Representing and Querying XML with Incomplete Information[*]

SERGE ABITEBOUL and LUC SEGOUFIN
INRIA, Paris 11

VICTOR VIANU
U.C. San Diego

September 15, 2005

Abstract

We study the representation and querying of XML with incomplete information. We consider a simple model for XML data and their DTDs, a very simple query language, and a representation system for incomplete information in the spirit of the representations systems developed by Imielinski and Lipski for relational databases. In the scenario we consider, the incomplete information about an XML document is continuously enriched by successive queries to the document. We show that our representation system can represent partial information about the source document acquired by successive queries, and that it can be used to intelligently answer new queries. We also consider the impact on complexity of enriching our representation system or query language with additional features. The results suggest that our approach achieves a practically appealing balance between expressiveness and tractability.

1 Introduction

In a warehouse for XML data – that we call an (XML) Webhouse – information is collected from Web sites and stored in a centralized fashion. In practice, the information held in a Webhouse is never complete. This is due to many reasons: limited storage capacity, the dynamic nature of Web data, expiration of data, etc. Thus, Webhouses have to deal with *incomplete information*. We view here the Webhouse as an incomplete repository of XML documents that is continuously enriched by exploration of Web sources, in response to queries or by crawling the Web. Documents may be entirely missing or may be partially available.

At any given time, the Webhouse contains a *representation* of incomplete information about XML documents satisfying given Document Type Definitions (DTDs). The incomplete information about the XML documents is *enriched* using answers to queries against remote documents. We assume that sources are either static, or that the accumulation of information occurs in-between consecutive source updates. Thus, when a source is modified, the information concerning it is reinitialized to its DTD, unless information about the update is available that allows salvaging some of the previously accumulated information.

When a query is posed against the Webhouse, two courses of action are possible:

1. The first alternative is to answer the query as best possible using the incomplete information already available. Since the data is not entirely known, the answer is not always complete. We represent the answer using the same representation as the one we use to

[*]Work supported in part by the National Science Foundation under grant number INT-0334764

describe our incomplete knowledge of the sources. Two important variations are the *sure* and *possible* answer modalities, i.e., providing the pieces of information that *surely* hold in all possible answers, or similarly those that *possibly* hold.

2. The second, similar to a mediator approach, is to seek from the sources the additional information needed to fully answer the query. In this case, we would like to use the incomplete information as a guide for determining what additional exploration of Web sources is needed by taking as much as possible advantage of the data already available.

This paper introduces *representations* for incomplete information, studies their incremental *maintenance* and addresses the issue of *answering queries* posed against an incomplete Webhouse.

The quest for simplicity and efficiency was the main motivating factor in our choice of model, and has led to many limitations. The most notable are (i) a very limited query language based on pattern matching and simple selection conditions on data values, (ii) the use of simplified DTDs that ignore the ordering of components in an element, and (iii) the assumption that XML elements have persistent identifiers. Despite its limitations, we believe that our framework captures a broad range of situations of practical interest. Our examples illustrate some of them.

The representation of incomplete information is quite natural. It uses partial XML trees to represent the data available, and typing information in the style of DTDs to represent the data that is still missing. The typing we use for the missing data is interesting in its own right and reminiscent of some extensions already proposed for DTDs. These include specifying ranges for some data values, e.g., *price* $\geq$ *100*, and a *specialization* mechanism that allows defining the type of a given element name depending on the context where it appears. We call such a representation an *incomplete tree*. As illustrated by our examples, incomplete trees exhibit in a user-friendly way the partial information available as well as the missing information, and can be itself naturally represented and browsed as an XML document.

We show that, given a simplified DTD satisfied by the input and a sequence of query-answer pairs on the input, the partial knowledge about the input can be represented by an incomplete tree which can be maintained incrementally in PTIME. Given a query and an incomplete tree, the set of possible answers to the query can again be represented by an incomplete tree computable in PTIME (for fixed set of labels). In particular, this shows that incomplete trees form a strong representation system with respect to our queries. Furthermore, it can be checked in PTIME whether a given query can be fully answered with the data currently available. For the case when the available data is not sufficient, we provide a PTIME algorithm that uses the incomplete tree to determine which additional information is needed from the sources in order to fully answer the query, and provides a nonredundant set of queries for retrieving the information.

Although incomplete trees can be incrementally maintained in PTIME, their size can grow exponentially in the overall sequence of query-answer pairs. We discuss several ways of dealing with the exponential blowup. We consider an extension to incomplete trees, called *conjunctive*, that intuitively adds a form of alternation. With this extension, the size of incomplete trees is shown to remain polynomial with respect to the entire sequence of query-answer pairs. However, many of the manipulations needed in handling incomplete information now become exponential in the representation. For example, checking emptiness of conjunctive trees becomes NP-hard, whereas it is in PTIME for regular incomplete trees. As an alternative approach, we exhibit a restriction of the input DTD and the queries ensuring that incomplete trees remain polynomial in the overall sequence of query-answer pairs. Thus, all manipulations remain polynomial.

Regardless of the complexity-theoretic bound, we exhibit two approaches for dealing with cases when the incomplete tree grows too large to be practical. The first approach consists of asking a small set of additional queries chosen so as to provide precisely the critical information needed to eliminate some of the unknown information and shrink the incomplete tree. We prove that the queries can be chosen such that the incomplete tree remains of polynomial size with respect to the entire sequence of query-answer pairs and input DTD. This approach can be used heuristically whenever needed. The second approach, discussed informally, is a heuristic for gracefully loosing some of the information represented in the incomplete tree, thus allowing a trade off of accuracy against size in incomplete trees. Once again, we show that this approach can be used to keep the incomplete tree polynomial in the sequence of query-answer pairs.

We argue that our core model provides a practically appealing starting point for dealing with incomplete information in XML Webhouses. However, the model has many limitations. We discuss the impact of various extensions to the model, and show that even minor extensions lead to significant difficulties. The representation system may no longer exhibit in a user-friendly way the partial information available, or there may no longer be a strong representation system. Most seriously, various decision problems, such as whether a query can be answered given the information currently available, have very high complexity or become undecidable. Some of the extensions concern the query language: the extra features include optional and negative subtrees in query patterns, constructed answers, recursive path expressions, data joins, and powerful restructuring modeled by k-pebble transducers [34, 35]. We also discuss other extensions to the framework such as the persistent ids assumption and the issue of order.

The paper is organized as follows. Section 2 introduces our formal model for XML, DTD, the various types we use, as well as the representation system. Section 3 deals with the acquisition and the use of incomplete information, and with the various approaches to the exponential blowup of incomplete trees. Section 4 discusses extensions and associated complexity and undecidability results. The paper ends with brief conclusions.

Related work. Incomplete information has been of interest early on in database systems [18]. Much of the focus has been on searching for the “correct” semantics for queries applied to incomplete databases [43, 38, 42]. Usually, the semantics of incompleteness is approached from two perspectives, either a *closed world assumption* (CWA) or an *open world assumption* (OWA). Intuitively, CWA states that nothing holds unless explicitly stated in the incomplete database, whereas OWA states that anything not ruled out is possible. Interestingly, incomplete trees reconcile the two approaches by allowing a combination of the two semantics. They allow to describe with flexible precision the missing information, by stating that some facts are not in the document (CWA) but also that some data still ignored may exist (OWA).

A landmark paper by Imielinski and Lipski [26] laid the formal groundwork for incomplete databases with nulls of the “unknown” kind, and introduced the notion of strong representation system. The representation system we use is in the spirit of the c-tables of Imielinski and Lipski [26], but addresses a tree model instead of the relational model. Most importantly, we use a more benign form of incompleteness, which is possible because our query language is more restricted than the relational algebra they consider (e.g., it has no data joins).

The complexity of handling incompleteness was studied in many works [4, 26, 42]. The program complexity of evaluation is usually higher by an exponential than the data complexity [19, 41]. This was first noted in the early 80s [24, 30], as part of the study of nulls in weak universal instances. Updating incomplete information was investigated by Grahne [22].

