$, so we can infer $\overline{y_1} = \overline{y_2}$, which is the right-hand size we wanted.

We now show the same for Axiom 19. Recall this states that if $X, q.Y \overset{Z}{\rightarrow} q.y.s$, q does not occur in X, we have $X, q.Y \subseteq Z$, and all paths in $q.y.W \subseteq q.Y$ end in α then $q, q.y.W \overset{Z}{\rightarrow} q.y.s$. There are two cases for applying this axiom; the first case is when $q.y.s$ is itself in $q.Y$, and the second is when it is not. For a particular derivation, we always know which case we are dealing with, so we can convert our applications separately. In the first case, the inference is an application of reflexivity and very simple. So we show only the second case.

Again, we start with an encoding of the left-hand side of what we are trying to derive. For clarity, we omit the attributes of R not involved in $X, q.Y, Z$ or $q.y.s$ as they are not relevant to our reasoning here. We reorder the remaining attributes of R so that they come in six categories of variables. In order these are as follows.

$\overline{a}$: The attributes of X
b: q
$\overline{c}$: $q.y.W$
$\overline{d}$: Attributes that are prefixes of a path in $\overline{c}$, extensions of $q.y$, and not already in $\overline{c}$
$\overline{e}$: All attributes of $q.Y$ not in $\overline{c}$ or $\overline{d}$, and also all the prefixes of such paths not mentioned in the previous categories
f: $q.y.s$ itself,
$\overline{g}$: Any prefixes of $q.y.s$ not mentioned in the previous categories
$\overline{h}$: Any paths in Z not mentioned in the previous categories

Observe that the attributes represented by $\overline{c}$ and $\overline{e}$ are a superset of those in $q.Y$.

Suppose now we have $\overline{a_i}, b_i, \overline{c_i}, \overline{d_i}, \overline{e_i}, f_i, \overline{g_i}, \overline{h_i}$ for $i = 1, 2$, such that

$$\bigwedge j = 1, 2R(\overline{a_j}, b_j, \overline{c_j}, \overline{d_j}, \overline{e_j}, f_j, \overline{g_j}, \overline{h_j})$$

Also suppose that $N(\overline{a_1}), N(b_1), N(\overline{c_1}), N(\overline{d_1}), N(\overline{e_1})$, and $N(\overline{h_1})$. Furthermore, suppose that$N(f_1)$ and $N(\overline{g_1})$ if and only if Z contains any of the f or g attributes. Finally, suppose that $b_1 = b_2$ and $\overline{c_1} = \overline{c_2}$. Given all this, we can apply an appropriate splicing constraint to conclude that

$$R(\overline{a_1}, b_1, \overline{c_2}, \overline{d_2}, \overline{e_1}, f_2, \overline{g_2}, \overline{h_1})$$

as the $\overline{c}$, $\overline{d}$, f and $\overline{g}$ variables represent exactly those attributes having $p.y$ as an extension, and those are the ones we are splicing.

Now, since we also know

$$R(\overline{a_1}, b_1, \overline{c_1}, \overline{d_1}, \overline{e_1}, f_1\overline{g_1}, \overline{h_1})$$

and the latter tuple satisfies the nonnull condition on all of Z, we are in a position to use our last premise, $\overline{c_1} = \overline{c_2}$. Observe that the a, c and e variables give us a superset of the paths in X and $q.Y$, so we can cover the left-hand side of the Horn encoding of the dependency $X, q.Y \overset{Z}{\rightarrow} q.y.s$. As the variables f represent $q.y.s$, $f_1 = f_2$ as required.

We now come to the last axiom we have left, which is Axiom 12. This axiom states that if $X \overset{Z}{\rightarrow} p$ and $nn(Xp, Z)$ then $X \rightarrow p$. However, it turns out that this inference requires a premise that is not in Γ_σ and — what is worse — cannot be expressed as a Horn clause, namely $\neg(N(x) \vee N(y)) \rightarrow x = y$ (note that this holds for single variables only, and is false for tuples in general).

To see this in detail, we begin our reasoning the usual way. The left-hand side of the conclusion states that we have $\overline{x_i}, p_i, \overline{z_i}$ $i = 1, 2$ such that

$$\bigwedge_{j=1,2} R(\overline{x_j}, p_j, \overline{z_j}) \wedge \overline{x_1} = \overline{x_2} \wedge N(\overline{x_1}) \wedge N(\overline{z_1})$$

We are seeking to derive $p_1 = p_2$. We know $N(p_1) \vee \neg(N(p_1))$ and the same for p_2. If we were allowed to use the premise $\neg(N(x) \vee N(y)) \rightarrow x = y$ we could conclude that we must have either $p_1 = p_2$, in which case we are done, or we have either $N(p_1)$ or $N(p_2)$. We know $N(\overline{x_1})$ and $\overline{x_1} = \overline{x_2}$, so $N(\overline{x_2})$. We can now use the Horn encoding of the nonnull constraint $nn(Xp, Z)$ and $(X \overset{Z}{\rightarrow} p)$ to obtain $p_1 = p_2$. However, if we are not allowed to use that one extra premise, we cannot make the inference. In particular, observe that we cannot conclude that $X \overset{X,p}{\rightarrow} p$ and $X \rightarrow p$ are equivalent, only that the latter implies the former.

However, this is not as serious a problem as one may think. We started out with the assumption that $\Sigma \models A \rightarrow B$, and from the completeness of our axioms we are free to work with the specific derivation that was produced in the completeness proof. In that particular style of derivation, observe that Axiom 12 is actually only used for one purpose, and this purpose is making the final transitions from $A \overset{A,b_i}{\rightarrow} b_i$ to $A \rightarrow b_i$ for each $b_i \in B$. The rest of the derivation makes no use of Axiom 12. Therefore it follows that our inability to express Axiom 12 with Horn clause inferences does not affect our ability to reason about everything else with Horn premises only.

Interaction of XFDs with the general theory of trees We now formalize the interaction between FDs and the theory of trees in this signature. Let Ψ_t denote all first-order formulae that hold on all trees (finite or infinite), $\mathcal{H}$ the set of all universally quantified Horn clauses, and $\mathcal{F}$ the set of all functional dependencies definable over the same signature. Finally, let $f(\sigma)$ be the set of FFDs that is directly equivalent to σ; in other words, if $\sigma = A \rightarrow B$ then $f(\sigma) = \left\{ A \overset{A,b_i}{\rightarrow} b_i \mid b_i \in B \right\}$. In this case above theorem has an immediate corollary.

Corollary 1. *For any $\Sigma \subseteq \mathcal{F}, \sigma \in \mathcal{F}$, $\Sigma \cup \Psi_t \vdash \sigma$ if and only if $\Sigma \cup (\Psi_t \cap \mathcal{H}) \vdash f(\sigma)$*

Intuitively, if we only wish to reason about implication of formulae in $\mathcal{F}$, we need not consider the entire first-order theory of trees, only its universal Horn fragment. Everything in Ψ_t which is not deducible from the Horn fragment alone is *completely independent* of the functional dependencies which may or may not hold on a tree.

Observe that the statement above can be read as a result about how the theory of trees and the theory of XFDs interact in a well-behaved way: any interaction relevant to XFDs can be guaranteed to occur within a small fragment of the theory of trees. Indeed, this fragment is as small as we could hope for, because we need our language to be able to express the XFDs themselves. We will see that we will be able to make corresponding statments in the presence of various classes of DTDs.

5 XFDs and Simple DTDs

5.1 Consistency

Simple DTDs are capable of introducing new XFDs into a document; however, they are not strong enough to assert that an XFD cannot hold. Thus there is no way way to write down a set of XFDs that would be inconsistent with a simple DTD.

Theorem 6. *If D is a simple DTD, then (Σ, D) is consistent for all Σ.*

Proof. Let T be the smallest tree (with respect to the number of nodes) such that $T \vDash D$, with a value assignment for the α nodes such that all values are unique in the entire tree. Observe that the minimality of T implies that T is finite, even if D is recursive; if we have a recursive D and no finite T satisfying it, D is inconsistent, and we ruled out inconsistent DTDs early on. We show that $T \vDash \sigma$ for *any* XFD σ. The result follows immediately, as T is a witness for consistency with any set Σ.

Observe that as D is simple, no path in T occurs more than once. The result we want follows immediately from the definition of XFDs and matchings; the existence of two different matchings for a path that could be a witness for $T \nvDash \sigma$ requires two different occurrences of a prefix of that path.

5.2 Trivial XFD implication

While it is the case that no set of XFDs can be inconsistent with a simple DTD, a DTD does exclude certain paths, and therefore can make certain XFDs hold vacuously. For, suppose we are given an implication problem instance (Σ, σ, D), and some path in $\varphi(\sigma)$ is not in paths(D). In this case $\Sigma \vDash_D \sigma$ trivially, because there is no tree satisfying D in which we can ever find a witness against σ, whether the tree satisfies Σ or not. We call such XFD implication instances *trivial*.

These cases of XFD implication are so pathological that we really want to remove them from consideration early. This allows us to produce a fairly concise set of axioms that is sound and complete for implication, on the assumption that reasoning is restricted to paths(D) and no trivial XFD implications occur.

Our decision to ignore these types of XFD implication is further bolstered by the fact that we can detect that $\Sigma \vDash \sigma$ via a trivial implication in $O(|\sigma| + |D|)$ time. Given a DTD, we construct a `Hashtable<Label,Set<Label>>` that encodes paths(D) in Algorithm 6. This algorithm clearly takes $O(|D|)$ time. The data structure `LabelPath` maps each label to the set of labels which are an allowable descendant. Given this data structure, the function IS_D_PATH in Algorithm 7 can determine whether $p \in$ paths(D) in $O(|p|)$ time. As a result, we ignore trivial XFDs for the rest of this section.

Algorithm 6 INITIALIZE_TABLE(D)

```
Require: $D$ is a simple DTD
Ensure: LabelTree encodes paths($D$)
1: LabelTable = new Hashtable<Label,Set<Label>>
2: Allocate a new Set<Label> and add it to LabelTable for key $\rho$.
3: for each $\ell \rightsquigarrow E \in D$ do
4:     if $\ell$ is not in LabelTable then
5:         Allocate a new Set<Label> and add it to LabelTable for key $\ell$.
6:     end if
7:     lSet = LabelTable.get($\ell$)
8:     for each alphabet symbol $a \in E$ (i.e. $a, a?, a^+$, or $a^* \in E$) do
9:         Add $a$ to lSet
10:    end for
11: end for
```

Algorithm 7 IS_D_PATH(p)

```
Require: LabelTable is initialized
Ensure: $p \in$ paths($D$) if and only if we return true.
1: let $\rho.p_1.p2.\ldots.p_k$ be the labels of $p$.
2: label = 1, set = LabelTable.get($\rho$)
3: while label $\leq k$ do
4:     if $p_{\texttt{label}}$ is not in set then
5:         return false
6:     end if
7:     set = LabelTable.get($p_{\texttt{label}}$)
8:     label = label+1
9: end while
10: return true
```

5.3 The chase algorithm

It is possible to modify our chase algorithm from Section 4.1 to include the presence of a simple DTD. For a problem instance (Σ, σ), we set up the tableau exactly as before, with a column for every path and prefix of a path in $\varphi(\sigma)$. However, in addition, we add a column for every path $p \in \text{paths}(\Sigma)$ such that D enforces $nn(\varphi(\sigma), p)$.

For example, suppose $\sigma = \{\rho.a, \rho.b.c\} \rightarrow \rho.b.d$ and $\Sigma = \{\rho.b.c.e \rightarrow \rho.b.d.f\}$. Also, suppose D contains the productions $c \rightarrow e^+$ and $d \rightarrow f$. Then $\mathcal{T}$ is as follows.

ρ	$\rho.a$	$\rho.b$	$\rho.b.c$	$\rho.b.c.e$	$\rho.b.d$	$\rho.b.d.f$
v_1	v_2	v_3	v_4	v_5	v_6	v_8
v_1	v_2	v_{10}	v_4	v_{11}	v_7	v_9

As before, we define the extension $\widehat{\Sigma}$ of Σ. However, we also add XFDs of the form $p.d_1 \rightarrow p.d_1.d_2$ for every column $p.d_1.d_2$ such that D contains a production of the form $d_1 \rightsquigarrow d_2\gamma$ or $d_1 \rightsquigarrow d_2?\gamma$, $d_2 \notin \gamma$. These additional XFDs represent the unique child constraints specified by D.

With this new tableau and $\widehat{\Sigma}$ we run the chase as before. Once again, the values in the b column are equal if and only if $\Sigma \vDash_D \sigma$.

Correctness of the Chase As before, when the chase terminates, we construct a tree T_{chase} from the tableau. Note that this tree may not satisfy the DTD D; however, we can extend T_{chase} to a document $\langle T_{chase} \rangle_D$ that does.

Lemma 2. *After the chase terminates, $\mathcal{T}$ corresponds to $U(R(T_{chase}))$ for some document T_{chase}. Furthermore, the tree T_{chase} can be extended to a (potentially infinite) tree $\langle T_{chase} \rangle_D$ which must satisfy both D and Σ. Furthermore, $\langle T_{chase} \rangle_D \models \sigma$ if and only if $T_{chase} \models \sigma$.*

Proof. We start by obtaining T_{chase} from $\mathcal{T}$ as in the case with no DTD. Then, we produce $\langle T_{chase} \rangle_D$, the Skolem hull of T_{chase} under D. That is, $\langle T_{chase} \rangle_D$ is a minimal tree $T \models D$ such that T_{chase} structurally embeds in T; note that as attributes require value assignments, this may not be unique. However, in general, we build the Skolem hull so that on any new attribute path, a fresh and unique attribute value is assigned.

Observe that we can always build the Skolem hull without violating D, because if T_{chase} should contain two occurrences of some path, they must be allowed by D. In particular, we included all relevant unique child dependencies in $\hat{\Sigma}$. And if T_{chase} contains only single occurrences of all its paths, then there is no other source of possible collision with D.

Recall that D is simple, and as such it cannot require more than one occurrence of a given path. Therefore, any nodes we added in creating $\langle T_{chase} \rangle_D$ must lie on paths that do not belong to Σ or σ, otherwise we would have added them to $\mathcal{T}$ earlier. Thus the addition of the new nodes does not have any impact on the satisfaction of Σ or σ.

This immediately gives us one direction of the correctness of our chase.

Theorem 7. *After the chase terminates, $\langle T_{chase} \rangle_D \models \sigma$ if and only if $\Sigma \models_D \sigma$.*

Proof. Clearly if $\langle T_{chase} \rangle_D$ does not satisfy σ, we have that $\Sigma \not\models \sigma$, as $\langle T_{chase} \rangle_D \models \Sigma$ by Lemma 2. For the other direction, the argument is similar to the case with no DTD. Suppose $\langle T_{chase} \rangle_D \models \sigma$. If in fact $\Sigma \not\models_D \sigma$, then there is some tree T' (infinite or finite) where Σ holds but σ does not. Find and extract any witness pair μ_1, μ_2 for this. Again, we know at least one of the matches is completely nonnull; suppose μ_1 is the nonnull match.

We now continue as for the proof of Theorem 2. We let $U(R(T'), \mu_1, \mu_2)$ be as before, except that we project out all columns that are not in the tableau. That is, the μ_i may be nonnull on some paths not in paths(Σ). By construction, none of these paths are references in $\widehat{\Sigma}$, and so they can be safely ignored throughout the chase. As before, we let $\mathcal{T}_0$ be the tableau at the start of the chase. We know that $\mathcal{T}_0$ covers $U(R(T'), \mu_1, \mu_2)$. We just need to show that at every step of the chase, $\mathcal{T}_n$ covers $U(R(T'), \mu_1, \mu_2)$.

Suppose we are stage n of the chase, and that $\mathcal{T}_n$ covers $U(R(T'), \mu_1, \mu_2)$. At this stage we attempt to unify with some $C \to d \in \widehat{\Sigma}$. If $C \to d$ was not introduced by D, we argue as before. So suppose we are chasing with an XFD $p.d_1 \to p.d_1.d_2$ given by D, where $p.d_1$, $p.d_1.d_2$ are columns in our tableau. As we are unifying with this XFD, $\mu_1(p.d_1) =_{fd} \mu_2(p.d_2) \neq$ null; in fact we know that they are the same node since $p.d_1$ is an internal node. The DTD must have a production of the form $d_1 \rightsquigarrow d_2\gamma$ or $d_1 \rightsquigarrow d_2?\gamma$, $d_2 \notin \gamma$ for us to have this XFD. Thus as $T' \models D$, $\mu_1(p.d_1.d_2)$ and $\mu_2(p.d_1.d_2)$ are the same node. Hence when we unify the $p.d_1.d_2$ column, $\mathcal{T}_{n+1}$ still covers $U(R(T'), \mu_1, \mu_2)$.

Complexity of the chase The only additional cost for running this chase comes from the additional number of columns and the additional XFDs for the DTD D. Each of these is bounded by $|\Sigma|$. Thus we have the following result.

Theorem 8. . *Given (Σ, σ, D) with D a simple DTD, implication can be decided in $O(|\Sigma| + |\sigma| + |D|)$ time.*

Proof. The primary difference between this theorem and Theorem 3 is that we have to add new columns and XFDs for the DTD D. The columns are the elements of `Tree` constructed in Algorithm 1. Instead of an array, which we allocate at the start, we use a growable array such as is typically used in hashtables. Note that adding a single new element is worst case $O(n)$, as we may need to increase the array capacity. However, as with hashtables, it is amortized $O(1)$ across the entire algorithm when we take into account the number of additions that were necessary to exceed the capacity of the array.

We initialize `Tree` as before. However, each `ReferenceTreeNode` has the additional field `uniqueChildren`, which is of type `Set<int>`. If the object represents a path p, then `uniqueChildren` is the set of integers for the paths $p.d$ such that the DTD added the XFD $p \to p.d$ to $\widehat{\Sigma}$. These sets will be empty by the end of Algorithm 6.

To initialize these sets, we need to keep track of the set of unique children for each production. We do that with the data structure `XFDTable` constructed by Algorithm 8. In addition construct a data structure `LabelTree` as in Algorithm 6, except that we restrict line 8 to only consider those a such that a or $a^+ \in E$. Note that for this modified version of `LabelTree`, Algorithm 7 returns `true` if and only if the DTD forces this path to be nonnull.

Algorithm 8 INITIALIZE_D_XFDS(D)

```
Require: $D$ is a simple DTD
Ensure: XFDTable encodes $\widehat{\Sigma} \setminus \Sigma$
 1: XFDTable = new Hashtable<Label,Set<Label>>
 2: VisitedTable = new Hashtable<Label,Set<Label>>
 3: Allocate a Set<Label> and add it to both XFDTable, VisitedTable for key $\rho$.
 4: for each $\ell \rightsquigarrow E \in D$ do
 5:     if $\ell$ is not in XFDTable then
 6:         Allocate a Set<Label> and add it to both XFDTable, VisitedTable for key $\ell$.
 7:     end if
 8:     lXFDSet = XFDTable.get($\ell$)
 9:     lVisited = VisitedTable.get($\ell$)
10:     for each $a$ or $a^+ \in E$ do
11:         if $a$ is in lVisited then
12:             Remove $a$ from lXFDSet {We have seen $a$ already; it cannot be a unique child}
13:         else
14:             Add $a$ to lXFDSet and lVisited
15:         end if
16:     end for
17: end for
```

We integrate our modified IS_D_PATH into the loop starting at line 9 in Algorithm 2, and the loop starting at line 4 in Algorithm 3 (i.e. they both have the same loop structure, so we combine the interiors). Suppose the checks at line 10 of Algorithm 2 or line 5 of Algorithm 3 fail, indicating that the path is not in the tableau. In this case, the modified `LabelTree` allows us to check in constant time whether the DTD forces the path to be not null. If so, we add a representative for this node to `Tree` as in Algorithm 1 and act as if the check did not fail.

Furthermore, just before the end of the loops at line 17 in Algorithm 2 and 16 in Algorithm 3, we check if $p_{\texttt{label}-1}$ (for `label` $\neq 1$) is in `XFDTable`. If so, we check whether $p_{\texttt{label}}$ is in the set `XFDTable.get`($p_{\texttt{label}-1}$). A positive result means that $p_{\texttt{label}}$ is a unique child of $p_{\texttt{label}-1}$, so we add the `column` for this path to the set `uniqueChildren` in the `ReferenceTreeNode` for $p_{\texttt{label}-1}$. This corresponds to adding an XFD of type $p.d_1 \to p.d_1.d_2$ generated by the DTD.

When we are done with our modified version of Algorithm 2, `Tree` will contain all the columns necessary for our modified chase. Furthermore, the fields `uniqueChildren` will encode all of the extra XFDs that were added by D. Finally, it is clear from our modifications that this algorithm still takes $O(|\Sigma|)$ time.

We now run Algorithm 5 as before with only one modification. After line 41, we loop through all of the elements of `Tree[next].uniqueChildren`. For each one that is not in `visited`, we add it to `visited` and `pending`; this corresponds to a unification for that unique child XFD. It is clear that this correctly implements the chase. As each of these elements corresponds to a unique column in our tableau, the `uniqueChildren` sets have a total of $O(|\sigma| + |\Sigma|)$ elements for all of the various `ReferenceTreeNode` objects combined. Therefore, this algorithm still runs in $O(|\sigma| + |\Sigma|)$ time.

5.4 Axiomatization

To axiomatize XFD implication for simple DTDs, we need only account for the additional XFDs that we added to $\widehat{\Sigma}$ in the chase. However, instead of an additional axiom, we add an *axiom schema*. That is, the actual axioms depend on our DTD D, but we have a single schema for specifying these axioms from the DTD.

Theorem 9. *In the presence of a simple DTD D, a sound and complete set of axioms for (unrestricted) XFD implication includes exactly Axioms 1-19 from Section 4.2 with*

20. $nn(p.x, p.x.y)$, *for each production* $x \rightsquigarrow y\gamma$, $x \rightsquigarrow y^{+}\gamma \in D$
    (i.e. the DTD does not allow certain children to be null).
21. $p.x \to p.x.y$, *for each production* $x \rightsquigarrow y\gamma$, $x \rightsquigarrow y?\gamma \in D$, *where* $\gamma \in (\text{EL} - \{y\})^*$
    (i.e. XFDs enforced by the DTD).

Proof. Soundness of the new axioms is clear, and the completeness proof is almost identical to that in the case with no DTD. The only difference is that the chase now proves $A \xrightarrow{Z} b$, where Z is the closure under all the nonnull constraints of all the paths in A, b and their prefixes. Z is also exactly the set of the paths that have designated columns in the chase tableau.

5.5 Reasoning about XFD implication with Horn clauses

In this section, we show again that our axioms can be encoded as Horn clauses, and that the interaction of simple DTDs with XFDs is very well-behaved, according to our formalization of that concept. More precisely, in the presence of a simple DTD, we still do not have to step outside the language of universal Horn clauses to carry out our reasoning.

Theorem 10. *Given a simple DTD D and an instance of an implication problem $\Sigma \models_D \sigma$, suppose we define a set $\Gamma_{\sigma,D}$ of Horn clauses over $\{R, N\}$ to contain exactly the following.*

- *All of Γ_σ as defined in Section 4.3.*
- *$H(nn(p.x, p.x.y))$, for all $p.x$ in paths(D), $x \in$ EL such that D contains a production $x \rightsquigarrow y\gamma$ or $x \rightarrow y^+\gamma$, $\gamma \in$ EL*, and both $p.x$ and $p.x.y$ are attributes of R (i.e. the DTD requires the children to be nonnull).*
- *$H(p.x \rightarrow p.x.y)$, for all $p.x \in$ paths(D) such that D contains a production $x \rightsquigarrow y\gamma$ or $x \rightarrow y?\gamma$, $\gamma \in$ EL*, and both $p.x$ and $p.x.y$ are attributes of R (i.e. unique child constraints imposed by the DTD).*

Then $\Gamma_D \vdash H(A \overset{A,b_i}{\rightarrow} b_i)$ for all $b_i \in B$ if and only if $\Sigma \models_D A \rightarrow B$.

Proof. The proof is exactly the same as in the Section 4.3, as the only new constraints induced by the DTD are new XFDs and nonnull constraints. Furthermore, the new axioms are only sets of those FDs and nonnull constraints, rather than new inference rules.

Interaction of XFDs with the general theory of trees We can formulate the appropriate analogue of Corollary 1 for the theorem above. Given a simple DTD D, let Ψ_D denote all first-order formulae that hold on all trees (finite or infinite) satisfying D, and let $\mathcal{H}$ and $\mathcal{F}$ be as before. Then, we have the following result.

Corollary 2. *For any $\Sigma \subseteq \mathcal{F}, \sigma \in \mathcal{F}$, $\Sigma \cup \Psi_D \vdash \sigma$ if and only if $\Sigma \cup (\Psi_D \cap \mathcal{H}) \vdash \sigma$*

Intuitively, the only part of the theory of simple DTDs that interacts in any way with FDs is the universal Horn fragment. This explains why simple DTDs have proven a relatively tractable class to work with. Indeed, we conjecture that they are the largest class of DTDs for which we are able to encode our axioms with universal Horn theory alone.

6 XFDs and #-DTDs

6.1 Consistency

In contrast to simple DTDs, it is possible to give a Σ and an #-DTD D such that (Σ, D) is inconsistent. An example is the DTD consisting of the productions $\rho \rightsquigarrow abb, a \rightsquigarrow \alpha, b \rightsquigarrow \alpha$, and the set of XFDs $\{\rho.b.\alpha \rightarrow \rho.b, \rho.a.\alpha \rightarrow \rho.b.\alpha\}$. The tree has two paths $\rho.b.\alpha$. The first XFD forces the two attribute values on those paths to be different, but the second forces them to be the same.

Fortunately, consistency for a set of XFDs Σ and an #-DTD D can be decided in polynomial time. For the sake of clarity, we start by giving a naïve exponential algorithm, and then show how to bring down the complexity. We first start with the following lemma.

Lemma 3. *Let T_D be the smallest tree such that $T_D \models D$. (Σ, D) is consistent if and only if there is a way of assigning attribute values to any α-nodes in T_D such that $T_D \models \Sigma$.*

The lemma essentially states that issues of consistency can be decided on the smallest tree satisfying D. Our naïve algorithm involves constructing this tree and assignment of values to α-nodes; our polynomial algorithm demonstrates that the entire tree T_D need not be constructed explicitly. The proof of the lemma requires us to introduce a useful technical concept — *uniformly matched trees*.

Definition 3. *A* uniformly matched *tree is any tree T such that for every rooted path p in T, either:*
- *$\forall \mu$ which are matches of $paths(T)$, $\mu(p) =$ null, or*
- *$\forall \mu$ which are matches of $paths(T)$, $\mu(p) \neq$ null*

Uniformly matched trees have several interesting properties.

Lemma 4. *If a tree T is uniformly matched, then $U(R(T))$ contains no null values.*

Proof. This is clear from the definition. If for some p, $U(R(T))$ contains a null value, then T contains a null match for p. Thus T contains no nonnull matches for p at all, and the p column should never have been included in the schema for $U(R(T))$ when the relation was constructed.

Lemma 5. *If T is uniformly matched, $\sigma = A \rightarrow B$, and $T \not\models \sigma$, it is not possible to extend T by adding a finite number of nodes to obtain a tree T' such that $T' \models \sigma$.*

Proof. In T above, we have at least two matches that witness $T \not\models \sigma$. These correspond to two tuples in $U(R(T)))$. By Lemma 4, these tuples contain no null values. Thus these witness tuples cannot be destroyed by adding extra nodes.

Lemma 6. *Given any #-DTD, the minimal tree T satisfying D is uniformly matched.*

Proof. Suppose not; then some path p, has a nonnull match and a null match in T. But this contradicts minimality, as a minimal tree for D would contain only nodes that are constrained to exist by the existence of their parents. In our case, the node which is the nonnull match for p can be removed to create a smaller T', $T' \models D$.

We can now return to the proof of the main lemma.

Proof (Lemma 3). One direction of the lemma is clearly trivial: if T_D, with an appropriate set of attribute values if applicable, satisfies Σ, then (Σ, D) is consistent. We argue the other direction. Suppose that there is no way to assign attribute values to T_D such that $T_D \models \Sigma$. However, observe that T_D must embed structurally into every other T' such that $T' \models D$. As T_D is uniformly matched, no further addition of nodes makes Σ satisfied. Thus Σ is not satisfied by any T' allowed by D, so (Σ, D) is indeed inconsistent.

In model theoretic terms, the T_D is the "prime model" of D, in that it embeds in all other documents satisfying D. We now state our naïve consistency checking algorithm that runs on T_D.

Naïve algorithm

1. Construct the minimal T_D from D. Initialize all the attribute nodes to different values from VAL, and convert T_D to the flat relation $U(R(T_D))$.
2. Partition Σ into Σ_a, which contains all the XFDs with an attribute node on the right-hand side, and $\Sigma_i = \Sigma \setminus \Sigma_a$.
3. Treat $U(R(T_D))$ as a very large chase tableau, and perform a standard relational chase (no use of the second rule) on it with Σ. Any unification from Σ_a is allowed to proceed normally, but any attempt at a unification from Σ_i causes the algorithm to return "no".
4. If the chase completes successfully, return "yes".

Theorem 11. *There is an assignment of attribute values to attribute nodes of T_D under which $T_D \models \Sigma$ if and only if the algorithm returns "yes".*

Proof. One direction is clear: if the algorithm returns "yes", our chase on $U(R(T_D))$ terminated with a tableau satisfying all of Σ. The chase performed no unifications on non-attribute columns, so it did not change the structure of the tree represented by the tableau. Thus the tableau can be transformed back to T_D with an attribute value assignment that satisfies Σ.

On the other hand, suppose the algorithm returns "no". Suppose for a contradiction that there is an assignment of attribute values to attribute nodes of T_D such that $T_D \models \Sigma$. Thus there is an assignment of attribute values to the attribute columns of $U(R(T_D))$ such that $U(R(T_D))$ satisfies Σ. However, the algorithm attempted some unification with $\sigma_i \in \Sigma_i$, $\sigma_i = A_i \rightarrow b_i$, and returned "no". Focus on the two tuples t_1, t_2 where this happened. There are two possible cases.

Case 1: t_1 and t_2 correspond to two matches for $\varphi(\sigma)$ that agree by identity on all the nodes in A_i but reach different b_i nodes. It is clear that no value assignment to attribute nodes can make such a σ_i satisfied, so we have our contradiction.

Case 2: t_1 and t_2 correspond to two matches that agree on some $a_i \in A_i$ by value only (not by identity). This means that the two a_i values of t_1 and t_2 have been unified by the chase (as they started out distinct). However, it is easily shown by induction on the order in which the chase performs unifications, and using the fact that the initial attribute values are all distinct, that every unification performed by the chase is necessary to the satisfaction of some $\sigma_a \in \Sigma_a$. Intuitively, this is true because every unification destroys some witness to σ_a not holding; if the unification is never performed, the witness is retained. Since each unification was necessary for the satisfiaction of some σ_a, any assignment satisfying our σ_i on t_1 and t_2 would violate that σ_a (possibly on some different tuples). We have a contradiction again.

This algorithm is possibly exponential because of the duplicate paths that may appear in T_D, resulting in a large $U(R(T_D))$. However, using special encoding techniques to compress duplicate paths, we can implement this algorithm in polynomial time.

To present our efficient algorithm, we begin by introducing a slightly different version of the naïve algorithm, which is completely equivalent to the previous one but easier to reason about. This version artificially divides the chase on $U(R(T_D))$ into three phases.