The Webhouse scenario that we consider here is in the spirit of data warehousing (see e.g., [28]). The simple query language we use is closely related to *tree patterns*, which form the core of the XPath language [7, 32]. Our queries are tree patterns specifying direct children, but not descendants. Also, tree patterns select just one node, whereas our queries select all the nodes involved in the pattern, thus yielding a prefix of the input tree.

Extensions of DTDs with specialization have been considered under various names and in various contexts [9, 17, 36]. XML Schema provides a specialization mechanism that allows decoupling element tags from element types. We use the specialization mechanism in our representation system.

Perhaps closest to our work is an investigation by Kanza, Nutt and Sagiv [27], which studies incomplete information in semistructured data. However, their framework and results are quite different. Incomplete information in semistructured data is also considered by Calvanese, De Giacomo, and Lenzerini [11]. Their work considers a schema mechanism for semistructured data extending the classical approach based on graph simulation. In the extended model, the schema is a graph with formulas associated to its edges. The formulas are expressed in a particular description logic, and can express both constraints and incomplete information. The results concern the complexity of checking subsumption among such schemas. The problem of querying databases with incomplete information is not considered.

In the context of XML, incomplete information resulting from repairs to XML documents violating functional dependencies is considered by Flesca et al. [20]. The focus is on computing the certain answers to queries on the set of possible repairs.

An XML extension with embedded service calls to Web services is defined by Abiteboul et al. [33]. The Web service calls return XML documents that may in turn contain other service calls. This model is called Active XML (AXML). An AXML document can be thought of as a mix of extensional and intensional data (the intensional part consists of the service calls). In this sense, it is similar in spirit to our incomplete trees. An answer to a query against an AXML document may be fully extensional XML document (if the input document contains enough information to answer the query) or it may contain service calls that may be needed to fully answer the query. The problem of answering a query on an AXML document using a minimum number of service calls has also been considered [1].

Answering queries using views is related in spirit to our investigation, since views are one way of providing incomplete information on the underlying database. This issue has been extensively investigated for relational databases, especially in the data integration context [29, 16, 37]. (See also the survey by Halevy [23].) The use of a model based on incomplete information to study this problem is proposed by Abiteboul and Dutschka [2]. The management of incompleteness due to data expiration is studied by Molina et al. [21]. In the context of semistructured data, answering queries using views has been investigated for regular path expressions [12, 14, 13, 15]. In the context of XML, the use of materialized XPath views stored in a semantic cache to answer XPath queries has been studied [8, 31]. The focus is on detecting when the query can be fully answered using one of the indivdual materialized views, and finding a rewriting of the query in terms of such a view. The use of views in combination, or the case when the views provide incomplete information on the answer to the query, are not considered.

The XStreamCast project (http://lambda.uta.edu/XStreamCast) considers the efficient processing of XQuery queries on streaming XML documents. This involves using fragments of XML documents that provide partial information about the input. One important issue addressed is determining whether a fragment is relevant to the query being processed [10].

The present article is an expanded and updated version of a conference paper by the same authors [5].

2 Formal Framework

We next present our core framework for Webhouses with incomplete information. We define in turn our model of XML documents and simplified DTDs, queries, and the representation system for incomplete information.

Data trees. Our formal model abstracts XML documents as labeled trees. Our abstraction simplifies real XML documents in several ways, some of which are minor and others more substantial. For example, the model does not distinguish between attributes and subelements, a distinction often considered cosmetic. A more significant simplification is that our trees are unordered, whereas XML documents are ordered. We will discuss the issue of order in Section 4.

We assume given an infinite set $\mathcal{N}$ of *nodes*; a finite set Σ of *element names* (labels); and a set $\mathbb{Q}$ of *data values*. We denote element names by $a, b, c...$, nodes by n, data values by v, possibly with sub-and superscripts. We denote sets of labels by $A, B, C, ...$ For simplicity, we assume that the set $\mathbb{Q}$ of data values is the rational numbers (the integers or reals would do just as well). Our simplified model for XML is defined next.

Definition 2.1 *A* (data) tree *over Σ is a triple $\langle t, \lambda, \nu \rangle$, where:*

1. *t is a finite rooted tree with nodes from $\mathcal{N}$;*
2. *λ, the labeling function, associates a label in Σ to each node in t; and*
3. *ν, the data value mapping, assigns a value in $\mathbb{Q}$ to each node in t.*

Data trees are denoted by $T, T', \ldots$. We will use the notion of *prefix* of a data tree. Since in our framework node identifiers are significant, we are led to parameterize the notion of prefix by a set of nodes. Specifically, let $T = \langle t, \lambda, \nu \rangle$, $T' = \langle t', \lambda', \nu' \rangle$ be data trees and $N \subset \mathcal{N}$ be a set of nodes. The data tree T' is a prefix of T relative to N if there is a mapping h from the nodes of t' to those of t such that:

- h is one to one;
- $h(n) = n$ for every node n of t' that is in N;
- h maps the root of t' to the root of t;
- if n_1 is the parent of n_2 in t', then $h(n_1)$ is the parent of $h(n_2)$ in t; and,
- n and $h(n)$ have the same labels and data values in t' and t.

In the following, whenever we say that T' a prefix of T without specifying N, we assume that N is empty.

Tree types. In XML, the structure of valid documents is described by DTDs. We use here a simplified version of DTDs that we call *tree type*. A tree type specifies, for each element name a, the set of element names allowed for children of nodes labeled a, together with some multiplicity constraint. We also specify a root name, a restriction that can easily be removed.

Consider the alphabet Σ of labels. We use the auxiliary notion of *multiplicity atom* to describe the children that nodes labeled a may have. A multiplicity atom is an expression $a_1^{\omega_1} \cdots a_k^{\omega_k}$ where the a_i are distinct labels in Σ and each ω_i is a symbol in $\{\star, +, ?, 1\}$, whose significance is explained below.

Definition 2.2 *A* tree type τ *(over alphabet Σ) is a triple (Σ, R, μ) where $R \subseteq \Sigma$ is a set of* root labels*, and μ associates to each $a \in \Sigma$ a multiplicity atom $\mu(a)$ called the* type of a*.*

Satisfaction of a tree type $\tau = (\Sigma, R, \mu)$ by a data tree t is defined in the obvious way as follows. The label of the root of t belongs to R and for each node n in t labeled a, if $\mu(a) = a_1^{\omega_1} \cdots a_k^{\omega_k}$, then all children of a have labels among $\{a_1 \cdots a_k\}$ and if $\mu(a)$ contains $a_i^{\omega_i}$, the number of children of n labeled a_i is restricted as follows:

$\omega_i = 1$	:	exactly one child is labeled a_i;
$\omega_i = ?$	:	at most one child is labeled a_i;
$\omega_i = +$	:	at least one child is labeled a_i.
$\omega_i = \star$	:	no restriction;

The set of trees satisfying τ is denoted by $rep(\tau)$.

We sometimes denote a tree type (Σ, R, μ) by μ when Σ and R are understood. In examples, we specify tree types as below:

root: *catalog*

$$\begin{array}{lll} catalog & \rightarrow & product^{+} \\ product & \rightarrow & name\ price\ cat\ picture^{\star} \\ cat & \rightarrow & subcat \end{array}$$

Observe that we omit the 1 in exponents, e.g., we write *name* for $name^1$. The same tree type is represented graphically in Figure 1 in tree form, with multiplicities placed on edges.

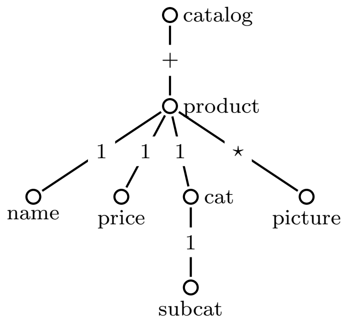


Figure 1: The tree type for the catalog example

Queries. We define a simple query language that selects prefixes of input trees. Although very limited, we claim that this language is often sufficient in practice. The query basically browses the input tree down to a certain depth starting from the root, by reading nodes with specified element names and possibly selection conditions on data values. All nodes involved in the pattern are extracted (so there is no projection), as well as subtrees of specified leaves. The pattern may also specify the non-existence of nodes with a given label. We call such a query a *prefix-selection query* (ps-query).

More formally, a ps-query is a labeled tree $\langle t, \lambda, \text{cond}\rangle$ where:

- t is a rooted tree;
- λ associates to each node of t a label in the extended alphabet $\Sigma \cup \{\overline{a} \mid a \in \Sigma\}$. Internal nodes can only have labels in Σ, and no two sibling nodes have labels among $\{a, \overline{a}\}$ for the same a.
- *cond* associates to each node of t a *condition*, which is a Boolean combination of expressions of the form $= v, \neq v, \leq v, \geq v, < v, > v$, where $v \in \mathbb{Q}$.

A node adorned with a bar indicates that the entire subtree rooted at that node is extracted.

We will use implicitly the following fact about the conditions defined above in several algorithms.

Lemma 2.3 *It can be checked in PTIME whether a given condition is satisfiable. Moreover each condition φ is equivalent to a union of intervals linear in the size of φ.*

Proof: Let φ be Boolean combination of expressions of the form $= v, \neq v, \leq v, \geq v, < v, > v$, where $v \in \mathbb{Q}$. Let $C = \{v_1, \ldots, v_n\}$ be the set of values mentioned in φ, where $v_i < v_{i+1}$, $1 \leq i < n$. Consider the set of open intervals $\mathcal{I} = \{(-\infty, v_1), (v_i, v_{i+1}), (v_n, +\infty) \mid 1 \leq i < n\}$. Clearly, for each interval $I \in \mathcal{I}$, φ has the same value for all numbers in I. Thus, to test satisfiability of φ it is enough to evaluate it on C together with one value from each interval in $\mathcal{I}$. Moreover φ is equivalent to the the union of the intervals in $\mathcal{I}$ on which φ is true with the set of closed intervals $\{[c, c] \mid c \in C, \varphi(c) = \mathit{true}\}$. This is obviously linear in φ. $\square$

Several example queries are shown next.

Example 2.1 We continue the previous catalog example. The following are all ps-queries.

- Query 1: find the name, price and subcategories of electronics products with price less than \$200.

Query 1 is represented in Figure 2 The catalog and product nodes are also shown, as they are part of the prefix leading to the desired nodes; the category is shown since it is part of the selection condition. In particular, note that there is no mechanism for projecting out nodes in ps-queries.

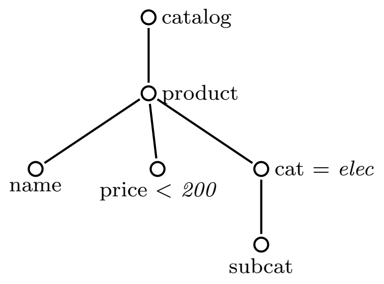


Figure 2: Query 1

- Query 2: find the name and picture of all cameras (inside the category electronics) whose picture appears in the catalog.

Query 2 is represented in Figure 3.

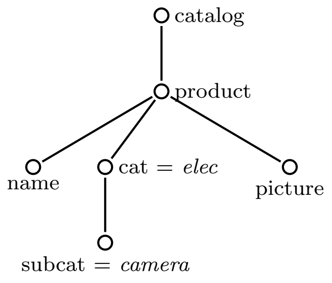


Figure 3: Query 2

- Query 3: find the name, price and pictures of all cameras (inside the category electronics) costing less than $100 and having at least one picture.

Query 3 is represented in Figure 4.

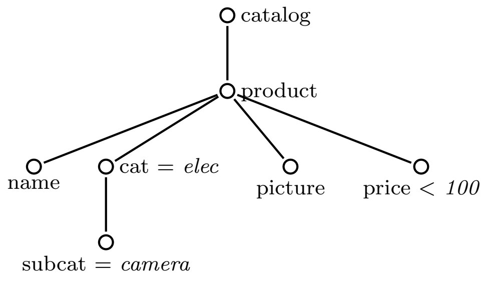


Figure 4: Query 3

- Query 4: list all cameras inside the category electronics.

Query 4 is represented in Figure 5.

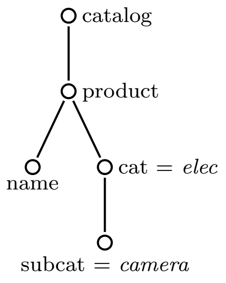


Figure 5: Query 4

We next formalize the notion of answer to a query using the auxiliary concept of valuation. Given a ps-query $q = \langle t, \lambda, \text{cond}\rangle$ and an input data tree T, a *valuation h from q to T* is a mapping from the nodes of t into nodes of T such that:
(0) $h(root(t)) = root(T)$;
(1) each edge $\langle n, m\rangle$ in t is mapped by h to an edge of T;
(2) for each node n in t such that $\lambda(n) \in \{a, \overline{a}\}$, $h(n)$ has label a in T;
(3) for each n, the value of $h(n)$ in T satisfies $cond(n)$.

The *answer* $q(T)$ is the prefix tree of T consisting of the nodes n which are in the image of some valuation h from q to T, or are descendants of such a node with label $\overline{a}$, $a \in \Sigma$. Possible answers to Queries 1 and 2 in the catalog example above are depicted in Figure 6.

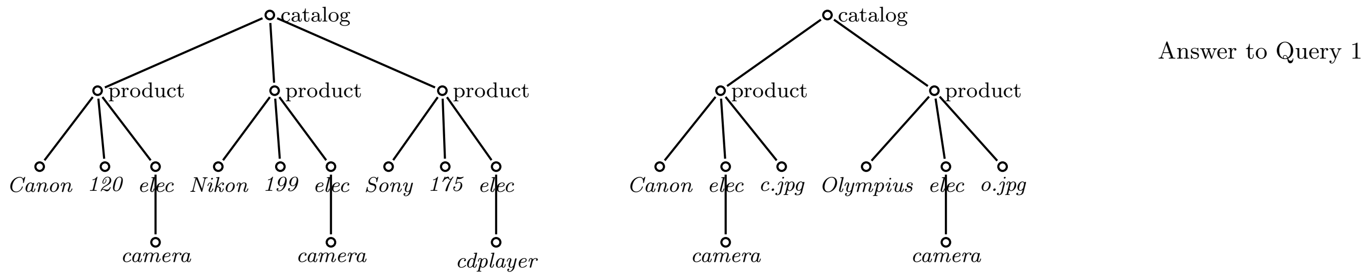


Figure 6: Possible answers of queries 1 and 2. In order to improve readability, we omitted labels of leaves (obvious from the context) and depicted in italics the data value associated with each node.

The following remark highlights an essential aspect of the model.

Remark 2.4 (Object identifiers) *Consider a tree T and two consecutive queries q_1, q_2. The answers $q_1(T)$ and $q_2(T)$ are both prefixes of T (relative to the nodes of T) and all their nodes are nodes of T. Thus, data from consecutive queries pertaining to the same node of T can be combined, enriching the information about that node. This aspect of the model is important; it amounts to having persistent node identifiers in the input and answers.*

Conditional tree types. We next discuss our representation of incomplete information. The main idea is that at any given time the Webhouse has obtained, as a result of previous queries, a prefix of the full data tree representing the complete data. In addition, from the queries and the initial type definitions of the sources, there is partial information about the missing portion of the full tree. Our representation of incomplete information includes the prefix tree obtained so far and the description of the missing information.

To describe the missing information we need to extend tree types in three ways: by allowing disjunctions of multiplicity atoms, by specifying *conditions* on data values, and by adding a *specialization* mechanism that allows defining several types for the same element name. For instance in the catalog example, after posing Query 1, we know that the missing data contains two types of products: *product1* and *product2*, the first for products whose category is not "electrical", and the second for products whose price is at least 200.

We next formally define our representation of incomplete information, extending the notion of tree type. Note that the data sources continue to be described by tree types, as previously defined.