Three-phase naïve algorithm

1. Construct the minimal T_D from D. Initialize all the attribute nodes to different values from VAL, and convert T_D to the flat relation $U(R(T_D))$.
2. Partition Σ into Σ_a, which contains all the XFDs with an attribute node on the right-hand side, and $\Sigma_i = \Sigma \setminus \Sigma_a$.
3. Treat $U(R(T_D))$ as a very large chase tableau, and perform a standard relational chase (no use of the second rule) on it with Σ. The chase proceeds in three phases, as follows.
   (a) Check whether each dependency in Σ_i is satisfied. If not, return "no", otherwise continue.
   (b) Chase in the standard manner only with Σ_a, performing normal unifications.
   (c) Again check whether each dependency in Σ_i is satisfied. If not, return "no", otherwise return "yes".

As the following lemma shows, these algorithms are the same.

Lemma 7. *Given the same input, the three-phase version of the naïve algorithm returns "yes" if and only if the one-phase version did. Moreover, if both algorithms succeed, the final tableau produced by both algorithms is the same.*

Proof. As the chase is Church-Rosser, this is intuitively clear; we are free to reorder the chase steps as long as we allow all dependencies to fire as many times as needed. That is, if firing some σ can affect the satisfaction of some σ' that fired previously, σ' must have a chance to fire again. We also give a more formal proof.

Suppose the three-phase version returns "no". Then some dependency $\sigma_i = A_i \to b_i$, $\sigma_i \in \Sigma_i$ was violated in Phase 1 or 3. If it was violated in Phase 1, we have a situation where the two witnesses to the violation agree by identity on A_i but not on b_i; no unifications of attribute nodes can make this σ_i satisfied. Thus, when the one-phase algorithm checks the satisfaction of σ_i — which it must do at some point — it will return "no" as well. On the other hand, if σ_i was violated in Phase 3, we have a situation where some number of unifications with Σ_a produced two witnesses against σ_i. However, all unifications performed in Phase 2 will be performed by the one-phase algorithm eventually, unless the one-phase algorithm returns "no" before this has a chance to happen. Thus the same witnesses against σ_i must come into existence at some point in the one-phase algorithm. Because such witnesses can never be destroyed (i.e. unification of internal node values is not allowed), the one-phase algorithm will return "no" when it checks this σ_i and finds the same witness.

On the other hand suppose the one-phase version returns "no" due to the existence of a pair of tuples t_1, t_2, which witness that $\sigma_i = A_i \to b_i$ does not hold. If the tuples agree by identity on all of A_i, the one-phase algorithm would have found this witness in Phase 1. If they do not agree by identity, the witness was created by some number of unifications with Σ_a. Those same unifications will be performed in Phase 2 of the three-phase algorithm, and the same witness will be produced, leading to a "no" answer in Phase 3 when σ_i is checked.

To see that the algorithms produce the same tableau when they succed, note that success means the same thing for both algorithms: during a chase with Σ, nothing in

Σ_i ever fired and all of Σ_a fired until it was satisfied. Thus the chase that was actually performed was really a chase with Σ_a, in both cases. The fact that both resulting tableaux are the same is now immediate.

Efficient algorithm The intuition behind turning the three-phase naïve algorithm into an efficient one is to note that we do not need to keep track of the value assignment in the entire tree; in fact, the crux of the satisfiability problem for Σ occurs in Phase 2, and has to do with the classes of attribute nodes whose values are unified by the application of dependencies in Σ_a. It turns out this is really all the information we need.

The data structure used in the algorithm is an *annotated tree*, with several (directed) back edges exiting certain attribute nodes. We begin by setting up an ordinary tree which has one occurrence of every path that satisfies the two following conditions.

1. It occurs in some $\sigma \in \Sigma$.
2. It is required to be nonnull by the DTD.

Annotate each edge of this tree with an integer corresponding to the appropriate production in the DTD. If there is an edge from parent a to child b, and the DTD requires a minimum of n b-children for every a-node, annotate the edge with n. For example, if the relevant DTD production is $a \rightsquigarrow bbbbc^*$, then $n = 4$. In addition, we insert several directed back edges. For every p a path that is the right-hand side of some XFD in Σ_a, let v be the match for p; add a back edge from v to itself. We note that the target of the back edge will change throughout the algorithm.

This initial annotated tree T_A is a concise encoding of the minimal tree T_D featured in the previous algorithm, or more precisely the part of the tree that is affected by Σ. Our efficient algorithm is a fairly direct encoding of our three-phase naïve algorithm, where the encoding allows the work to be carried out on T_A.

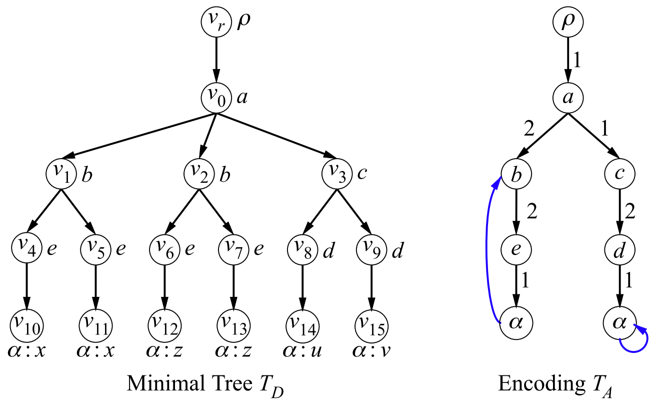


Fig. 6. A Uniformly Matched $T_D \models D$ and its Annotated Encoding T_A

Initially, T_A is to be understood as a tree where all the attribute nodes have different values. Later, some of the values will be unified, and the back edges will be used to keep track of this. Every attribute path p where some values have been unified will have, in the encoded tree, a back edge pointing to some ancestor of the match for p. That ancestor represents the scope of the unification. That is, if the target of the back edge is itself a match for some path q, then the values of p are understood to be the same only within every subtree rooted at q. Note however, that they will differ between different subtrees rooted at q. An example of a uniformly matched T_D part way through the run of our algorithm, as well as its annotated encoding T_A, is illustrated in Figure 6.

The construction of this tree takes $O(|D| + |\Sigma|)$ time. We modify Algorithm 8 so that `XFDTable` keeps track of not just of the unique children, but those children that must have a nonzero occurrence. Furthermore, instead of a set of labels, each `XFDTable.get`(p) is a hashtable mapping labels to integers, representing the number of occurrences of that label. For example, suppose we have the production $a \rightsquigarrow bbbbc^*d$. Then `XFDTable.get`$(a)$ maps b to 4 and d to 1, but c is not present in this hashtable at all. From this new construction of `XFDTable`, we can construct T_A by starting out with a `Tree` containing only a single `ReferenceTreeNode` for the root and using the modified version of Algorithm 2 for Theorem 8.

We now give the encoding of the chase Phases 1, 2, and 3 in our naïve algorithm.

Phase 1. This is where we check for each $\sigma_i \in \Sigma_i$ whether $T_D \models \sigma_i$. We proceed as follows. Given $\sigma_i = A \rightarrow b$, check whether there is a match for all of $\varphi(\sigma_i)$ in the annotated tree T_A. If not, we know $T_D \models \sigma_i$ trivially. Otherwise, identify p, the longest common prefix of b and any path in A. If $q = b - p$ is null, that is $b = p$, σ_i is satisfied, because all extensions of b in A are either internal nodes or have unique attribute values at this point. So suppose q is not null. Calculate the product of all the annotations along q: if this is equal to 1, answer "yes", otherwise answer "no".

Lemma 8. *T_A with the initial value assignment satisfies σ_i if and only if the above encoding returns "yes".*

Proof. Observe that, due to the way T_A is constructed, the product of the labels along q indicates the exact number of distinct matches for b that correspond to the same match for A. Thus, if that number is not equal to 1, we see immediately that $T_D \not\models \sigma_i$, and otherwise $T_D \models \sigma_i$ as all attribute values are distinct, and so we cannot have two matches for $\varphi(\sigma_i)$ agreeing on A by value but not by identity.

The complexity of this step is $O(|\Sigma_i|)$.

Phase 2. This is where we unify certain classes of attribute nodes, and our corresponding action on T_A is to raise certain back edges. We start by removing from consideration all dependencies $\sigma_a \in \Sigma_a$ such that T_A does not contain a full match for $\varphi(\sigma_a)$, as those are satisfied. Now, iterate the following process until no more changes in T_A can be made.

Choose a dependency $\sigma_a \in \Sigma_a, \sigma_a = A \rightarrow b$. Identify a split of b into p and q as before; by assumption b ends in an attribute node, so q is certainly not null unless $b \in A$ and the dependency is trivial. Now, we may or may not raise the back edge exiting b further towards the root, according to the following procedure.

– (Type 1 raising) If the back edge exiting b is a self-loop, and the product of the annotations along q is greater than 1, raise the back edge to p. Whether or not a change was made here, continue to step 2.

– (Type 2 raising) Let v be the node currently reached by the back edge that exits b, and let r be the path from the root to v. Identify t, the shortest rooted path that is a prefix of r and such that all the paths in A having t as a prefix end in attribute nodes and have back edges that hit either ancestors of t, or t itself. If there is no such t or $t = r$, do nothing. Otherwise, check whether the product of the annotations on the subpath *between* the relevant matches for t and r is strictly greater than 1. If so, raise the back edge exiting b to t, otherwise raise no edges.

Lemma 9. *The above process terminates in polynomial time; when it does terminate, the tree T_D obtained from expanding the encoding in T_A is such that $T_D \models \Sigma_a$. Moreover, every single back edge raising is necessary — no tree with a back edge pointing lower than those in T_A at the end of the above process satisfies Σ_a.*

Proof. With regards to termination, observe that there are at most $|\Sigma_a|$ back edges. Every back edge can only be raised to the root, and the process must terminate at that time. For the complexity, first notice that product of the edges between any two nodes remains the same for the whole algorithm. For each node v and t, we can store the product of these edge annotations in the `ReferenceTreeNode` for v. We just add a field `edgeProduct` which is a `Hashtable<int,int>` mapping the column of a prefix of v to the product of the annotations from the end of this prefix to v. Note that this addition increases the complexity of initializing `Tree` from $O(|D| + |\Sigma_a|)$ to $O(|D| + |\Sigma_a|^2)$ time. However, it does keep us from having to compute the value at each edge raising, which would be even more expensive.

For the edge raising, note that a Type 1 raising can happen at most once for each XFD in Σ_a. So we need only keep track of the possible Type 2 raisings. To do this, for each $A \to b \in \Sigma_a$, we build a data structure like we did in Algorithm 2 in the proof of Theorem 3. We start with a new summary structure.

```
struct Summary {
    PrefixNode[1:$k$] prefixes;
    int bColumn;
    int backEdge;
    int pDepth;
    int tDepth;
}
```

The fields `prefixes` and `bColumn` are just as in the proof of Theorem 3. This summary structure corresponds to an XFD $A \to b$ where b is an attribute path. Therefore b has a back edge; the field `backEdge` is the depth of the node to which this back edge points. The value `pDepth` is the depth of the prefix $p \preceq b$ in the algorithm above. Similarly, `tDepth` keeps track of the depth of the shortest prefix $t \prec b$ such that

– All paths $a \in A$ extending t are attribute paths.
– All of these paths have back edges above or equal to t.

Notice that this is similar to our value of t in the Type 2 raising, though we do not require that it is a prefix of v. This allows us to quickly perform a Type 2 raising when necessary.

As before, for each XFD $A \to b$, a `PrefixNode` represents a prefix of b.

Algorithm 9 INITIALIZE_XFD_EDGES($A \rightarrow b$)

```
Require: $A \rightarrow b$ is an XFD.
Ensure: AttList contains the set of ANode* in $A \rightarrow b$ that depend on each attribute.
 1: summary = INITIALIZE_EDGE_SUMMARY($b$)
 2: if summary == null then
 3:     return
 4: end if
 5: Add summary to XFDMap.get(summary.bColumn)
 6:
 7: allocated = new List<ANode>, maxagree = 0
 8: for each $a \in A$ do
 9:     let $\rho.a_1.a_2.\dots a_m$ be the labels of $a$.
10:     label = 1, position = 0, agree = 0
11:     while label $\leq m$ do
12:         if $a_{\texttt{label}}$ is in Tree[position].children then
13:             tableau = Tree[position].children.get($a_{\texttt{label}}$)
14:             agree = ($a_{\texttt{label}}$ == $b_{\texttt{label}}$ ? agree+1 : agree)
15:             maxagree = MAX(agree,maxagree)
16:             label = label+1, position = tableau.column
17:         else
18:             return
19:         end if
20:     end while
21:     if $a_m = \alpha$ then
22:         node = new ANode
23:         node.prefix = &summary.prefixes[agree]
24:         node.column = position
25:         node.backEdge = $m$
26:     end if{Let all the prefixes know they have a back edge}
27:     for position = agree to 0 do
28:         if $a_m = \alpha$ then
29:             summary.prefixes[position].backBelow++,
                summary.prefixes[position].attribOnly = true
30:         else
31:             summary.prefixes[position].attribOnly = false
32:         end if
33:     end for
34:     {This element of $A$ forces our $t$ node to be lower}
35:     if summary.tDepth $\leq$ node.prefix.depth then
36:         summary.tDepth = node.prefix.depth+1
37:     end if
38:     Add node to allocated
39: end for
40: summary.pDepth = maxagree {$p$ is maximal common prefix with some path}
41: for each node in allocated do
42:     Add node to AttList[node.column]
43: end for
```

Algorithm 10 INITIALIZE_EDGE_SUMMARY(b)

```
Ensure: Returns a Summary structure for b.
1:  let ρ.b_1.b_2. ... b_k be the labels of b.
2:  summary = new Summary, summary.prefixes = new PrefixNode[1:k]
3:  label = 1, position = 0
4:  while label ≤ k do
5:    if d_label is in Tree[position].children then
6:      tableau = Tree[tableauPos].children.get(b_label)
7:      prefixes[label] = new PrefixNode
8:      prefixes[label].summary = &summary
9:      prefixes[label].column = tableau.column
10:     prefixes[label].depth = label
11:     prefixes[label].attribOnly = true
12:     prefixes[label].backBelow = 0
13:     label = label+1, position = tableau.column
14:   else
15:     return null
16:   end if
17: end while
18: summary.backEdge = k, summary.tDepth = 0
19: summary.bColumn = position
20: return summary
```

```
struct PrefixNode {
    Summary* summary;
    int column;
    int depth;
    int backBelow;
    boolean attribOnly;
}
```

Again each `PrefixNode` corresponds to a single XFD and so has a single `Summary` structure. The values `column` and `depth` are also as before. The value `attribOnly` indicates that there only attribute paths in A beneath this prefix, and `backBelow` keeps track of how many of them still have back edges beneath this node.

Finally, for each XFD $A \to b$, an `ANode` represents an attribute path $a \in A$.

```
struct ANode {
    PrefixNode* prefix;
    int column;
    int backEdge;
};
```

The values `prefix` and `column` are the same as in `CNode` in Algorithm 2. As this `ANode` represents an attribute path, it has a back edge; the field `backEdge` is the depth of the node to which this back edge points.

As in the proof of Theorem 3, we let n be the size of `Tree`, and allocate an array `AttList[1:`n`]`. Each element of AttList is the set the set of `ANode` objects that represent this attribute in the tableau. In addition, we have a `XFDMap`, which is a `Hashtable<int,Set<Summary>>`. For each path b, if i is the column for b in

Tree, then XFDMap maps i to the set of XFDs $A \rightarrow b \in \Sigma_a$ that have b on the right-hand side. All of these sets start out empty; we initialize them in Algorithm 9. The subroutine INITIALIZE_EDGE_SUMMARY is shown in Algorithm 10.

As usual, it is clear that Algorithm 9 intializes our structures properly. It also runs in $O(|\Sigma_a|)$ time. For each $A \rightarrow b \in \Sigma_a$ and $a \in A$, it performs a linear scan down a once to find its position in Tree and is greatest common prefix with b. It then performs another linear scan back up a to set the backBelow and attribOnly fields properly in each PrefixNode.