Before introducing specialization, we define *simple conditional tree types.* A *condition* is defined as for queries. A *simple conditional tree type* over alphabet Σ is a tuple $(\Sigma, R, \mu, \mathrm{cond})$ where:

- $R \subseteq \Sigma$ is the set of root labels;
- μ is a mapping associating to each $a \in \Sigma$ a disjunction $\mu(a)$ of multiplicity atoms; and,
- cond associates a condition to each $a \in \Sigma$ (the condition applies to the data value of nodes with label a).

The *set of trees* represented by a simple conditional tree type $(\Sigma, R, \mu, \mathrm{cond})$ is defined in the obvious manner and denoted $rep(\Sigma, R, \mu, \mathrm{cond})$. We extend our notation for tree types to conditional tree types by allowing, on right hand sides of productions, disjunctions of multiplicity atoms, as in $a \ \rightarrow \ ab^{\star} \ \vee \ c^{?}d^{+}$. The *cond* mapping is specified independently of μ, as in $cond(a)$ = "$> 0 \wedge < 1$", $cond(b)$ = "=0", etc.

Next we consider specialization, already found useful in the context of DTDs for expressing structural properties that are dependent on the context of a node. Specialization has been considered previously in several investigations [9, 17, 36]. Specialization is achieved by allowing several possible *types* for the same element name. This suggests the following definition. A specialization mapping σ is a mapping from some specialized alphabet Σ' to some alphabet Σ. It transforms a data tree T with names in Σ' into a data tree $\sigma(T)$ with names in Σ in the obvious manner, by replacing each label a by $\sigma(a)$. We are now ready to define the most complex types used in the paper.

A *conditional tree type* over Σ and specialized alphabet Σ' is a tuple $(\Sigma', R, \mu, \mathrm{cond}, \sigma, \Sigma)$ where:

- $(\Sigma', R, \mu, \mathrm{cond})$ is a simple conditional tree; and,
- σ is a specialization mapping from Σ' to Σ.

Intuitively, the labels in Σ' specialize the labels in Σ via the specialization mapping σ.

The semantics of conditional tree types is defined as follows. A data tree T over Σ is in $rep(\Sigma', R, \mu, \mathrm{cond}, \sigma, \Sigma)$ if there exists a tree T' in $rep(\Sigma', R, \mu, \mathrm{cond})$ such that $\sigma(T') = T$.

Intuitively, there is a similarity between conditional tree types and unranked tree automata [6]. Both are used to define valid sets of trees, and the role of the specialized alphabet in conditional tree types is similar to that of states in a non-deterministic top-down tree automaton. The analogy does not hold fully because of the lack of order and the presence of data values in our trees. However, some of the flavor of the automata techniques carries through, and the sets of trees definable by conditional tree types have some properties similar to regular tree languages.

A key technical point for our algorithms is testing emptiness of the set of trees satisfying a conditional tree type. An easy reduction to and from testing emptiness of context-free grammars shows that:

Lemma 2.5 *(i) Let $(\Sigma', R, \mu, cond)$ be a simple conditional tree type. Checking emptiness of $rep(\Sigma', R, \mu, cond)$ is* PTIME-*complete. (ii) Let $(\Sigma', R, \mu, cond, \sigma, \Sigma)$ be a conditional tree type. Checking emptiness of $rep(\Sigma', R, \mu, cond, \sigma, \Sigma)$ is* PTIME-*complete.*

Note that (ii) follows trivially from (i), since $rep(\Sigma', R, \mu, \text{cond}, \sigma, \Sigma)$ is nonempty if and only if $rep(\Sigma', R, \mu, \text{cond})$ is nonempty.

A consequence of part (i) of Lemma 2.5 is the following.

Corollary 2.6 *Let $(\Sigma', R, \mu, cond)$ be a simple conditional tree type. One can test in* PTIME *whether a symbol $a \in \Sigma'$ is* useful*, i.e. whether there exists some T' in $rep(\Sigma', R, \mu, cond)$ with some node labeled a.*

Proof: Let $\Sigma'_a = \{(b, a) \mid b \in \Sigma', b \neq a\}$. Intuitively, (b, a) labels a node with original label $b \neq a$ additionally constrained so that all subtrees rooted at the node must contain some node labeled a. Consider the simple conditional tree type $(\Sigma' \cup \Sigma'_a, R', \mu', \text{cond}')$ where $R' = \{(r, a) \mid r \in R, r \neq a\} \cup \{a \mid a \in R\}$, $cond'(b) = cond'(b, a) = cond(b)$, and μ' is defined as follows:

- if $b \in \Sigma'$, then $\mu'(b) = \mu(b)$;
- if $(b, a) \in \Sigma'_a$, then $\mu'(b, a)$ is obtained by modifying the disjuncts of $\mu(b)$ as follows. Let $a_1^{\omega_1} \dots a_k^{\omega_k}$ be a disjunct of $\mu(b)$. If there exists $i \in [1, k]$ such that $a_i = a$ and $\omega_i \in \{1, +\}$, then $a_1^{\omega_1} \dots a_k^{\omega_k}$ is in $\mu'(b, a)$. Otherwise, $a_1^{\omega_1} \dots a_k^{\omega_k}$ is replaced by k disjuncts obtained by substituting, for each $i \in [1, k]$, the multiplicity atom $a_i^{\omega_i}$ by $(a_i, a)^1$ if $a_i \neq a$, and by a^1 if $a_i = a$.

Clearly, $rep(\Sigma', R, \mu, \text{cond})$ contains some tree with a node labeled a if and only if $rep(\Sigma' \cup \Sigma'_a, R', \mu', \text{cond}') \neq \emptyset$. By Lemma 2.5, the latter can be tested in PTIME. □

Incomplete trees. As discussed earlier, the representation of incomplete information consists of two aspects: a known portion of a full data tree, and information on the missing portion of the tree. To represent the mix of known and missing information, we enrich conditional tree types with the ability to specify a set N of instantiated nodes, together with their labels and data values. The definition is complicated by the fact that incomplete trees must be able to represent trees containing different subsets of N as instantiated nodes (the need for this is illustrated in Example 2.2). Also, the same instantiated node must be allowed to have different types in different contexts, just like labels that are allowed to have multiple specializations. To deal with this uniformly, instantiated nodes are also viewed as labels. This gives rise to the notion of incomplete tree.

Definition 2.7 *An* incomplete tree *over Σ is a 4-tuple (N, λ, ν, τ) where:*

1. *$N \subset \mathcal{N}$ is a finite set of nodes;*
2. *$\lambda : N \to \Sigma$ is a labeling of the nodes in N;*
3. *$\nu : N \to \mathbb{Q}$ associates to each node in N a data value in $\mathbb{Q}$*
4. *τ is a conditional tree type over alphabet $N \cup \Sigma$ such that for each data tree $T \in rep(\tau)$:*
   - *for each $n \in N$ there is at most one node of T labeled n;*
   - *if a node in T has label in N, then its parent's label is also in N.*

Note that, given a conditional tree τ over alphabet $N \cup \Sigma$, Requirement (4) above can be verified in time polynomial with respect to τ. We denote incomplete trees by $\mathbf{T}$, $\mathbf{T}_1$, $\mathbf{T}_2$, etc.

The set of trees represented by an incomplete tree $\mathbf{T}$ as in the definition consists of the data trees T over Σ such that there exists a data tree $T_0 = \langle t_0, \lambda_0, \nu_0 \rangle$ over $N \cup \Sigma$ such that:

- T_0 satisfies τ;
- for each node n of T_0, $n \in N$ if and only if $\lambda_0(n) \in N$, in which case $n = \lambda_0(n)$;
- if n is a node of T_0 and $n \in N$, then $\nu_0(n) = \nu(n)$; and,
- T is obtained from T_0 by changing each label $n \in N$ to $\lambda(n) \in \Sigma$.

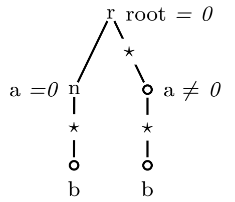
The set of trees represented by $\mathbf{T}$ is denoted *rep*($\mathbf{T}$). Similarly to conditional trees, it is decidable in PTIME whether *rep*($\mathbf{T}$) is empty.