Given these initialized data structures, Algorithm 11 represents out edge-raising algorithm in the following way. For each XFD $A \rightarrow b \in \Sigma_a$ we have an associated Summary structure. The *candidate t value* is the prefix of b whose depth is summary.tDepth. Whenever we raise an edge, either by a Type 1 or a Type 2 raising, we have to recompute these candidate t value for all of the XFDs $C \rightarrow d$ such that $b \in C$. If the candidate t value moves, then the XFD $C \rightarrow d$ is a candidate for a new Type 2 raising; in this case we add it to the queue, provided that it is not there already. It is clear that, after the first round of Type 1 raisings guaranteed by our initialization at line 2, the only way that we would raise the back edge of an XFD a second time is if the value of the candidate t changed. Therefore Algorithm 11 correctly implements our edge-raising algorithm.

Like the previous two algorithms, Algorithm 11 is $O(|\Sigma_a|^2)$. Consider each XFD $A \rightarrow b$ for which we raise the back edge. If we raise raise the back edge k steps up, then for each XFD $C \rightarrow d$ with $b \in C$, we take $O(k)$ steps to decrement the backBelow counters, and possibly decrement tDepth in the Summary structure. However, we can do this at most $|b|$ for every such XFD, and as $b \in C \rightarrow d$, this is at most $O(|\Sigma_a|)$ for all of the edge raising that we do for $A \rightarrow b$. As we have to do this for each XFD in Σ_a, this gives us our total cost of $O(|\Sigma_a|^2)$. However, note that there may be times that an XFD appears in pending, but we perform no edge raising at all when we remove it from the queue. Fortunately, we only add an XFD to pending if the field tDepth changes; this can happen at most $|b|$ times for the XFD, and it is constant time to process the XFD if we do not raise anything. So the sum of all our algorithms are $O(|\Sigma_a|^2)$, giving us our polynomial time complexity.

We now prove that the edge-raisings performed by the algorithm are sufficient. Suppose for a contradiction that we have some $\sigma_a = A \rightarrow b$, σ_a is not satisfied when the above step has finished. This means we have two matches μ_1 and μ_2 that witness this fact. We know furthermore that the two matches need to agree by identity on all paths in A that are internal nodes. Consider now the values they may take on b. Define r to be the longest path reached from a back edge exiting b. The matches for b must fork somewhere properly above r, otherwise they couldn't take different values. Identify the longest prefix of r that the two matches agree on, and call it s.

Now, there are two cases. Either there are paths in A having s as a *proper* prefix, or there are not. If there are not, we see that the two matches agree on p (i.e. longest common prefix of A and b). Thus there are two b nodes reachable from a single p branch, and the algorithm terminated too early, as a type 1 edge-raising should have been performed to raise the edge to p, which is no longer than s.

Algorithm 11 RAISE_EDGES()

Require: `AttList` and `XFDMap` are initialized.
Ensure: All back edges are raised that can be raised.

```
pending = new Queue<int>, inqueue = new Set<int>
for each key index in XFDMap do
  Add index to pending and inqueue
end for
while pending is not empty do
  index = pending.removeFromFront(), remove index from inqueue
  for each summary in XFDMap.get(index) do
    if summary.backEdge > summary.pDepth then
      summary.backEdge = summary.qDepth {Type 1 Raising}
    end if
    if summary.backEdge > summary.tDepth then
      backColumn = summary.prefixes[summary.backEdge].column
      product = Tree[backColumn]edgeProduct.get(summary.tDepth)
      if product > 1 then
        summary.backEdge = summary.tDepth {Type 2 Raising}
      end if
    end if
    {Recompute the $t$ values for the other XFDS}
    for each anode in AttList[summary.bColumn] do
      {Edge raising has no effect if it does not go above RHS}
      if summary.backEdge $\geq$ anode.prefix.depth then
        aSummary = anode.prefix.summary
        depth = aSummary.tDepth
        start = MAX(anode.backEdge, anode.prefix.depth)
        for i = start to summary.backEdge do
          current = aSummary.prefixes[i]
          current.backBelow--
          if current.backBelow == 0, !current.attribOnly then
            aSummary.tDepth = current.depth
          end if
          anode.backEdge = summary.backEdge
        end for
        {Load it in the queue if the candidate $t$ value changed}
        if aSummary.tDepth < depth, aSummary.bColumn $\notin$ inqueue then
          Add aSummary.bColumn to pending and inqueue
        end if
      end if
    end for
  end for
end while
```

On the other hand, suppose there are paths in A having s as a proper prefix. If our matches are to take equal values on those paths, then they must all be attribute paths, and their edges must hit at or above s. Now, consider b. We know that its back edge currently hits properly below s, namely r. Since the two matches for b are different, we know that there exist at least two subtrees rooted at r below s. This, however, means that the product of the annotations between s and r must be strictly greater than 1. We see that we have just assembled all the conditions that should have caused our algorithm to perform a type 2 edge-raising, so we have a contradiction in this case as well.

Finally we argue that both types of edge raising above are necessary. The necessity of the first type is clear. For the second type, suppose we are in a situation where we should perform a particular type 2 raising, but we choose not to. In this case, we can build a witness against $A \to b$ as follows. Take the same nodes by identity for the part of the match for A that does not have t as a prefix and for t itself. Fork at any one allowed fork point between t and p. Note that least one fork is always allowed as the product of the annotations on that section of the tree is strictly greater than 1; also, two full matches for A are always found because T_D is uniformly matched. Complete the matches for the rest of A and for b on either side of the fork, in any arbitrary manner. Since all the A paths we just added end in attribute nodes and their back edges hit at or above t, the two matches are equal on A. Since the back edge from b does not reach t, the two b values we encounter are always different; recall our encoding specifies that the values are different except within each of the unified subtrees. Thus we have the required witness pair.

It now remains to show how to re-check the satisfaction of Σ_i.

Phase 3. Work through Σ_i as in Phase 3(a) of the naïve algorithm. For every $\sigma_i = A \to b$, if all nodes in $\varphi(\sigma_i)$ are internal nodes, attribute nodes with no back edges or attribute nodes with self-loop back edges, proceed to the next dependency; σ_i is still satisfied. Otherwise, let p be the longest common prefix of A and b as before; note that we may have $b = p$.

We know that σ_i is not satisfied when exactly the following conditions hold: All the paths in A having p as a prefix end in attribute nodes and have back edges that hit proper ancestors of p. Also, the product of the annotations *between* p and t, where t is the longest prefix of p hit by such a back edge, is strictly greater than 1.

Lemma 10. *The above correctly identifies the situation where T_D with the value assignment obtained in Phase 2 does not satisfy some σ_i.*

Proof. We first argue that the condition given is necessary for the existence of witnesses to σ_i not holding; then, we argue it is sufficient.

Suppose σ_i was satisfied before Phase 2 was carried out, but is not satisfied now. Observe that we know that $p \to b$ from the way we tested dependency satisfaction in Phase 1. If $p = b$, this is a trivial statement, and if $p \neq b$ but $p \to b$ is not true then σ_i would have been broken in Phase 1 as well.

We now note that in any tree T, under the assumption that $p \to b$, the two dependencies $A \to b$ and $A \to p$ must either both be satisfied or both be broken. If $A \to p$, given that $p \to b$ and $nn(b, p)$, we get $A \to b$. On the other hand if $A \to b$, $b \to p$ is trivial as b is an internal node and $nn(p, b)$, so $A \to p$. So we can reduce the problem of looking for witnesses against σ_i to that of looking for witnesses against $A \to p$.

Consider what a witness against $A \to p$ looks like. p is by definition a prefix of some set of paths in A; call this set A'. If p itself is in A, $A \to p$ always holds, so all the paths in A' must be proper extensions of p. If any path in A' ends in an internal node, $A \to p$ holds, so they must all end in attribute nodes. It must be possible to pick up in T two matches for A' which agree on all the leaves but have a different match for p itself. Thus all these (attribute) paths must have back edges that go strictly above p itself. Finally, we argue the necessity of the cardinality condition between t and p. But observe that if the cardinality condition does not hold, our encoded tree is equivalent to one that has the back edge into t pulled back down to p itself. As we have just shown we need the edges to hit proper ancestors of p, we are done.

The sufficiency argument is very similar to the necessity part of the proof of the previous lemma. Suppose we have a dependency and a tree where the condition above holds. Then we can obtain a witness for σ_i not holding as follows. Let p'' be the longest common prefix of p and $A - A'$, that is those paths in A that are not extensions of p. Take a single match for A up to and including p''. Then build two different matches for all of A by forking somewhere between t and p, where t is as defined in the algorithm description. We can always fork because the product of the annotations is greater than 1, and two full matches for A exist because T is uniformly matched. We thus have two different matches for p, but since all the back edges hit p's proper ancestor t or higher, our two matches for A are equal (i.e. we are comparing attribute paths by value equality). Pick up the appropriate matches for b; it is clear they are different as b is an internal node. We have our required witness pair.

The complexity of this step is no more than $O(|\Sigma_i|^2)$. Thus the total running time of our algorithm is quadratic in $|\Sigma|$, plus an initial scan of D.

Theorem 12. *Given a pair (Σ, D) where D is a #-DTD, checking consistency requires $O(|D| + |\Sigma|^2)$ time.*

6.2 Trivial XFD implications

As with simple DTDs, it is possible for a #-DTD to have trivial XFD implications — those that have a path not in paths(D). Again, we ignore trivial XFD implications for the same reasons we gave before. The linear-time Algorithm given in Algorithms 6 and 7 work for #-DTDs as well. The only difference with a #-DTD is that the scan of E in line 8 Algorithm 6 may read the same symbol twice.

6.3 The chase algorithm

As we have seen, for a $\#$-DTD D, it is possible that a given set of dependencies is inconsistent with D. Thus, an important step when we are given an implication problem instance (Σ, σ, D) must be to check whether (Σ, D) is consistent. If not, clearly $\Sigma \vDash_D \sigma$, because there is no tree satisfying both D and Σ (and not satisfying σ).

A second pathological situation that can arise is that, while Σ is consistent with D, we cannot have a tree T such that $T \vDash D$, $T \vDash \Sigma$ and T contains even one nonnull match for all of $\varphi(\sigma)$. This can arise, for instance, in the case where D has productions

$\rho \rightsquigarrow a^*b, a \rightsquigarrow cc$, $\Sigma = \{\rho.a \to \rho.a.c\}$ and $\sigma = \rho.b \to \rho.a$. Clearly there is only one tree satisfying D and Σ: the tree consisting of the root and one b child. On this one tree, σ also holds, precisely because there is no nonnull match for all of $\varphi(\sigma)$. In cases like this one also, $\Sigma \models_D \sigma$ vacuously. We want to check for both these cases at the start. This can be done efficiently, as explained below.

Let U denote the set of all paths that are either

– in $\varphi(\sigma)$, or
– in paths(Σ) and required to be nonnull by D if $\varphi(\sigma)$ is not null, or
– prefixes of paths in the above two categories

We now explain how to check for both the above pathological situations.

Theorem 13. *The following are equivalent.*

1. *There is no tree T such that $T \models D$, $T \models \Sigma$ and T contains even one nonnull match for all of $\varphi(\sigma)$*
2. *$\Sigma \models_D p.x \overset{U}{\to} p.x.y$ for some $x, y \in$ EL, $p.x, p.x.y \in U$, and D contains a production $x \rightsquigarrow yy\gamma$ for some arbitrary γ. Moreover, we can obtain a derivation to show that $\Sigma \models_D p.x \overset{U}{\to} p.x.y$, using Axioms 1-11 and 13-21 in our axiomatizations.*
3. *Let T' be the smallest uniformly matched tree that satisfies D and contains at least one nonnull occurrence of $\varphi(\sigma)$. $T' \not\models \Sigma$ for any attribute value assignment to the α nodes of T'.*

A few important observations concerning the statement of Theorem 13 are in order. First of all, it is easy to check the Axioms 1-21 are still sound in the presence of #-DTDs. Second of all, we know that there is a uniformly matched tree satisfying D and containing at least one nonnull occurrence of $\varphi(\sigma)$ because of our assumption that every path in $\varphi(\sigma)$ is in paths(D). Since D makes no use of disjunction, this implies that we are free to match each path in $\varphi(\sigma)$ such that a match for the entire pattern is produced (this need not hold if the DTD contains disjunction). Observe also that checking that there is an assignment making $T' \models \Sigma$ is a check for both the pathological situations described above. In particular, the inconsistency of (Σ, D) implies statement 3 in the theorem (but not vice versa).

Theorem 13 is significant because statement 3 can be checked in quadratic time using a variant of our efficient consistency checking algorithm. To do this, we build T_A for the efficient algorithm, but instead of starting with `Tree` containing only the root node, we initialize `Tree` to start with $\varphi(\sigma)$, exactly as we do in Algorithm 1. Thus the initialization now takes $O(|\sigma| + |\Sigma| + |D|)$ time, while the consistency algorithm itself is still $O(|\Sigma|)$. Hence we can check statement 3 in $O(|\sigma| + |\Sigma|^2 + |D|)$ time.

Proof (Theorem 13). The implication from statement 1 to statement 3 is trivial as T' is a tree satisfying the first and third conditions mentioned in statement 1, so it cannot ever satisfy the second condition without falsifying statement 1. Statement 2 gives statement 1 almost immediately, as in the presence of D, $p.x \overset{U}{\to} p.x.y$ (and consequently Σ) can clearly only hold in trees where there are no total matches for U at all. As $nn(\varphi(\sigma), U)$ by assumption, there are no nonnull matches for $\varphi(\sigma)$ either.

It remains to show the implication from statement 3 to statement 2. The proof makes use of the following lemma about T'.

Lemma 11. *Let p be any path that occurs twice or more in T' and that ends in an internal node. Let μ_1 and μ_2 be two matches for p and its proper prefixes, such that they do not agree on p; note that both $\mu_1(p)$ and $\mu_2(p)$ are nonnull, as T' is uniformly matched. Let q be the longest prefix of p on which the two matches agree, let x be the last character of q, and y the next character in p after x. Then D must contain a production $x \rightsquigarrow yy\gamma$ for some γ.*

Proof. We start by observing that the fork at q must certainly be allowed by the DTD, so D must contain a production of the form $x \rightsquigarrow y^+\gamma$, $x \rightsquigarrow y^*\gamma$ (in both cases $y \notin \gamma$), or $x \rightsquigarrow yy\gamma$. So we only need to show the first two cases cannot happen.

Suppose D does indeed contain $x \rightsquigarrow y^+\gamma$ or $x \rightsquigarrow y^*\gamma$ (in both cases $y \notin \gamma$). Consider T^*, a tree obtained from T' by removing either one of the subtrees rooted at $\mu_1(q.y)$ or $\mu_2(q.y)$; the choice can be arbitrary. Clearly T^* still satisfies D.

We can show T^* is still uniformly matched. Consider any match for a path $q.y.r$ that was previously found within the removed subtree. If this subtree was the only place in T' where nonnull matches for $q.y.r$ were found, T^* contains no matches for $q.y.r$ and is uniformly matched with respect to $q.y.r$. On the other hand, if there were other nonnull matches for $q.y.r$ outside the removed subtree, they are all preserved in T^*. If T^* is not uniformly matched with respect to $q.y.r$, neither was T'.

We can also show that T^* must contain at least one nonnull match for all of $\varphi(\sigma)$. We know T' contained a nonnull match for all of $\varphi(\sigma)$. The only way T^* would not do so is if the removed subtree contained the only nonnull match for some path in $\varphi(\sigma)$ having $q.y$ as a prefix. However T' was uniformly matched, so any path found in the removed subtree was also found in the one that was retained, so this cannot happen.

Observe that we have just produced a tree T^* satisfying three of the four conditions that define T'. The fourth condition on T' is minimality; but T^* has properly fewer nodes, so we have contradicted T''s minimality. By the definition of p and q, p is $s.x.y.t$ for some t and $q = s.x$. Consequently, the relevant production at the fork must indeed be $x \rightsquigarrow yy\gamma$, and we are done.

We now return to the proof of the implication from statement 3 to statement 2. So suppose that $T' \not\models \Sigma$. Now, the efficient algorithm that we ran on T' returned "no" either in the encoding of Phase 1 or Phase 3.

Case 1: the algorithm returned a negative answer the encoding of Phase 1, because some $\sigma = A \to b$, b an internal node, was checked and found not to be satisfied. We now need the following result.

Lemma 12. *In this case, b must be of the form $q.x.y.r$ for $x, y \in$ EL, where*

1. *D contains a production $x \rightsquigarrow yy\gamma$ for some γ, and*
2. *$q.x.y$ is not a prefix of any path in A.*

Intuitively, b must have a DTD-induced fork somewhere outside of A.

Proof. From the algorithm description, we see that in this case T' must contain a single match for A that can "pick up" two different matches for b; before phase 2 is carried out, all attribute values in T' are different, so the match for A must truly include all the same nodes, comparing by identity. This immediately means b cannot be a prefix of any path in A. Let p denote the longest common prefix of A and b.

Fix a witness pair of matches against $A \to b$; call them μ_1 and μ_2. Observe that both are completely nonnull, as T' is uniformly matched. Also note that they must both agree on p. Now let $p.s$ be the longest prefix of b on which μ_1 and μ_2 agree; note that s may be null. Thus b must be of the form $p.s.t$, where t is not null, and the two b branches fork exactly at the end of s. Let x be the last character of $p.s$. By Lemma 11, D must contain a production $x \rightsquigarrow yy\gamma$.

From the lemma we just proved, we can now argue as follows. We know that Σ contains $A \to q.x.y.r$, with $q.x.y$ not a prefix of any path in A. Divide A into those paths that do not contain $q.x$ as a prefix, which we call A', and those that do, which we call A''. Note either of these parts may be empty. We thus have $A', A'' \to q.x.y.r$. Consequently $A', A'' \overset{A,b}{\to} q.x.y.r$. Now by our axiom 19, since nothing in A'' has $q.x.y$ as a prefix, we can deduce that $q.x \overset{A,b}{\to} q.x.y.r$, which gives by axiom 13 $q.x \overset{U}{\to} q.x.y.r$. As b is an internal node, we have $q.x.y.r \to q.x.y$, so $q.x.y.r \overset{U}{\to} q.x.y$, and by transitivity $q.x \overset{U}{\to} q.x.y$ as required. This gives us the required derivation for $\Sigma \vdash_D q.x \overset{U}{\to} q.x.y$, and so by the soundness of the axioms we have $\Sigma \models_D q.x \overset{U}{\to} q.x.y$ as well.

Case 2: the algorithm returned a negative answer in Phase 3, because some $\sigma = A \to b$, b an internal node, was checked and found not to be satisfied. In this case, we need the following lemma.

Lemma 13. *In this case, b must be of the form $q.x.y.r$ for $x, y \in$ EL, where*

1. *D contains a production $x \rightsquigarrow yy\gamma$ for some γ, and*
2. *$q.x.y$ **may** be a prefix of any path in A (the fork may now occur inside A)*

Moreover, denote by $q.x.y.A$ be those paths in A having $q.x.y$ as a prefix. If $q.x.y.A$ is not empty, we must have $\Sigma \vdash_D q.x \overset{U}{\to} q.x.y.A$, via a derivation using Axioms 1-11 and 13-21.

Proof. The argument that b must contain a DTD-induced fork follows, as in the previous case, from Lemma 11. We now show that if $q.x.y.A$ is not empty, $\Sigma \vdash_D q.x \overset{U}{\to} q.x.y.A$. Observe that all paths in $q.x.y.A$ must end in attribute nodes, and Phase 2 must have raised all their back edges to or above $q.x$, otherwise the algorithm could not have found our two witness matches and returned a negative answer.

Define $q.x.y.Z$ to be the set of all paths s in U having $q.x.y$ as a prefix, ending in an attribute node, and such that the algorithm that we ran on T' terminated in Phase 2 with a back edge from s that reaches to or above $q.x$. We show that $\Sigma \vdash_D q.x \overset{U}{\to} s$ for all $s \in q.x.y.Z$. From this it follows that $\Sigma \vdash_D q.x \overset{U}{\to} q.x.y.A$, as $q.x.y.A \subseteq q.x.y.Z$. The proof uses induction on the order in which the algorithm raised the appropriate back edges from the $q.x.y.Z$'s to or above $q.x$ in Phase 2.

Consider the particular $s \in q.x.y.Z$ whose back edge was the first to be raised to or above $q.x$. We see this raising must have been a type 1 raising due to some XFD from $C \to s \in \Sigma$, with $q.x.y$ not a prefix of any path in C; otherwise neither raising rule could have fired yet. In this case $C \to s$ gives us $C \overset{U}{\to} s$, and from there, by a simple application of Axiom 19, we have $q.x \overset{U}{\to} s$ as required.

For the inductive case, suppose our s is not the first path in the set to have its back edge raised to or above $q.x$. If this raising happened for the same reason as in the base case (i.e. there was some $C \rightarrow s$ with no path in C having $q.x.y$ as a prefix), then the same argument as in the base case applies. In the other case, the raising must have been a type 2 raising due to some XFD $C \rightarrow s$. Moreover, at least one of the back edges involved must have come from an s', also having $q.x.y$ as a prefix. The back edge of that s' must have been raised to or above $q.x$ in a previous step. Denote the set of all such $s' \in C$ by C'.

However, by our inductive hypothesis we already have $q.x \overset{U}{\rightarrow} s'$ for all appropriate s'. Thus $q.x \overset{U}{\rightarrow} C'$, since FFDs compose normally on the right-hand side, using their own transitivity and augmentation axioms. But by Axiom 19 applied to $C \rightarrow s$, we have $q.x, C' \overset{U}{\rightarrow} s$. Augmentation and transitivity (for FFDs) give us $q.x \overset{U}{\rightarrow} s$ as required.

From the lemma just proved, we can now reason as follows. As in the previous case, we know We know that $A \rightarrow q.x.y.r$. As before, we deduce $A', A'' \overset{A,b}{\rightarrow} q.x.y.r$. Now by our Axiom 19, we deduce that $q.x, q.x.y.A \overset{A,b}{\rightarrow} q.x.y.r$, which by Axiom 13 gives $q.x, q.x.y.A \overset{U}{\rightarrow} q.x.y.r$. If $q.x.y.A$ is empty this is the same as $q.x \overset{U}{\rightarrow} q.x.y.r$. If it is not empty, we know from Lemma 13 that $q.x \overset{U}{\rightarrow} q.x.y.A$. Hence $q.x \overset{U}{\rightarrow} q.x, q.x.y.A$ by augmentation, $q.x \overset{U}{\rightarrow} q.x.y.r$ once again by transitivity. Finally, as b must again be an internal node in this case, by transitivity $q.x \overset{U}{\rightarrow} q.x.y$ as required.
This completes the proof of Theorem 13.

Having ruled out all above pathological cases, it turns out that we can now use exactly the same algorithm as in the case of simple DTDs to complete our decision procedure. As a result, the complexity of implication falls out immediately from Theorem 8 and our discussion above.

Theorem 14. *Given (Σ, σ, D) where D is a #-DTD, implication can be decided in $O(|\Sigma|^2 + |\sigma| + |D|)$ time.*

The proof of correctness, of course, is different for this case.

Proof of correctness As was the case with simple DTDs, we still wish to prove the following result.

Theorem 15. *After the chase terminates, the resulting tableau Tab corresponds to $U(R(T_{chase}))$ for some document T_{chase}. Furthermore, T_{chase} can be extended to a (potentially infinite) tree T'' which must satisfy both D and Σ, and $Tab \not\models \sigma$ implies $T'' \not\models \sigma$*

Proof. This proof covers several pages and has many parts. For clarity, each part of the proof is prefaced by a section number prefixed by **P**, with the heading in small caps. For example, our first part of the proof is to construct T''.

P.1 CONSTRUCTION OF T''

Observe that T_{chase}, the tree derived from the tableau as in the previous constructions, is a tree that consists of two nonnull occurrences of all the paths in U. Some of these occurrences may share initial prefixes, but there will in general be forks — unless the chase terminated with all columns unified. Define a *fork point* as the maximal prefix of some path $p \in U$ on which the two occurrences of p in T_{chase} agree. Identify the set of all fork points in T_{chase}, denoting the set as $\{ q_1, q_2, \cdots q_k \} \subseteq U$. Each q_i has a corresponding set of paths in U, $\{ p_{i1}, p_{i2}, \cdots p_{il} \}$, such that each p_{ij} is of the form $q_i.x$ for some $x \in$ EL —we only consider single-character extensions of q_i – and the two occurrences of p_{ij} in T_{chase} fork exactly at q_i. Of course, the set of fork points may be empty.

We say that a particular q_i is a *fork point with respect to* a path s if $s = q_i.x.t$ for some t, and $q_i.x$ is one of the p_{ij}s for that q_i. Thus q_i is a fork point with respect to s if and only if T_{chase} contains two occurrences of s that agree on q_i and fork immediately after q_i. Observe that if q_i is a fork point with respect to s, then it is also a fork point with respect to all t such that $q_i \prec t \preceq s$.

The first stage of the construction of T'' builds the structure of the tree, the second stage deals with assigning attribute values to any attribute nodes.

P.1.1 BUILDING THE STRUCTURE OF T''

For the structural construction, start with T' as featured in Theorem 13; ignore the attribute value assignments at this point. If the set of fork points identified above is empty, there is nothing to add to T' — proceed to attribute value assignment. Otherwise, we may or may not add extra subtrees to T', according to the following procedure.

Observe T' contains at least one nonnull occurrence of U, when U is considered as a tree pattern in the obvious way. Designate this occurrence by coloring it red. Our aim is to add nodes to T' to build a tree T'' which contains two designated occurrences of U, the red one and another one. The two occurrences will fork exactly at the q_i's identified above, so that the structure of T_{chase} will isomorphically embed into T''.

We construct T'' by iterating through all q_i. For each q_i, iterate through all p_{ij}. For each p_{ij}, consider the T' we started the construction with. We know T' contains at least one occurrence of p_{ij} - within the red occurrence of U. It may also contain a second occurrence of p_{ij} that is a direct sibling of the red occurrence. This second occurrence will exist if and only if the DTD required two children with the same (appropriate) label immediately below q_i. The operation we perform depends on whether the second occurrence is present or not.

1. If T' does contain a second occurrence of p_{ij} as a sibling of the red one, make no modifications at all and move to the next p_{ij}.
2. Otherwise, identify the subtree within T' rooted at the red occurrence of p_{ij}. Call that entire subtree S. Make a structurally isomorphic copy of S, and paste the copy into the big tree as a sibling of S.

Observe here that our operations for each p_{ij} are completely independent, in the sense that any two p_{ij} operations always involve disjoint sets of nodes. Thus earlier modifications do not affect later ones. In particular, this means that whenever we do insert an extra copy of a subtree, we are always inserting a copy of a subtree already present in T' itself.

This completes the construction of the *structure* of T''. All that remains is to show how we assign attribute values to any attribute nodes. Before we do this, however, we can already observe several things.

Lemma 14. *T'' is uniformly matched.*

Proof. This is clear because the only operation we perform is adding subtrees which are structurally identical to subtrees in the uniformly matched T'.

Lemma 15. *$T'' \models D$.*

Proof. $T' \models D$, so any violations must have come from the added subtrees. However, the DTD allowed a fork at q_i for each p_{ij} (or the tableau chase terminated too early). Thus, we cannot have violated D by the addition of an extra occurrence of p_{ij} itself. As for the rest of the subtree we added below the new occurrence of p_{ij}, it is structurally identical to a subtree S that existed in T', so again the addition cannot have caused a violation of D.

Lemma 16. *T'' contains two occurrences of U, forking exactly as required by the tableau. That is, the structure of T_{chase} embeds into the structure of T'' as required.*

Proof. For any p_{ij} of interest, let U_{ij} denote the subset of U having p_{ij} as a prefix. For each U_{ij}, we need to show we have a second occurrence of U_{ij} as a sibling of the red one that was found in T'. There are two cases according to our construction above. If the second case fired in the construction of this $U_{i,j}$ we added the appropriate occurrence explicitly when we copied the subtree containing the red occurrence. On the other hand if the first case fired, using the fact that T' is uniformly matched, we see that the other (non-red) sibling subtree we found at the appropriate place must already have contained a full nonnull occurrence of U_{ij}. So we are done in all cases. Designate these two occurrences of U as special for the remainder of the proof by extending the red coloring to the other occurrence.

P.1.2 CHASING TO ASSIGN ATTRIBUTE VALUES TO T''

We are now ready to assign attribute values. Start by working on our two designated occurrences of U, i.e. focus on all the red-colored attribute nodes. Assign to them any values, as long as the assignment is consistent with Tab. That is, the two occurrences of any red path t receive identical attribute values if, and only if, the tableau unified the t column. For the remainder of nodes in T'', whether they themselves belong to other occurrences of U or not, assign fresh and unique values everywhere.

The remainder of the attribute value assignment is done by running a three-phase chase on $U(R(T''))$, exactly as we did in our naïve consistency checking algorithm. Observe that our two red matches for U are present as two tuples in $U(R(T''))$ - we refer to these as the red tuples. Again we chase on $U(R(T))$ with Σ partitioned into Σ_i and Σ_a, and we return "yes" if and only if no dependency in Σ_i ever attempts to fire, either in Phase 1 or in Phase 3. After the chase terminates successfully, as we shall prove it always does, convert $U(R(T''))$ back to a tree. As its structure has not changed, this tree still satisfies D. It also satisfies Σ, as we have just chased with Σ successfully.

This completes the description of the attribute value assignment algorithm. It remains to prove that this is correct.

P.2 CORRECTNESS OF THE CONSTRUCTION

To prove correctness, we need to prove the following two propositions.

Proposition 5. *The three-phase chase on $U(R(T''))$ always returns "yes"*

Proposition 6. *T_{chase} still embeds into the final tree T'' after Phase 2 of the chase*

We prove Proposition 5 by splitting it further into the following two propositions.

Proposition 7. *Phase 1 of the chase on $U(R(T''))$ never returns "no"*

Proposition 8. *Phase 3 of the chase on $U(R(T''))$ never returns "no"*

The above Propositions are sufficient for the proof of the whole theorem. In this case we will have found a T'' that satisfies D and Σ and embeds a copy of T_{chase}. Finally, the statement $Tab \not\models \sigma \Rightarrow T'' \not\models \sigma$ will be clear, because T_{chase} embeds into T'', and thus T'' will still contain the two witnesses against σ that were present in the tableau.

We need only present the proofs of Propositions 6, 7 and 8. In all three proofs, we use the fact that the chase on the small tableau Tab succeeded, and that the three-phase chase on T' succeeded. Since three chases are involved, it is useful and avoids confusion to summarize their most important similarities and differences.

– All three chases are done with the same set of XFDs Σ.
– The chase on the small tableau uses the second chase rule as well as the first. The other two do not, because they work on tableaux that are closed under a transformation to a tree and back.
– The chases on T' and T'' are three-phase algorithms, the chase on the small tableau is one-phase.
– The chase on the small tableau and the T' chase are at this point known to have succeeded.
– The chase on the small tableau allows unification of internal node columns, the other two do not (i.e. they throw an error if this is attempted). Because the chase in T' succeeded, in particular, we know no unification of internal node columns on $U(R(T'))$ was ever attempted.

P.2.1 PROOF OF PROPOSITION 7

Suppose for a contradiction the chase throws an error in this phase, because the large tableau does not satisfy some $A \to b \in \Sigma_i$, b ends in an internal node (from the definition of Σ_i). As usual, we have two matches μ_1, μ_2 for A and b that witness this fact and caused the algorithm to return "no". Since T'' is uniformly matched, both matches are completely nonnull.

Consider the matches on A. Clearly, for every a in A, we must have that either

– $\mu_1(a)$ and $\mu_2(a)$ agree by identity, or
– a is an attribute path. Moreover, $\mu_1(a)$ and $\mu_2(a)$ pick up exactly the two red matches for a - that is the only way to find two equal attribute values for a in our current T''.