Given an incomplete tree $\mathbf{T}$ as above, we refer to N as the set of *data nodes* of $\mathbf{T}$. Note that a tree $T \in rep(\mathbf{T})$ need not contain *all* nodes in N. However, for each such node it contains, its label and data value are those specified by λ and ν. Also note that, by condition (4), the restriction of T to N forms a prefix of T. We call this subtree the *data tree* of T, denoted T_d.

Example 2.2 We illustrate and motivate the definition of incomplete tree with two examples. The first is a very simple incomplete tree with two data nodes. Let $\Sigma = \{a, b, root\}$ and $\mathbf{T} = (N, \lambda, \nu, \tau)$ be the incomplete tree over Σ where:

- $N = \{r, n\}$;
- $\lambda(r) = root$, $\lambda(n) = a$, $\nu(r) = 0$, $\nu(n) = 0$;
- $\tau = (\Sigma \cup N, R, \mu, \text{cond}, \sigma, \Sigma \cup N)$ where σ is the identity, $R = \{r\}$, $\mu(r) = na^*, \mu(a) = b^*, \mu(n) = b^*, \mu(b) = \epsilon$, $cond(r) = cond(n) =$ "$= 0$", $cond(a) =$ "$\neq 0$", $cond(b) = true$.

Thus, *rep*($\mathbf{T}$) consists of trees with root r labeled *root*, and a child n labeled a. In addition, r may have zero or more children labeled a; and n, as well as all other nodes labeled a, may have zero or more children labeled b. The incomplete tree $\mathbf{T}$ is represented informally in Figure 7 (left).



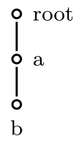


Figure 7: The incomplete tree $\mathbf{T}$ (to the left) and the query q (to the right)

Now consider the ps-query q in Figure 7 (right).

Suppose q is posed to data trees in *rep*($\mathbf{T}$). This results in a set of possible answers. One of the desired uses of incomplete trees is to describe such possible answers. We show how this can be done in our example using a second incomplete tree. Note that although all inputs contain the data nodes r and n, not all answers contain these nodes. Some answers contain both r and n, others contain r but not n (if n has no children), and yet others contain neither r nor n (the empty tree is a possible answer). Let $\mathbf{T}' = (N, \lambda, \nu, \tau')$, where N, λ, ν are as above and $\tau' = (\Sigma', R', \mu', \text{cond}', \sigma', \Sigma \cup N)$ is defined by:

- $\Sigma' = \{r_1, r_2, n, a, b\}$;
- $R' = \{r_1, r_2\}$;
- $\sigma'(r_1) = \sigma'(r_2) = r$, $\sigma'(n) = n, \sigma'(a) = a, \sigma'(b) = b$;
- $cond'(r_1) = false$, $cond'(r_2) = cond'(n) =$ " $= 0$", $cond'(a) =$ " $\neq 0$", and $cond'(b) = true$;
- $\mu'(r_1) = \epsilon, \mu'(r_2) = na^* + a^+, \mu'(a) = \mu'(n) = b^+$, $\mu'(b) = \epsilon$.

It can be easily checked that $rep(\mathbf{T}')$ consists of all answers obtained by applying q to the trees in $rep(\mathbf{T})$.

Given an incomplete tree, it is often of interest to check whether some facts or sets of facts are *certain*, or whether they are *possible* given the partial information available. In our framework, the most natural facts of interest are usually prefixes of trees. Given an incomplete tree $\mathbf{T}$ with data nodes N and another data tree T over Σ, we say that T is a *certain* prefix of $\mathbf{T}$ if $rep(\mathbf{T}) \neq \emptyset$ and *every* tree in $rep(\mathbf{T})$ has T as a prefix relative to N, and it is a *possible* prefix if *some* tree in $rep(\mathbf{T})$ has T as a prefix relative to N. We can show the following.

Theorem 2.8 *Given a data tree T and an incomplete tree $\mathbf{T}$ over Σ, it can be checked in* PTIME *whether T is a certain prefix or whether T is a possible prefix of $\mathbf{T}$.*

Proof: We use the notation in Definition 2.7. Let $\mathbf{T} = (N, \lambda, \nu, \tau)$ and $\tau = (\Sigma', R, \mu, \text{cond}, \sigma, \Sigma \cup N)$. If $rep(\mathbf{T}) = \emptyset$, then T is not a certain prefix nor a possible prefix of $\mathbf{T}$ (as noted earlier, this can be checked in PTIME). Suppose $rep(\mathbf{T}) \neq \emptyset$. Without loss of generality, we can assume that the specialized alphabet Σ' of τ has no useless symbols (otherwise we can first eliminate them in PTIME). Note that, in order for T to be a certain or possible prefix of $rep(\mathbf{T})$, its restriction to N has to be a prefix of itself and the labeling and data values of nodes in N have to be compatible with λ and ν. If this is the case, we modify T by replacing the label of each node $n \in N$ by n. Then we proceed as described below.

To test whether T is a certain prefix, we construct for each node n of T the set $Cert(n)$ consisting of the labels $a \in \Sigma'$ for which the subtree of T rooted at n is a certain prefix of the incomplete tree $\mathbf{T}^a$, which is identical to $\mathbf{T}$ except that τ is modified so that $R = \{a\}$. Clearly, T is a certain prefix of $\mathbf{T}$ if and only if $R \subseteq Cert(root(T))$ (where R is the original set of root types for τ).

We use the notation $cond(a) = v$ to mean that the union of intervals equivalent to $cond(a)$ equals $[v, v]$ (in other words, $cond(a)$ is equivalent to the condition " $= v$"). By Lemma 2.3, this can be checked in linear time. We also write $v \models cond(a)$ to denote that v satisfies $cond(a)$.

$Cert(n)$ is computed recursively, starting with the leaves of T. If n is a leaf, $Cert(n) = \{a \mid \sigma(a) = \lambda(n), cond(a) = \nu(n)\}$. Suppose n is an internal node. Consider some $a \in \sigma^{-1}(\lambda(n))$ such that $cond(a) = \nu(n)$. Note that, since there are no useless symbols, $rep(\mathbf{T}^a) \neq \emptyset$. In general, n has some children in N (in which case n is also in N) and some that are not in N. Consider $\mu(a)$. If there is some disjunct in $\mu(a)$ which is not compatible with the children of n, then $a \notin Cert(n)$. Otherwise, consider a disjunct $a_1^{\omega_1} \cdots a_k^{\omega_k}$ in $\mu(a)$. Thus, for every child m of n in N there exists a unique $i \in [1,k]$ with $a_i \in \sigma^{-1}(\lambda(m))$. In order for a to be in $Cert(n)$, a_i must be in $Cert(m)$. Additionally, there must exist an injective mapping f from the children of n not in N to $[1,k]$ such that $a_{f(m)}$ is not in $\sigma^{-1}(N)$, $\omega_{f(m)} \in \{+,1\}$ (so the presence of a node of type $a_{f(m)}$ is guaranteed), and $a_{f(m)} \in Cert(m)$ for all m. Checking the existence of f can be done in PTIME by checking the existence of a perfect matching of the children of n not in N with the appropriate indices in $[1,k]$. If the properties above are satisfied for all disjuncts in $\mu(a)$, it follows that $a \in Cert(n)$. It is easily verified that the above recursive procedure computes the desired mapping $Cert$.