Now, let q be the longest prefix of b on which $\mu_1(b)$ and $\mu_2(b)$ agree; this always exists as the root will do in the worst case. Suppose b is $q.x.t$ for some $x \in$ EL. t may be null but x must exist, otherwise the matches agree on b after all.

If any $a \in A$ has $q.x$ as a prefix, we can obtain a contradiction easily. Let A' be the set of all such a. Observe that all of them must end in attribute nodes, so they must be colored red in the tree. So we have two red matches for each $a \in A'$ which agree on q but disagree on $q.x$. Consequently, we see that q must be a fork point in T_{chase} with respect to $q.x$. This tells us the small tableau did not unify the $q.x$ column (from the definition of fork points). We also see that the two rows in the small tableau must have agreed on A', and also on q. This sets up all the conditions necessary for an application of the second chase rule on the small tableau to use $A \rightarrow b$ and unify the b column. This unification in turn must have led to the unification of $q.x$ in the small tableau. The μ_1, μ_2 witness that this unification did not happen, as $\mu_1(q.x)$ and $\mu_2(q.x)$ are different by assumption, but both red. In this case, we have a contradiction — the chase on the small tableau terminated too early.

Therefore, we can assume that at most q is a prefix of any path in A. This tells us that A and b have a longest common prefix p that is strictly shorter than b, and that p is a prefix of our q. Consequently, we have found in our tree T'' a situation where a single match for p extends to two matches for b. We know, however, that this situation did not arise in our T', because Phase 1 of the chase on T' would have caught just such a situation. So consider how this problem could have been created in moving from T' to T''.

First, it is posible that $\mu_1(p) = \mu_2(p)$ itself was not in T''. This means it was added as part of some new subtree, together with both of its b matches. However, we know that we only add subtrees structurally identical to ones already in T', so T' must have contained another subtree with a single occurrence of p branching out to two b's. Then, however, we had the same problem in T' and the chase in T' should have failed.

So it must be that $\mu_1(p) = \mu_2(p)$ was in T', but one or both of $\mu_1(b)$ and $\mu_2(b)$ were not. If both were not, they were both added within the same new subtree, where the new subtree was rooted at some p_{ij}, where $p \prec p_{ij} \preceq q$. However, in this case there must have been in T' another occurrence of p_{ij} sitting below $\mu_1(p)$, with its own two occurrences of b. Once again, we would have had the same problem in T'.

The only remaining possibility is that one of $\mu_1(b)$ and $\mu_2(b)$ was in T' and the other was not. In this case, one of these b matches came from a new subtree and the other did not. This means that q is a red fork point with respect to $q.x$. Thus $\mu_1(q) = \mu_2(q)$ is a red node and the small tableau did not unify the $q.x$ column. Now, transporting our thinking back to the small tableau and recalling that no path in A has $q.x$ as prefix, we see that once again, the second chase rule should have fired to unify the b columns in the tableau. By transitivity, as $q.x$ is an ancestor of the non-attribute node b, the chase would have continued upwards, eventually disallowing the red fork at $q.x$. We have a contradiction in this case as well, so we are done.

P.2.2 PROOF OF PROPOSITION 6

This proposition is an immediate corollary to Lemma 17 below. For this lemma, let U' be the subset of U consisting of attribute paths whose columns were *not* unified by the chase on the small tableau. That is, for any $u \in U'$, the red matches for u in our T'' before the chase do not agree. For a given $u \in U'$, let p_u be the longest prefix of u on which the two tableau rows do agree, which may be the root in the worst case. Suppose $u = p_u.x.t$ for some $x \in \textsc{El}$, where x and t are both nonnull as u is an attribute path.

Lemma 17. *Consider T'' after Phase 2 of the chase has been completed. Suppose μ_1, μ_2 are two matches in this T'' such that $\mu_1(u) = \mu_2(u)$ for some $u \in U'$. Then $\mu_1(p_u.x) = \mu_2(p_u.x)$.*

Informally, this lemma states that if some attribute u was not unified in the small tableau, then any unifications in the second stage of the final chase will be limited to "local" unifications — within the same subtree rooted at $p_u.x$. No unifications will be propagated wider in the tree.

Proof. Suppose not. Observe that the property in the lemma holds at the start, before Phase 2 of the chase. So, if it is broken afterwards, there must have been a first unification where it was violated. Identify the $u \in U'$ that was the first to have its value propagated "illegally" (i.e. beyond its $p_u.x$ subtree). Identify the tuples in $U(R(T''))$ where this happened, and identify the tuples with two matches μ_1, μ_2. We therefore know $\mu_1(p_u.x) \neq \mu_2(p_u.x)$. The unification of $\mu_1(u)$ with $\mu_2(u)$ that caused the illegal propagation was due to some XFD $C \to u$.

Suppose $p_u.x$ was not a prefix of any path in C. Then, however, the chase on the small tableau should have unified the two u columns when considering $C \to u$ (recall the small tableau rows agree on p_u). So we must have $p_u.x$ a prefix of some paths in C. Of course, all these paths must be attribute paths, otherwise the two tuples μ_1, μ_2 would not agree on C. Call these paths $c_1, \cdots c_k$.

Now, suppose all of the c_i's had their values unified in the small tableau. In this case, again the chase on the small tableau should have unified u by the second chase rule. So some specific c_j was not unified in the small tableau. However, this gives us a contradiction too. To allow a unification with $C \to u$ on μ_1, μ_2, we see that the c_j value must have been unified between both tuples (and between both $p_u.x$ subtrees) earlier. However, we assumed that our u was the first one to have its value propagated outside of its own $p_u.x$ subtree, so we are done.

Lemma 17 gives us Proposition 6 as a corollary. Phase 2 does not change the structure of the tree, so T_{chase} still embeds structurally into the red tree present in T''. Any attribute values unified in T_{chase} stay unified in the red tree. The only problem could arise if for some $u \in U'$, the two variables in the two red tuples were unified by the chase with Σ_a. However, the two red tuples still would not agree on $p_u.x$ after Phase 2, giving us a counterexample to Lemma 17 — a contradiction.

P.2.3 PROOF OF PROPOSITION 8

As usual, we argue by contradiction, and suppose we have some dependency $A \to b$ not satisfied in Phase 3 of the chase on T'', with b an internal node, and μ_1, μ_2 are matches for U in the tree (and rows in $U(R(T''))$) witnessing the non-satisfaction. No null values are involved anywhere, because T'' is uniformly matched.

Let p be the longest common prefix of A and b, as usual, where we allow $p = b$. If $\mu_1(p) = \mu_2(p)$, we already have a contradiction: either $p = b$ and the two matches actually agree on b, or $p \neq b$ and we are back in the case where some single match for p extends to two matches for b. We would have caught this in phase 1 of the chase on T''. Thus the matches must fork somewhere properly above p; call the last prefix of p on which they agree q_m; the m here stands for "matches". Let $b = q_m.x.t$ for some $x \in$ EL; if there is no such x, $q_m = b$ and the matches agree on b, which is a contradiction. Since we have just argued that q_m is not the longest common prefix of A and b, there must be a set of paths in A having $q_m.x$ as a prefix. Call that set A'. As usual, we know that all paths in A' must be attribute paths. It is clear from this discussion that $\rho \preceq q_m \prec p \preceq b$.

On the other hand, the small tableau may or may not have had the b column unified, but the two rows must have a longest prefix of b on which they agree, the root in the worst case. Call that prefix q_t; the t here stands for "tableau". In this case must have $\rho \preceq q_t \preceq b$. We now prove a useful lemma.

Lemma 18. *Below the node $\mu_1(q_m) = \mu_2(q_m)$ there is at most one red b node. Also, the same property must hold for every s such that $q_m \prec s \preceq b$.*

Proof. The proof is by cases on the possible relationship between q_t and q_m. First, suppose q_t is a proper prefix of q_m. We have the following ordering of paths,

$$\rho \preceq q_t \prec q_m \prec p \preceq b$$

In this case, suppose $q_m = q_t.y.s$ for some $y \in$ EL; s may be null. Observe that q_t is a fork point with respect to $q_t.y$. Thus the two red occurrences of b in T'' do not agree on $q_t.y$. Now, μ_1 and μ_2 must agree on $q_t.y$ by our assumption that $q_t \prec q_m$, so the most they can do is take one of the two red $q_t.y$ nodes, not both. As $q_t.y \preceq q_m$, it is clear that once we reach $\mu_1(q_m)$, we no longer have access to both red b branches in the tree. The red tree is uniformly matched, as it was generated from the small tableau, which contains no null values. Therefore, there can only be one red $q_m.s$ below $\mu_1(q_m)$, where $q_m \prec q_m.s \preceq b$; if f there were two red $q_m.s$ nodes below $\mu_1(q_m)$ but not two b nodes, we would have contradicted the uniform matching of the red tree.

The second case is when $q_m \preceq q_t$. We show that in this case we must have $q_t = b$. In other words, there is only one red occurrence of b, and consequently each of its proper prefixes, in T''. Lemma 18 follows immediately — if there is only one red b in the entire tree, it is also unique below a specific q_m match.

Suppose for a contradiction $q_t = b$ is not true. Then $b = q_t.x.s$ for some $x \in$ EL and potentially null s. Consider any paths $a \in A$ having $q_t.x$ as a prefix; call this set A_t. First observe that all paths in A_t, if they exist at all, must be attribute paths; μ_1 and μ_2 fork at or above q_t, but they were able to pick up equal values for all the $a \in A_t$.

First, suppose A_t is empty or that the small tableau unified all A_t. In this case the second chase rule would have fired on the small tableau to unify the two b instances and $q_t = b$ after all. On the other hand, suppose the tableau did not unify some $a \in A_t$. But then by Lemma 17, even after the chase with Σ_a on T'', two matches disagreeing on $q_t.x$ cannot pick up two equal values for this a, and μ_1 and μ_2 certainly disagree on $q_t.x$. So we have a contradiction. This tells us that there is exactly one red b in T'' as required.

Corollary 3. *Let s be any path such that $q_m \preceq s \preceq b$. No occurrence of s below $\mu_1(q_m)$ is a red fork point with respect to b.*

Proof. If $q_t \prec q_m$ the fork point affecting b occurs properly before q_m is reached, so s is too late for it to happen. On the other hand if $q_m \preceq q_t$, there is no red fork point with respect to b branch at all, at s or anywhere else.

The remainder of of the proof of Proposition 8 is to show that a violation of $A \to b$ would have been detected in Phase 3 of the chase on T'. This proof has the following outline.

– We identify a subtree within T'', coloring it green. Within $U(R(T''))$, the coloring will produce some rows that are completely green, and some rows that are partially green. We refer to the rows which are completely green as the *green subtableau*.
– We identify a blue subtree within T' which is structurally isomorphic to the green one. We show that before Phase 2 of each respective chase on T'' and T' occurs, the green and blue subtrees actually embed in each other, and consequently the corresponding green and blue subtableaux also embed into each other.
– We show that for any two green occurrences of a path p in T'' whose values are unified during Phase 2 of the chase on T'', the same unification could also have been performed at the same point in the chase using two tuples in $U(R(T''))$ that contained only green nodes.
– It follows that it is possible to reorder the steps of the chase with Σ_a on T'' so that all unifications involving pairs of green nodes happen first, and all other unifications later. Thus we split Phase 2 of the chase on T'' into two subphases. First, a green subphase involving only the green subtableau that terminates exactly when the green subtableau satisfies Σ_a, and second, a non-green subphase performing all other unifications that do not affect the green subtableau. As the chase is Church-Rosser, this splitting into subphases is perfectly legal.
– It then follows that it is also possible to reorder the steps of Phase 2 of the chase on T' so that we start with a corresponding blue subphase. By following exactly the same chase order in the blue subphase as we did in the green subphase, we will be guaranteed to produce a blue subtableau satisfying Σ_a that embeds to and from the green one.
– Finally, we show that the green subtableau must contain a witness against $A \to b$ at the end of the green subphase of the chase. It follows that the blue subtableau also contains such a witness after the blue subphase. As the non-colored subphases of the chase with Σ_a do not affect the colored subtableaux, it follows that T' contains a witness against $A \to b$ after phase 2 of the chase. Thus Phase 3 would have found that witness and returned “no”.

P.2.3.1 DEFINITION OF THE GREEN TREE WITHIN T''. We begin by describing our green subtree within T''. For nodes at the end of paths p not having $q_m.x$ as a prefix, color green either one of $\mu_1(p)$ or $\mu_2(p)$; make the choice arbitrarily but consistently for all p. Any further green coloring occurs only below our one designated q_m which is already green. Below this q_m, color green all paths p having $q_m.x$ as a prefix, except if the following are both true.

– The green q_m is also red.
– There is some path of the form $q_m.x.s$ for possibly null s which is a red fork point, so that below our green q_m there are two red occurrences of some $q_m.x.s.y$ for some $y \in$ EL.

If this is true, color green only one of the two subtrees rooted at the red occurrences of $q_m.x.s.y$. This coloring below q_m is equivalent to removing from consideration all subtrees that were added during the construction of T'' at fork points $q_m.x.s$, and also those that were not added, but found in the tree and marked as second red occurrences. We now note several properties of the green tree we have just produced.

Lemma 19. *The green tree contains at most one red occurrence of any path t.*

Proof. For paths not having $q_m.x$ as a prefix, the green tree contains a unique occurrence of such paths, red or not. For paths having $q_m.x$ as a prefix, suppose that there are indeed two occurrences of some such t. By Lemma 18, there is at most one red occurrence of $q_m.x$ below the green q_m, so the two matches for t must agree on $q_m.x$. In fact, they must agree on the longest common prefix of b and t, by the same argument. Thus they were generated by a red fork occurring at that longest common prefix, which was of the form $q_m.x.s$. However, in our coloring above, we explicitly retain only one side of all such forks, so we cannot have retained both red occurrences of t after all.

Corollary 4. *Before T'' is chased with Σ_a, all the attribute values at the end of green paths are distinct in the green tree.*

Proof. As we have seen, the only way to pick up two equal attribute values before the chase with Σ_a is to take two occurrences of a red path, but the last lemma states this cannot happen.

Lemma 20. *Both $\mu_1(b)$ and $\mu_2(b)$ are green.*

Proof. Suppose not, then one or both of the b's were in a subtree that was not colored green because it was on one side of a red fork and rooted at some $q_m.x.s.y$. This means $q_m.x.s.y$ is a prefix of b and consequently $q_m.x.s$ is a fork point with respect to b, given our definition of what it means to be a "fork point with respect to" some path. However, this contradicts Corollary 3.

P.2.3.3 DEFINITION OF THE BLUE TREE WITHIN T'. We now need to show how to find a blue tree within T' structurally isomorphic to the green one we found in T''. We choose any arbitrary occurrence of q_m in T' and color it blue. For any path t not having $q_m.x$ as prefix, choose any one occurrence of t in T' that is consistent with our blue q_m (in the sense that our blue matches for q_m and t must agree by identity on the longest common prefix of these paths). Such an occurrence can always be found for each t because T' is uniformly matched.

For paths having $q_m.x$ as a prefix, color blue all the nodes required to produce a structural isomorphism between the blue and green trees. We need to show this can always be done. We know that in the construction of T'', there was no addition of a sibling subtree rooted at $q_m.x$ (by Lemma 18 q_m was not a red fork point with respect to $q_m.x$). Thus any x children below the green q_m were required to exist by the DTD and the uniform matching constraint with respect to $\varphi(\sigma)$, and we can expect to find the same number of x children below our blue q_m. Now, since at most one green $q_m.x$ is red, all but one of the subtrees rooted at green $q_m.x$ nodes were entirely generated by the DTD and the uniform matching requirement with respect to $\varphi(\sigma)$, so the subtrees we find below our blue $q_m.x$ nodes are structurally isomorphic to each other and all but one of the green subtrees. It remains to argue about the one green $q_m.x$ which might also have been red. However, we have explicitly removed from the green coloring any subtrees below that $q_m.x$ that might have been added in the construction of T'' (and we may indeed have removed nodes that actually were in T' too). So a structurally isomorphic copy of this last green subtree can be found in any $q_m.x$ subtree in T'. Thus we can always extend the blue coloring appropriately.