The algorithm for testing whether T is a possible prefix is similar. For each node n of T we construct the set $Poss(n)$ consisting of the labels $a \in \Sigma'$ for which the subtree of T rooted at n is a possible prefix of the incomplete tree $\mathbf{T}^a$ which is identical to $\mathbf{T}$ except that τ is modified so that $R = \{a\}$. Clearly, T is a possible prefix of $\mathbf{T}$ if and only if $R \cap Poss(root(T)) \neq \emptyset$ (where R is the original set of root types for τ). $Poss(n)$ is computed recursively, starting with the leaves of T. If n is a leaf, $Poss(n) = \{a \mid \sigma(a) = \lambda(n), \nu(n) \models cond(a)\}$. Suppose n is an internal node. Consider some $a \in \sigma^{-1}(\lambda(n))$ such that $\nu(n) \models cond(a)$. Note that, since there are no useless symbols, $rep(\mathbf{T}^a) \neq \emptyset$. In general, n has some children in N (in which case n is also in N) and some that are not in N. Consider $\mu(a)$. In order for a to be in $Poss(n)$, there must exist a disjunct $a_1^{\omega_1} \cdots a_k^{\omega_k}$ in $\mu(a)$, with the following properties. First, for every child m of n in N there exists a unique $i \in [1,k]$ with $a_i \in \sigma^{-1}(\lambda(m))$, such that $a_i \in Poss(m)$. Consider now the set C of children of n not in N. For each $m \in C$ there must exist $i \in [1,k]$ such that $a_i \in Poss(m)$. If for some such i, $w_i \in \{+,*\}$, then remove m from C. For the remaining C, there must exist an injective mapping f associating to each m in C an $i \in [1,k]$ such that $a_i \in Poss(m)$ and $w_i \in \{1,?\}$. As earlier, the existence of such f can be checked in PTIME by verifying the existence of a perfect matching between C and the appropriate indices in $[1,k]$. If the above properties are satisfied for some disjunct in $\mu(a)$, it follows that $a \in Poss(n)$. It is easily seen that this recursive procedure computes $Poss$. $\square$

3 Acquiring and Using Incomplete Information

We present here the main results of the paper, showing that our framework can be efficiently used to deal with incomplete information in the scenario we described. We first deal with acquiring partial information, and discuss the possible exponential blowup of the representation. We then consider the problem of answering queries when our knowledge consists of an incomplete tree, i.e., the instantiation to our model of the classical problem of querying incomplete databases. Finally, we consider the issue of "completing" our knowledge in order to fully answer a given query.

3.1 Acquiring incomplete information

In our basic scenario, information about the Web is acquired gradually using answers to queries. We next show how this can be done in the framework we developed. For simplicity, we assume that the input is a single document described by a tree type. The case of multiple sources can be easily reduced to this case by virtually merging the sources into a single document.

Consider an input tree T. As consecutive ps-queries are asked, each answer provides partial information about T, and refines the information obtained from previous queries. In addition, we know that T satisfies a given tree type, say τ. We describe the information available after a sequence of queries using an incomplete tree. At each stage of the process, we have an incomplete tree $\mathbf{T}$, a ps-query q and the answer $A = q(T)$. Using q and A, we refine our incomplete information provided by $\mathbf{T}$ by computing $\mathbf{T}'$ which describes *precisely* the trees in $rep(\mathbf{T})$ and compatible with the answer A to query q. The refinement algorithm is called *Refine*$(\mathbf{T}, q, A)$ and is defined formally after the following example.

Example 3.1 As a warm-up, we first illustrate the algorithm using the catalog example of Figures 1 and 6 and Example 2.1.

Assume that the first query of the sequence is Query 1 whose answer is the data tree of Figure 6. The incomplete tree $\mathbf{T}_1$ after Query 1 contains the data tree, and an incomplete tree that describes the missing products. A product is not returned by Query 1 if (i) it is not an electronics product or (ii) it is an electronics product but its price is not less than 200. The incomplete tree is obtained by creating two new labels *product1* and *product2* with obvious conditions attached to them and which are specializations of *product*. The incomplete tree T_1 is depicted in Figure 8.

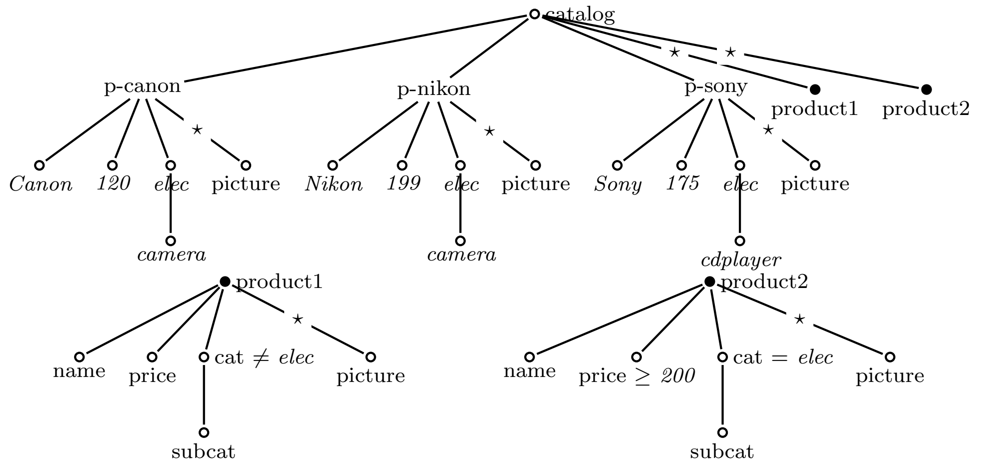


Figure 8: The incomplete tree after Query 1. We omitted some labels when a data value is present and the label is clear from the context.

Assume now that the next query is Query 2 whose answer is the data tree of Figure 6. The construction of the new incomplete tree $\mathbf{T}_2$ requires us to represent several kinds of products:

Products returned by both Query 1 and 2: these are the cheap cameras with pictures. Suppose the node ids indicate that the products with name *Canon* in the answers to Queries 1 and 2 are the same. The information returned for this node by the two queries can be merged. Note that persistent node id assumption is critical here.

Products returned by Query 2 and not Query 1: The typing information is used to register the fact that the (unknown) price of these products must be at least 200. This is the case for the Olympus camera, which is of type *p2-olympus*.

Products returned by Query 1 and not Query 2: This is the case of the Nikon camera. A product returned by Query 1 is in this category either because it is not a camera or because it is a camera and has no picture. In the case of Nikon, we already know it is a camera, so we can infer that it has no picture, i.e., its type is *p-nikon*.

Missing products: A product may be returned neither by Query 1 nor by Query 2 because it is not an electronic product, because it is expensive but not a camera, or because it is an expensive camera without pictures. This yields the three categories of missing nodes (colored black).

Note that *product2b* and *product2c* are refinements of *product2*.

The resulting incomplete tree is depicted in Figure 9.

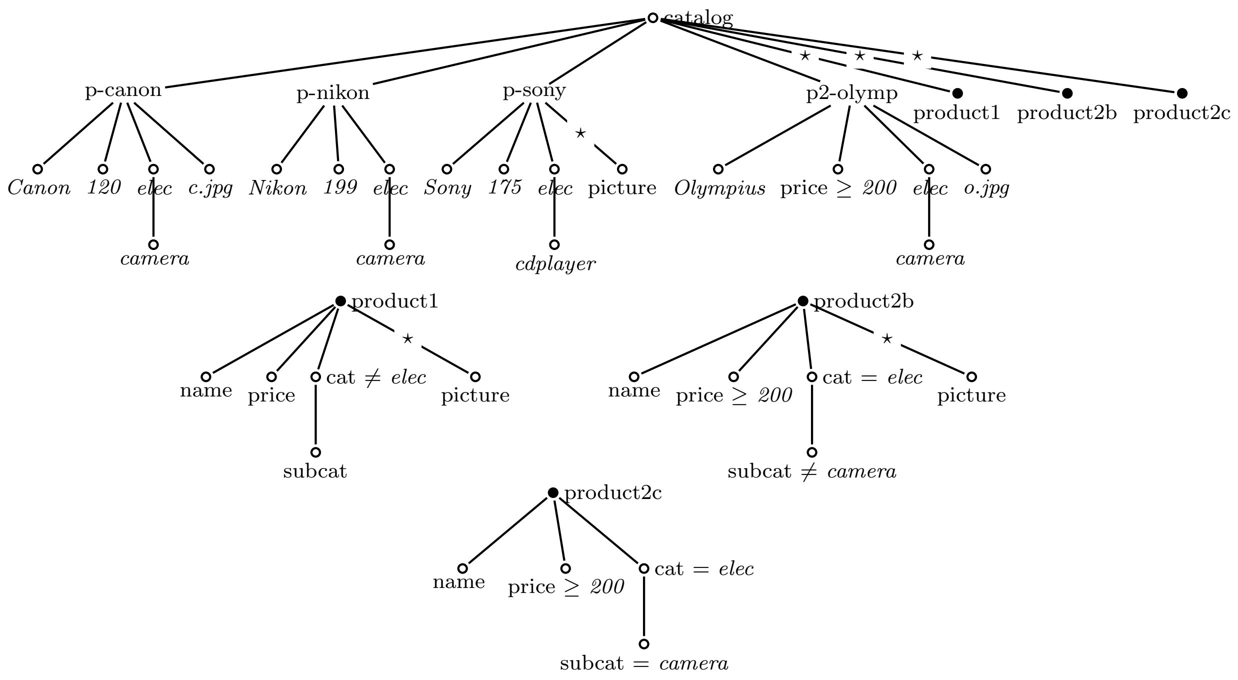


Figure 9: The incomplete tree after query 2

□

Algorithm Refine. It will turn out that in order to describe the information obtained by a sequence of ps-queries, it is sufficient to use incomplete trees with a particularly simple structure, called "unambiguous", and defined next.

Definition 3.1 *An incomplete tree* $\mathbf{T} = (N, \lambda, \nu, \tau)$ *where* $\tau = (\Sigma', \mu, cond, \sigma, \Sigma \cup N)$ *is* unambiguous *if for every* $a \in \Sigma'$ *and multiplicity atom* α *in* $\mu(a)$*:*

1. *if* a^ω *occurs in* α *and* $\sigma(a_i) \in N$*, then* $\omega = 1$*; otherwise,* $\omega = *$*;*
2. *if* a_i^* *and* a_j^**,* $i \neq j$*, occur in* α *and* $\sigma(a_i) = \sigma(a_j) \in \Sigma$*, then* $cond(a_i) \wedge cond(a_j)$ *is unsatisfiable.*
3. *if* a_i^* *and* a_j^**,* $i \neq j$*, occur in* α *and* $\sigma(a_i) = \sigma(a_j) \in \Sigma$*, then there exists* a_k^1 *occurring in* α *such that* $\sigma(a_k) = n \in N$ *and* $\lambda(n) = \sigma(a_i) = \sigma(a_j)$*.*

Thus, (1) says that, in an unambigous incomplete tree, types of data nodes have multiplicity 1 and types representing missing information have unrestricted multiplicity; (2) says that different specializations of the same label are either data nodes or have mutually exclusive conditions; finally, (3) says that every label that has multiple specializations is also the label of some data node. In particular, if $\mathbf{T}$ is unambiguous, (2) ensures that a node in a data tree can only be associated with one type in Σ' in any successful typing of the tree.

The input to Algorithm Refine is an unambiguous incomplete tree $\mathbf{T}$ and a ps-query q with answer A. The output is a new unambiguous incomplete tree $\mathbf{T}'$ such that

$$rep(\mathbf{T}') = rep(\mathbf{T}) \cap q^{-1}(A).$$

The computation of $\mathbf{T}'$ is done in two steps. The first step shows that there exists an unambiguous incomplete tree representing $q^{-1}(A)$, that is the set of all data trees T such that $q(T) = A$. The second step shows how to compute the intersection of two unambiguous incomplete trees.

Lemma 3.2 *Given a ps-query q and an answer A to q, there exists an unambiguous incomplete tree $\mathbf{T}_{q,A}$ for which $rep(\mathbf{T}_{q,A}) = \{T \mid q(T) = A\}$. Furthermore, $\mathbf{T}_{q,A}$ can be computed in time polynomial in q and A.*

Proof: Let $q = \langle t_q, \lambda_q, \text{cond}_q\rangle$, and $A = (t_A, \lambda_A, \nu_A)$. We construct an incomplete tree $\mathbf{T}_{q,A} = (N, \lambda, \nu, \tau)$ where $\tau = (\Sigma', \mu, \text{cond}, \sigma, \Sigma \cup N)$. The idea of the construction is the following. First, if $q(T) = A$, then A must be a prefix of T. Second, all other nodes of T should violate some condition of q. The type τ must describe all reasons for such a violation.

To simplify the presentation, assume first that q contains no nodes labeled with $\bar{a}$.

First, set $(N, \lambda, \nu) = A$. We define Σ' as the set consisting of all τ_a for $a \in \Sigma$, all τ_n for $n \in N$ and all $\bar{\tau}_m, \hat{\tau}_m$ for $m \in t_q$. The meaning of each type is the following: τ_a is the type of all nodes labelled a without any constraint on the node and its subtree; τ_n is the type of the output node n in A; $\bar{\tau}_m$ describes the nodes with label $\lambda_q(m)$ that make q false at m by violating $cond_q(m)$; finally, $\hat{\tau}_m$ describes the nodes with label $\lambda_q(m)$ that satisfy $cond_q(m)$ but for which the subtree of q rooted at m cannot be matched below the node.

We obtain this behavior as follows. We set for all $a \in \Sigma$, $\sigma(\tau_a) = a, \text{cond}(\tau_a) = \text{true}$. Assume $\Sigma = \{a_1 \cdots a_n\}$. Let all* be the multiplicity atom $\tau_{a_1}^\star \cdots \tau_{a_n}^\star$. Set $\mu(\tau_a) = \text{all}^\star$.

Consider now $m \in t_q$. We set $\sigma(\bar{\tau}_m) = \lambda_q(m)$, $\text{cond}(\bar{\tau}_m) = \neg\text{cond}_q(m)$, and $\mu(\bar{\tau}_m) = \text{all}^\star$.

If m is not a leaf, then let $m_1 \cdots m_l$ be the children of m. We set $\sigma(\hat{\tau}_m) = \lambda_q(m)$, $\text{cond}(\hat{\tau}_m) = \text{cond}_q(m)$ and $\mu(\hat{\tau}_m) = \bigvee_{1\le i\le l} \alpha_i$ where α_i is the multiplicity atom $\bar{\tau}_{m_i}^\star \hat{\tau}_{m_i}^\star \text{else}_i$, where else_i contains $\tau_a^\star$ for each $a \in \Sigma$ such that $a \neq \lambda(m_i)$. Therefore a node n is of type $\hat{\tau}_m$ if there at least one m_i for which the subquery of q rooted at m_i cannot be matched against any subtree rooted at a child of n. The children of n with labels other than $\lambda_q(m_i)$ are immaterial and therefore not constrained.

Finally, consider $n \in N$. We set $\sigma(\tau_n)$ to $\lambda_A(n)$ and $\text{cond}(\tau_n)$ to "$= \nu_A(n)$". If n is a leaf, we set $\mu(\tau_n) = \text{all}^\star$. Otherwise, let m be the node of t_q such that there is a valuation from q to A mapping m to n. (Recall that A is the output of q for some data tree.) Let $n_1 \cdots n_k$ be the children of n and $m_1 \cdots m_l$ be the children of m. Set $\mu(\tau_n) = \tau_{n_1} \cdots \tau_{n_k} \bar{\tau}_{m_1}^\star \hat{\tau}_{m_1}^\star \cdots \bar{\tau}_{m_l}^\star \hat{\tau}_{m_l}^\star \text{else}_n$, where else_n contains $\tau_a^\star$ for each $a \in \Sigma$ that is not a label of any of the children of n in A (and therefore of m in t_q). In words, this says that n has exactly the children already present in the output A, some children with the same labels but which violate q, and possibly nodes with other labels that are irrelevant to the query.

To conclude, set R to $\{\tau_r\}$ if A is non-empty and r is the root of τ_A. If A is empty we set R to $\{\bar{\tau}_r, \hat{\tau}_r, (\tau_a)_{a\neq\lambda_q(r)}\}$, where r is the root of t_q and $a \in \Sigma$.