It is clear from the description of the blue coloring that the blue and green trees are structurally isomorphic. Now, observe that before T' is chased with Σ_a, all the attribute values in the entire T' are distinct. Thus they are distinct in the blue subtree. From this observation, the structural isomorphism of the blue and green trees, and the last corollary in the section on the green tree, we see that before T' and T'' are chased with Σ_a, the blue and green trees actually embed into each other. As the blue and green subtableaux are exactly the relational representations of the blue and green subtrees, it follows that the blue and green subtableaux also embed into each other.

P.2.3.3 GREEN UNIFICATIONS CAN BE PERFORMED USING GREEN TUPLES ONLY. At this step of the proof, we focus on the following lemma.

Lemma 21. *Suppose Phase 2 of the chase on T'' (with Σ_a) unifies the values at the end of two green occurrences of an attribute path t using some dependency $C \to t$. Then, at that same point in the chase, it is possible to make the same unification with $C \to t$, but instead make it using two tuples found in the green subtableau. If intermediate chase steps are required to make this fully green unification possible, the intermediate steps themselves also occur entirely within the green subtableau.*

Proof. The proof is by induction on the green unifications. That is, we start with the first two green occurrences of some path t that have their values unified by the chase, and then proceed in the order in which subsequent green unifications – not necessarily for the same t — proceeded. Our inductive hypothesis assumes that all previous unifications between pairs of green paths were performed entirely on the green subtableau.

We now focus on the two green t values that were unified due to some dependency $C \to t \in \Sigma_a$, via two matches μ_1 and μ_2 for $C \cup \{ t \}$. If all the nodes in μ_1 and μ_2 were green, we are done. So suppose not all are green. The inductive hypothesis enables us to reuse the green parts of μ_1 and μ_2 in producing our new, fully green matches that will cause the same unification. To see why this is the case, suppose $\mu_1(p)$ and $\mu_2(p)$ agree on some attribute p and both are green. Then either they agree on p by identity or the occurrences of p they pick up were unified in a previous chase step. Thus we can assume we have a fully green chase sequence unifying those p's, and so we can reuse the green parts of μ_1 and μ_2 freely. We now consider which nodes within μ_1 and μ_2 might *not* be green.

Clearly $\mu_1(t)$ and $\mu_2(t)$ are both green, by assumption. This tells us they both have $q_m.x$ as a prefix, as this is the only way to find two different green occurrences of the same path. We also know that $\mu_1(q_m) = \mu_2(q_m)$ and this node is the one green q_m. We now argue by cases on what else may not be green.

First, suppose that all non-green nodes in μ_1 and μ_2 are matches for paths that do not have $q_m.x$ as a prefix. In this case we can build matches μ_3, μ_4 that both pick up the unique green occurrence of each of these paths, and take $\mu_3(s) = \mu_1(s), \mu_4(s) = \mu_2(s)$ for all paths s that do have $q_m.x$ as prefix. These new matches are available to us at this point in the chase and they are completely green, so they correspond to tuples found in the green subtableau. Moreover they agree on C because they are unchanged from μ_1 and μ_2 respectively on any $c \in C$ having $q_m.x$ as a prefix, and they agree by identity on all $c \in C$ that do have $q_m.x$ as a prefix. Thus they also force a unification of the two occurrences of t.

So suppose we have some nongreen nodes in either μ_1 or μ_2 or both that are in C and have $q_m.x$ as a prefix. Call these problematic paths $C'' \subseteq C$. This means at least one of our matches took its values on C'' from a subtree that did not receive green coloring, as it was rooted at one of two red children of a red fork point. We may in general have multiple such problems, involving multiple subtrees that did not receive green coloring, so in fact it may be possible to partition C'' into several subsets, one per problematic subtree. However, the subtrees are all independent. For ease of presentation, we prove the case for when there is only one problematic subtree; in this case all $c'' \in C''$ have a common prefix $q_m.x.s.y$, which had a red fork point at $q_m.x.s$. If there are multiple problematic subtrees, the argument can easily be repeated for each one.

We consider the subcases. First suppose both $\mu_1(C'')$ and $\mu_2(C'')$ are not green. But $\mu_1(t)$ and $\mu_2(t)$ are green, so $q_m.x.s.y$ cannot be a prefix of t. Thus we are free to find a green match for C'' in the sibling subtree thad did receive the green coloring, because it was rooted at the other red occurrence of $q_m.x.s.y$. Find such a match for C'' that is green, and call it $\mu_g(C'')$. Now build μ_3 and μ_4 to agree with μ_1 and μ_2 respectively on everything that is not in C''. On C'', both pick up $\mu_g(C'')$ instead. These matches are completely green and agree on all of C; note that on $C \setminus C''$ they are unchanged and on C'' they agree by identity. They are also valid matches, because neither t nor anything in $C \setminus C''$ can have $q_m.x.s.y$ as prefix, otherwise μ_1 and μ_2 could not have been green on those paths either. Thus we can splice in the match for C'' from the sibling subtree, and produce valid matches. Again, those matches are available for us to use at this point in the chase, and we can use them to unify $\mu_1(t)$ with $\mu_2(t)$ instead.

The only remaining subcase is when $\mu_1(C'')$ is green but $\mu_2(C'')$ is not (note that the choice of which match is not green is arbitrary). We claim that we can go into $\mu_2(C'')$'s sibling subtree as in the previous case and pick up a $\mu_g(C'')$ that we can swap for $\mu_2(C'')$. However, we must show that we can find such a $\mu_g(C'')$ with the property that $\mu_g(c'') = \mu_1(c'')$ for each $c'' \in C''$, which is not obvious. Fist we deal with the case where μ_1 and μ_2 actually agree by identity on $q_m.x.s$, the red fork point, and fork only at $q_m.x.s.y$; in this case we can make μ_g agree with μ_1 by identity. In the other remaining case, $\mu_1(C'')$ is somewhere else in the tree. This is the most complicated subcase and requires the help of a separate sublemma.

Sublemma 1 *We know that μ_2 and μ_g agree on $q_m.x.s$; call this path u for short. We also know μ_1 and μ_2 fork properly above u, say at some u' which is an ancestor of u. So we have $q_m \prec u' \prec u \prec c''$ for all $c'' \in C''$. If for some $c \in C''$ chasing with Σ_a on T'' unifies $\mu_1(c'')$ and $\mu_2(c'')$, then there are further unifications that can immediately be performed on the green subtableau to unify $\mu_1(c'')$ and $\mu_g(c'')$.*

Proof. This argument is also by induction on the order in which the $\mu_1(c'')$ and $\mu_2(c'')$ were unified by the chase on T''. Each c'' was at some point unified using some dependency $D \to c''$. Some of the D's involved in the unifications may themselves have contained attribute paths d that had u as a prefix. If this was the case, for these $D \to c''$ to fire, there must also have been earlier unifications between the values at the end of two specific occurrences of d, one consistent with $\mu_1(c'')$ and the other with $\mu_2(c'')$. Identify all such d, and extend μ_1 and μ_2 to include them. μ_1 and μ_2 are now matches not just for C'', but also for some extra d paths also having u as a prefix. Let Q be the set containing all of C'' and the relevant d's. We show the lemma statement holds for any $q \in Q$ — as often happens, we find we need to prove a stronger statement for a stronger inductive hypothesis.

This new proof is by induction on the order in which the chase on T'' unifies the values in Q. We assume the statement is true for all previous unifications, if any, and follow on the next $q \in Q$. Suppose we unify $\mu_1(q)$ and $\mu_2(q)$ via an XFD $E \to q$. This unification occurred via two matches μ_3 and μ_4 that agree on E, and where $\mu_3(q) = \mu_1(q)$, $\mu_4(q) = \mu_2(q)$ by identity. If u' is not a prefix of any path in E, our task is easy. Build μ_5 and μ_5, matches for $E \cup \{ q \}$, such that $\mu_5(q) = \mu_g(q)$, $\mu_6(q) = \mu_1(q)$ by identity. For E, take a completely green match that is consistent with $\mu_3(u') = \mu_4(u')$, and make μ_5 and μ_6 pick up this E match; this match exists because the green tree is uniformly matched. We see that μ_5 and μ_6 are valid matches, they are completely green, and they can serve to unify $\mu_g(q)$ with $\mu_2(q)$ using $E \to q$.

The second possibility is that some $E' \subseteq E$ has u' as a prefix, but nothing has u as prefix. Here we build our two matches as follows. We set $\mu_5(q) = \mu_g(q)$, $\mu_6(q) = \mu_1(q)$ by identity as before, and for all $e' \in E'$ we take $\mu_5(e') = \mu_4(e')$ and $\mu_6(e') = \mu_3(e')$ by identity. For the remaining $e \in E \setminus E'$, we take a green match as before. Again these are two valid fully green matches that allows to unify $\mu_g(q)$ with $\mu_1(q)$ using $E \to q$.

The third case is when some $E'' \subseteq E'$ has u as a prefix. If any of these paths in e'' are internal nodes, the unification of $\mu_3(q)$ and $\mu_4(q)$ could not have happened. On the other hand, if they are attribute paths, we see that they are all in Q. Note that $\mu_3(e'') = \mu_4(e'')$ for all $e'' \in E''$, but that these two values were different initially, as we showed in Lemma 19 that they were not both red. Therefore, we know that we must have had $\mu_3(e'')$ unified with $\mu_4(e'')$ at some earlier stage of the chase. So we can apply our inner inductive hypothesis to assume we have a fully green chase sequence to unify all of $\mu_g(E'')$ with $\mu_1(E'')$; perform this chase sequence,

At this point build μ_5 and μ_6 in the obvious manner. Proceed as in the previous case except for E'', where we take $\mu_5(E'') = \mu_g(E'')$ and $\mu_6(E'') = \mu_1(E'')$ by identity. Again these are valid fully green matches which allow to unify $\mu_g(q)$ with $\mu_1(q)$ using $E \to q$. This completes the proof of our sublemma.

We can now use Sublemma 1 to complete the last case of Lemma 21. Take μ_g to be any match for C'' in the sibling subtree. Via the procedure in Sublemma 1, produce a green subtableau chase to ensure that $\mu_g(c'')$ and $\mu_1(c'')$ are unified for all $c'' \in C''$. We can now splice in $\mu_g(C'')$ instead of our non-green $\mu_2(C'')$ to produce two green matches that trigger a unification of our two crucial occurrences of t, and we are done.

P.2.3.4 THE CHASES CAN BE REORDERED. The fourth and fifth parts of our proof outline in **P.2.3** follow from a corollary of Lemma 21.

Corollary 5. *We can reorder the chase on T'' with Σ_a as follows.*

- *Start by performing a green subphase of the chase on the green subtableau. That is, perform all unifications affecting pairs of green attribute nodes, by redirecting the original chase as suggested in the last lemma. Continue until the completely green tuples all satisfy Σ_a.*
- *Finish by making any further unifications still required by Σ_a between pairs of nodes which are not both green. The green subtableaux before and after this second pass must embed into each other.*

P.2.3.4 WITNESSES EXIST IN THE GREEN SUBTABLEAU. There is one last lemma to prove in order to finish the proof of Proposition 8.

Lemma 22. *If the chase on T'' threw an error in phase 3 as originally stipulated due to some $A \to b$, then there were witnesses against $A \to b$ within the green subtableau.*

Proof. Consider the original witnesses μ_1 and μ_2. We showed earlier, in Lemma 20, that both $\mu_1(b)$ and $\mu_2(b)$ are green. If for any $a \in A$ either or both of the matches are not green, we can produce a green pair of matches — exactly as we did in the proof of Lemma 21 — that still witness $A \to b$ does not hold.

This lemma tells us that identical witnesses must exist in the blue tree in T'. Therefore, Phase 3 on T' would have found the blue witnesses against $A \to b$, and returned "no". This gives us the required contradiction and completes the proof of Proposition 8. This also completes the proof of Theorem 15.

Theorem 15 gives us one direction of our correctness result in Theorem 16. The other direction is exactly the same proof given for simple DTDs.

Theorem 16. *After the chase terminates, T'' as defined above satisfies σ if and only if $\Sigma \models \sigma$.*

6.4 Axiomatization

To axiomatize implication for #-DTDs, we need to account for the two pathological cases described above. Formally, the pathological cases arise because the DTD prevents certain XFDs from holding unless all matches for the left-hand side are null. Again, this requires the introduction of an axiom schema.

Theorem 17. *In the presence of a simple DTD D, a sound and complete set of axioms for (unrestricted) XFD implication includes exactly Axioms 1-21 from Section 5.4 with*

22. *If $p.x \overset{X}{\to} p.x.y$, then for $W \overset{X,p.x}{\to} Z$, for all W, Z and all $x \rightsquigarrow yy\gamma \in D$.*

Proof. We first show soundness of the new axiom. Because of the DTD production $x \rightsquigarrow yy\gamma$, we know that any document T satisfying D containing at least one occurrence of $p.x$ contains two occurrences of $p.x.y$ below it. Consequently, if $U(R(T))$ contains some tuple t such that t is nonnull on X and on $p.x$, then $U(R(T))$ must contain some other tuple t' which agrees with t on $p.x$ but not on $p.x.y$ (t' need not itself be nonnull on all of X, but this is not necessary for our agrument). Thus, if there is any such $t \in U(R(T))$, then $p.x \overset{X}{\to} p.x.y$ does not hold on $U(R(T))$ (t and t' are witnesses to that fact). Consequently if we know that $p.x \overset{X}{\to} p.x.y$ holds on some T, we know $U(R(T))$ contains no tuple that is nonnull on both X And $p.x$. Thus, $W \overset{X,p.x}{\to} Z$ is vacuously true on $U(R(T))$, because it is a statement about an empty subset of tuples of that relation.

For completeness, assume $\sigma = A \to B$, for $A, B \subseteq$ paths(D), $B = \{\, b_1, \cdots b_k \,\}$. We give only the derivation for $A \to b_i$ for arbitrary $b_i \in B$, which is sufficient. As discussed in the proof of the chase for #-DTDs, there are two possibilities: either there is no tree satisfying Σ and D and containing a single nonnull occurrence of $\varphi(A \to b_i)$, or there is such a tree. To build a derivation to show $\Sigma \vdash_D A \to b_i$ using axioms $1 - 22$, we start by deciding which of the above is true; we can do so effectively and indeed reasonably quickly, as explained in Theorem 13.

If the first case is true, we obtain the derivation for $\Sigma \vdash_D q.x \overset{U}{\to} q.x.y$ from the proof of Theorem 13. Here U is the subset of paths(D) such that $nn(A \cup \{b_i\}, U)$. We now complete the derivation as follows.

- $q.x \overset{U}{\to} q.x.y$ (from the proof of Theorem 10)
- $A \overset{U}{\to} b_i$ (Axiom 22 and the fact that $q.x \in U$)
- $nn(A \cup \{b_i\}, U)$ (by definition of U above)
- $A \overset{A\cup\{b_i\}}{\to} b_i$ (Axiom 13)
- $A \to b_i$ (Axiom 12)

On the other hand if the second case is true, we obtain the derivation for $A \to b_i$ by induction on the steps of a chase algorithm, exactly as in the case for simple DTDs in Theorem 9. That derivation does not use Axiom 22, so the proof genuinely carries over verbatim.

6.5 Reasoning about FD implication with first-order hereditary Harrop clauses

Unfortunately for #-DTDs, our previous encoding of axioms into Horn clauses no longer works. However, a generalization of Horn clauses that is known in the proof theory community, and which is still useful for automated theorem provers: *first-order hereditary Harrop clauses* [9]. We can extend our language fragment to these new clauses to obtain an analogous encoding.

Theorem 18. *Given a simple DTD D and an instance of an implication problem $\Sigma \vDash_D \sigma$, suppose we define a set $\Gamma_{\sigma,D}$ of first-order formulae over $\{R, N\}$ to contain exactly the following.*

- *All of $\Gamma_{\sigma,D}$ as used in the encoding for simple DTDs.*
- *The following clause for all $p.x \in paths(D)$ such that D contains a production $x \rightsquigarrow yy\gamma$, $\gamma \in \text{EL}^*$ and every X that is a subset of U:*

$$H(p.x \overset{X}{\to} p.x.y) \to \big(\forall p_1\, \overline{x_1}\, \overline{s_1} R(p_1, \overline{x_1}, \overline{s_1}) \wedge N(\overline{x_1}) \wedge N(p_1) \to \bot\big)$$

where p_1 is the variable for $p.x$, $\overline{x_1}$ are the variables for X, and $\bot$ is the constant $FALSE$.

Then $\Gamma_{\sigma,D} \vdash H(A \overset{A,b_i}{\to} b_i)$ for all $b_i \in B$ if and only if $\Sigma \vDash A \to B$.

The second type of sentence is a nested Horn clause of the type known as first order hereditary Harrop formulas. Our formulation assumes that $p.x \notin X$. However, if it is, an alternate encoding is easy to give, so we do not concern ourselves with this issue.

Proof. The forward direction is again clear from the soundness argument we gave for Axiom 22 and our previous proofs with Horn clauses. For the backward direction, we show how to encode the inference in Axiom 22.

For readability, we give the argument on the assumption that W, X and Z are all disjoint and none of these sets includes $p.x$. We also assume that there are no other attributes in R. It is easy to see how to modify our argument in case any of the above are not true.

The left-hand side of the conclusion of Axiom 22 gives us an $\overline{w_i}, \overline{z_i}, \overline{x_i}, p_{x,i}, p_{xy,i}$ or $i = 1, 2$ such that

$$\bigwedge_{j=1,2} R(\overline{w_j}, \overline{z_j}, \overline{x_j}, p_{x,j}, p_{xy,j} \wedge N(\overline{x_1}) \wedge N(p_{x,1}) \wedge \overline{w_1} = \overline{w_2}$$

One of the premises of Axiom 22 is $H(p.x \xrightarrow{X} p.x.y)$. This premise allows us to use the last type of clause we added into $\Gamma_{\sigma,D}$ to infer

$$\forall p_{x,1}, \overline{x_1}, \overline{s_1} R(p_{x,1}, \overline{x_1}, \overline{s_1}) \wedge N(\overline{x_1}) \wedge N(p_{x,1}) \rightarrow \bot$$

But we have a $N(p_{x,1})$ and a $N(\overline{x_1})$ above, so we can infer $\bot$ directly. From this we are free to infer anything, including the statement most relevant to us, namely $\overline{z_1} = \overline{z_2}$, and we are done.

Interaction of XFDs with the general theory of trees We can formulate the appropriate analogue of Corollary 1 for the theorem above. Given a #-DTD D, let Ψ_D denote all first-order formulae that hold on all trees (finite or infinite) satisfying D. We let $\mathcal{F}$ be as before, and let $\mathcal{HA}$ be all first-order hereditary Harrop formulae that are true on all trees satisfying D. Then we have the following result.

Corollary 6. *For any $\Sigma \subseteq \mathcal{F}, \sigma \in \mathcal{F}$, $\Sigma \cup \Psi_D \vdash \sigma$ if and only if $\Sigma \cup (\Psi_D \cap \mathcal{HA}) \vdash \sigma$*

Interestingly enough, we can indeed exhibit many formulae that belong to the theory of trees satisfying a particular $\#$-DTDs that are truly irrelevant to functional dependency implication. For instance, given a DTD D containing a production $a \rightsquigarrow$ *bbbb*, it is part of the theory of all trees satisfying D that "all a nodes have exactly four b children". However, as we see from our implication algorithms and axiomatization, the number four is irrelevant; for the purposes of implication of FDs, a DTD identical to D but with $a \rightsquigarrow$ *bbbb* replaced by $a \rightsquigarrow$ *bbbbb* would behave exactly the same way. We conjecture that in fact, formulae such as the one just presented cannot be expressed as first-order hereditary Harrop clauses, and consequently that our result above explains formally why they are not relevant.

7 XFDs with $\vee$-DTDs and Arbitrary DTDs

As we shall see, the presence of disjunction in a DTD results in very substantial complications to all the problems we investigate in this paper. In what follows, we present the results that we have, but we also try to give some intuition as to why the extra complications ensue.

7.1 Consistency

Like simple DTDs, it is not possible for a $\vee$-DTD to be inconsistent with a set of XFDs.

Theorem 19. *If D is a $\vee$-DTD, then (Σ, D) is consistent for all Σ.*

Proof. The argument for this is the same as for simple DTDs. A $\vee$-DTD cannot force two occurrences of the same path, so the minimal tree satisfying D contains only one occurrence of each path. Therefore, by the definition of a witness, it cannot contain a witness against some dependency.

On the other hand, for arbitrary DTDs, the consistency problem unfortunately becomes NP-hard. This can be shown by adapting the reduction given in Section 7.3 of Arenas and Libkin [2].

Theorem 20. *Deciding the consistency of (Σ, D), for arbitrary D, is NP-hard.*

Proof. We give a reduction of SAT-CNF to consistency. Given a CNF formula ϕ of the form $C_1 \wedge C_2 \wedge \cdots C_k$, where each C_i is a clause. Assume that ϕ uses the variables $x_1, \cdots x_m$. Build the following D and Σ.

For D, the label alphabet involved is ρ, C and $\{\, C_{ij} \mid C_i$ mentions literal $x_j\, \} \cup \{\, \overline{C_{ij}} \mid C_i$ mentions literal $\overline{x_j}\, \}$. For every C_i, define $f(C_i)$ as

$$f(C_i) = C_{i,j_i}|\cdots|C_{i,j_p}|\overline{C_{i,k_i}}|\cdots|\overline{C_{i,k_q}}$$

where the set of literals in clause C_i is exactly $x_{j_i}, \cdots, x_{j_p}, \overline{x_{k_1}}, \cdots, \overline{x_{k_q}}$. Given this definition, our DTD has the single production $\rho \rightsquigarrow f(C_1)f(C_2)\cdots f(C_k)CC$

For our set Σ, we define it to contain exactly the dependencies $\{\rho.C_{ij}, \rho.\overline{C_{kj}}\} \rightarrow \rho.C$, where each $C_{ij}, \overline{C_{kj}}$ are in our language. We claim that (Σ, D) is consistent if and only if ϕ is satisfiable.

First, suppose (Σ, D) is consistent. Then there is a tree T satisfying Σ and D. Define a truth assignment to the x_j variables as follows. If in T the root has a child with label C_{ij} for some i, then set x_j to true, otherwise set it to false. We show that each clause C_i is be satisfied.

For a given C_i, the ith child of the root in T is either some $C_{i\ell}$ or some $\overline{C_{i,\ell}}$. In the former case, by definition x_ℓ received the value true. As we know that x_ℓ occurs positively in C_i, C_i is satisfied by this x_ℓ. On the other hand, suppose the child is $\overline{C_{i\ell}}$. Suppose for a contradiction that x_ℓ received the value true in the assignment. In this case, there is some j such that in T, the root has a child $C_{j\ell}$. On the other hand, we now have in T a nonnull match for $C_{j\ell}, \overline{C_{i\ell}}$. This match extends to two different matches for $\rho.C$, so the dependency $C_{j\ell}, \overline{C_{i\ell}} \rightarrow \rho.C \in \Sigma$, does not hold, a contradiction. So x_ℓ must have received the value false in the assignment. Therefore, as x_ℓ occurs negated in C_i, C_i is satisfied in this case too.

Now suppose ϕ is satisfiable. Then we build a tree T as follows. The root of T is ρ and has the two required C children. For every clause C_i, we add the ith child to the root. We choose a literal x_ℓ or $\overline{x_\ell}$ in C_i such that the truth assignment that satisfies ϕ makes the literal true. If the literal is of the form x_ℓ, add a child $C_{i\ell}$, otherwise add $\overline{C_{i\ell}}$. It is clear that T satisfies$D$. To see it satisfies Σ, observe that we can never add two children $C_{j\ell}$ and $\overline{C_{k\ell}}$ for any ℓ in the construction of T, because that would mean the truth assignment makes both x_ℓ and $\overline{x_\ell}$ true in two different clauses. Thus no dependency in Σ finds in T a nonnull match for its left-hand side, and consequently all of Σ is satisfied.

Theorem 20 is slightly disappointing because it is only a lower bound. It is still open whether consistency checking is even decidable. Naïvely, it would seem that we could decompose an arbitrary DTD into a disjunction of #-DTDs and check the prime models for each component DTD. However, the #-DTD algorithm does not work unadapted in this case. While it is still true that there must be a finite set of minimal trees such that every T in the set satisfies D and every T' satisfying D structurally embeds one such T, it is no longer the case that all such minimal T are uniformly matched. As a consequence, it is perfectly possible to have a DTD D such that none of the prime models has an assignment that satisfies Σ, but another, bigger tree that embeds one of the prime models does have such an assignment. For example, consider the DTD D containing the productions $\rho \rightsquigarrow cc, c \rightsquigarrow af|bf^*, f \rightsquigarrow \alpha$, and the set of XFDs $\Sigma = \{\, \rho \to \rho.c.a, \rho \to \rho.c.b, \rho \to \rho.c.f.\alpha \,\}$. One tree T that satisfies both D and Σ is shown in Figure 7. Note that the prime model that structurally embeds in this tree is not uniformly matched and has no assignment satisfying Σ, as there is no f branch for the second copy of c; however, there is an extension of this prime model which does have a satisfying value assignment. Moreover, the other two prime models cannot be extended to any tree satisfying Σ.

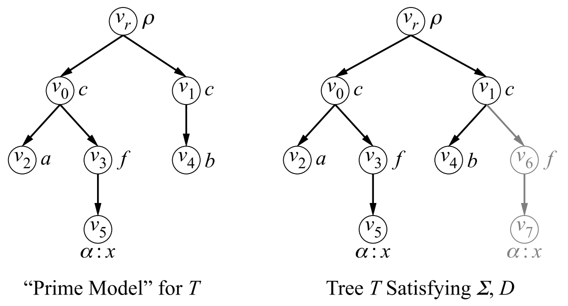


Fig. 7. A Minimal Tree for Σ, D and its Uniformly Matched Extension

7.2 Implication complexity

Arenas and Libkin [2] have already shown that implication is infeasible once the DTD contains disjunction. In particular, implication for $\vee$-DTDs is co-NP-complete. Implication for arbitrary DTDs remains open, though it is obviously co-NP-hard. Because arbitrary DTDs include #-DTDs, the implication problem for arbitrary DTDs must includ consistency checking as well. However, we have already demonstrated that extending our consistency checking techniques to arbitrary DTDs is difficult.

7.3 Axiomatization

We have strong reason to believe that, in the presence of disjunction, the question of axiomatization becomes substantially harder. Arenas and Libkin [2] have in some sense shown that no finite axiomatization is possible. But this result is limited to their axiom language, which is less rich than ours; this result need not carry over to all possible axiom languages.

8 Related work

In the relational model, functional dependencies [1] and chase algorithms [1, 4] have been studied extensively. Our axiomatization draws on the theory developed for relational FDs in the presence of null values [3, 16]. Functional dependencies have also been studied for nested relations [12].

In XML, functional dependencies have attracted much research attention [2, 20, 14, 19, 15, 13]; see [21] for a survey. Our work uses a definition of XFD that is equivalent to the one in [2]; we have highlighted throughout the paper how our results extend these. The other FD definitions in the literature differ from ours in several ways, such as taking a stronger semantics for XFD satisfaction in the presence of null values [20], the use of different equality relations for node comparison, and the language used to identify the nodes involved in functional dependencies. The literature also makes varying assumptions about the documents on which the FDs are to hold: some work assumes that there is a DTD present in all cases [2], while other definitions do not require one [20, 14].

However, we believe that our definitions allow us to strike a good balance between generality of the framework and the ability to obtain strong tractability and axiomatization results. It is worth noting that much of the framework we developed can also be reused if a different definition of XFDs is desired: for instance, to reason about XFDs with a stronger null value semantics, we can retain our equivalence of XFDs to relational FDs on $U(R(T))$ and use relational work on these other null value semantics [16] to assist with the development of axiomatizations and algorithms.

Finally, there has been much work on other constraints in XML documents, notably keys [6], foreign keys [11], path constraints [7] and XICs [8]. For a recent, general survey on XML constraints, see [10]. DTDs are known to interact in a complex fashion with keys [11] and to affect the satisfiability and containment of XPath queries [18, 5]. However, we are not aware of any study of the interaction between DTDs and FDs.

9 Future directions

	Consistency	Implication	Axioms	Language Fragment
No DTD	N/A	Linear	1-19	Universal Horn theory
Simple DTD	Trivial	Linear	1-21	Universal Horn theory
#-DTD	Quadratic	Quadratic	1-22	FO Hereditary Harrop theory
$\vee$-DTD	Trivial	co-NP-complete ([2])		
Arbitrary DTD	NP-hard	co-NP-hard ([2])		

Fig. 8. Summary of our results

Several areas emerge as obvious directions for future work at this point. The first, of course, is filling in the missing entries in our summary matrix, such as the upper bounds for both consistency and implication with an arbitrary DTD. We conjecture that both problems are decidable. Furthermore, the language used in the definition of XFDs can be made richer, for instance by adding wildcard descendant navigation to our tree patterns, as in [17]. Also, our investigation of XFD interaction with DTDs can be broadened to include other popular XML constraint specifications, such as XML Schema.

10 Acknowledgments

We would like to thank Al Demers, Johannes Gehrke, Dexter Kozen and David Martin for valuable discussions and comments.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. M. Arenas and L. Libkin. A normal form for XML documents. *ACM TODS*, 29(1):195–232, 2004.
3. P. Atzeni and N. Morfuni. Functional dependencies and constraints on null values in database relations. *Information and Control*, 70(1):1–31, 1986.
4. C. Beeri and P. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM TODS*, 4(1):pp. 30 – 59, 1979.
5. M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *Proc. PODS*, pages 25–36, 2005.
6. P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about keys for XML. *Inf. Syst.*, 28(8):1037–1063, 2003.
7. P. Buneman, W. Fan, and S. Weinstein. Path constraints in semistructured and structured databases. In *Proc. PODS*, pages 129–138, 1998.
8. A. Deutsch and V. Tannen. Containment and integrity constraints for XPath. In *KRDB*, 2001.
9. D.Miller. Sequent calculus and the specification of computation. http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/scsc.pdf.
10. W. Fan. XML constraints: Specification, analysis, and applications. In *Proc. DEXA*, pages 805–809, 2005.
11. W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. In *Proc. PODS*, 2001.
12. C. Hara and S. Davidson. Reasoning about nested functional dependencies. In *Proc. PODS*, pages 91–100, 1999.
13. S. Hartmann and S. Link. More functional dependencies for XML. In *ADBIS*, pages 355–369, 2003.
14. S. Hartmann and S. Link. On functional dependencies in advanced data models. In *Electronic Notes in Theoretical Computer Science*, volume 84. Elsevier Science B. V., 2003.
15. M. Lee, T. Ling, and W. Low. Designing functional dependencies for XML. In *Proc. EDBT*, pages 124–141, 2002.
16. M. Levene and G. Loizou. Axiomatisation of functional dependencies in incomplete relations. *Theor. Comput. Sci.*, 206(1-2):283–300, 1998.
17. G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proc. PODS*, pages 65–76, 2002.
18. F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *Proc. ICDT*, pages 315–329, 2003.
19. K. Schewe. Redundancy, dependencies and normal forms for XML databases. In *ADC*, pages 7–16, 2005.
20. M. Vincent, J. Liu, and C. Liu. Strong functional dependencies and their application to normal forms in XML. *ACM TODS*, 29(3):445–462, 2004.
21. J. Wang. A comparative study of functional dependencies for XML. In *APWeb*, pages 308–319, 2005.