The above construction can be easily modified to take into account the occurrence of labels and $\bar{a}$ in the query q. If $n \in N$ is such that the corresponding $m \in t_q$ is labelled with $\bar{a}$, then we no longer set $\mu(\tau_n) = \text{all}^\star$ as we are sure that all nodes below n have already been extracted. We omit the details.

It is easy to verify that $rep(\mathbf{T}_{q,A}) = q^{-1}(A)$, $\mathbf{T}_{q,A}$ is unambiguous, and $\mathbf{T}_{q,A}$ can be computed in time $O((|q| + |A|) \cdot |\Sigma|)$. □

Two incomplete trees $(N_1, \lambda_1, \nu_1, \tau_1)$ and $(N_2, \lambda_2, \nu_2, \tau_2)$ are said to be *compatible* if for each $n \in N_1 \cap N_2$, we have $\lambda_1(n) = \lambda_2(n)$ and $\nu_1(n) = \nu_2(n)$.

We next show the following.

Lemma 3.3 *Let $\mathbf{T}_1$ and $\mathbf{T}_2$ be unambiguous incomplete trees such that $\mathbf{T}_1$ and $\mathbf{T}_2$ are compatible. Then there exists an unambiguous incomplete tree $\mathbf{T}$ such that $rep(\mathbf{T})= rep(\mathbf{T}_1) \cap rep(\mathbf{T}_2)$. Moreover, $\mathbf{T}$ can be constructed in time polynomial with respect to $\mathbf{T}_1$ and $\mathbf{T}_2$.*

Proof: $\mathbf{T}$ is constructed as a carefully chosen product of $\mathbf{T}_1$ and $\mathbf{T}_2$. Intuitively, the construction resembles the intersection of two tree automata. The difficulty is to merge two disjunctions of multiplicity atoms into a new disjunction describing their intersection.

Let $\mathbf{T}_i = (N_i, \lambda_i, \nu_i, \tau_i)$ where $\tau_i = (\Sigma_i, R_i, \mu_i, \text{cond}_i, \sigma_i, N_i \cup \Sigma)$, $i = 1, 2$. We construct $\mathbf{T}= (N, \lambda, \nu, \tau)$ where $\tau = (\Sigma', R, \mu, \text{cond}, \sigma, N \cup \Sigma)$ as follows.

Let $N = N_1 \cup N_2$. For all $n \in N_1$ set $\lambda(n) = \lambda_1(n)$ and $\nu(n) = \nu_1(n)$. For all $n \in N_2$ set $\lambda(n) = \lambda_2(n)$ and $\nu(n) = \nu_2(n)$. Compatibility ensures that this construction is well defined.

We now construct $\tau = (\Sigma', R, \mu, \text{cond}, \sigma, N \cup \Sigma)$. Two types $t_1 \in \Sigma_1$ and $t_2 \in \Sigma_2$ are *compatible* if one of the following holds:

(i) $\sigma_1(t_1) = \sigma_2(t_2) \in \Sigma \cup (N_1 \cap N_2)$;

(ii) $\sigma_1(t_1) \in N_1 - N_2$ and $\sigma_2(t_2) = \lambda_1(\sigma_1(t_1))$; or,

(iii) $\sigma_2(t_1) \in N_2 - N_1$ and $\sigma_1(t_1) = \lambda_2(\sigma_2(t_2))$.

Let Σ' consist of of all pairs of compatible types from Σ_1 and Σ_2. For $(t_1, t_2) \in \Sigma'$, we set $\sigma((t_1, t_2))$ to $\sigma_1(t_1)$ if (i) or (ii) hold, and to $\sigma_2(t_1)$ if (iii) holds. Note that if at least one of the compatible types is a specialization of a data node, then (t_1, t_2) is a specialization of the same data node. In all cases, we set $cond((t_1, t_2)) = \text{cond}_1(t_1) \wedge \text{cond}_2(t_2)$, as expected. Also, R is defined as $\{(t_1, t_2) \in \Sigma' \mid t_1 \in R_1 \wedge t_2 \in R_2\}$.

The definition of μ requires more care. For each $(t_1, t_2) \in \Sigma'$, we consider a disjunct α_1 in $\mu(t_1)$ and a disjunct α_2 in $\mu(t_2)$, and combine them to create a set of disjuncts in $\mu(t_1, t_2)$ (as we will see, the set constructed is empty or a singleton). Consider two disjuncts α_1 in $\mu_1(t_1)$ and α_2 in $\mu_2(t_2)$. We denote the set of disjuncts resulting from combining α_1 and α_2 by $\alpha_1 \bowtie \alpha_2$. A *matching* ρ between α_1 and α_2 is the maximum subset of $\Sigma' \cap \{(a_1, a_2) \mid a_i^{\omega_i}$ occurs in $\alpha_i,\ i = 1, 2\}$ satisfying the following constraints:

1. for each a_1^1 occuring in α_1 there exists a_2 such that $(a_1, a_2) \in \rho$;
2. for each a_2^1 occuring in α_2 there exists a_1 such that $(a_1, a_2) \in \rho$; and
3. if $(a_1, a_2) \in \rho$ and $\sigma_1(a_1) \in N_1 - N_2$, then $\nu_1(a_1) \models cond_2(a_2)$, and conversely.

Note that the uniqueness of ρ follows from the unambiguity of $\mathbf{T}_1$ and $\mathbf{T}_2$. If ρ is empty, then $\alpha_1 \bowtie \alpha_2 = \emptyset$. Otherwise, $\alpha_1 \bowtie \alpha_2$ consists of the multiplicity atom containing $(a_1, a_2)^{\omega_1 \wedge \omega_2}$ for each $(a_1, a_2) \in \rho$, where $\wedge$ is the operation on $\{1, *\}$ defined by $1 \wedge \omega = \omega \wedge 1 = 1$ and $* \wedge * = *$. Finally, $\mu(t_1, t_2) = \cup\{\alpha_1 \bowtie \alpha_2 \mid \alpha_1 \in \mu_1(t_1), \alpha_2 \in \mu_2(t_2)\}$. It is easily seen that $rep(\mathbf{T})= rep(\mathbf{T}_1) \cap rep(\mathbf{T}_2)$. Moreover, $\mathbf{T}$ is unambiguous and can be constructed in time polynomial with respect to $\mathbf{T}_1$ and $\mathbf{T}_2$. □

From Lemma 3.2 and Lemma 3.3 we have:

Theorem 3.4 *Given an unambiguous incomplete tree* $\mathbf{T}$ *and a ps-query* q *with answer* A, *Algorithm* Refine *computes in polynomial time an unambiguous incomplete tree* $\mathbf{T}'$ *such that* $rep(\mathbf{T}') = rep(\mathbf{T}) \cap q^{-1}(A)$.

So far, the partial information computed by consecutive applications of Algorithm Refine does not take into account the known tree type of the input tree. This information can be combined whenever desired with the current result of Algorithm Refine, yielding another incomplete tree also computable in polynomial time.

Theorem 3.5 *Given an unambiguous incomplete tree* $\mathbf{T}$ *and a tree type* ρ, *there exists an incomplete tree* $\mathbf{T}'$, *computable in polynomial time from* $\mathbf{T}$ *and* ρ, *such that* $rep(\mathbf{T}') = rep(\mathbf{T}) \cap rep(\rho)$.

Proof: Let $\mathbf{T}= (N, \lambda, \nu, \tau)$ where

$$\tau = (\Sigma', R, \mu, \text{cond}, \sigma, N \cup \Sigma)$$

and $\rho = (\Sigma, R_\rho, \mu_\rho)$. We construct an incomplete tree $\mathbf{T}' = (N, \lambda, \nu, \tau')$ where $\tau' = (\Sigma', R', \mu', \text{cond}, \sigma, N \cup \Sigma)$. We set R' to $\{a' \in R \mid \exists\, a \in R_\rho(\sigma(a') = a \vee \lambda(\sigma(a')) = a)\}$. Thus, R' consists of those elements in R that are specializations of labels in R_ρ, or specializations of data nodes in N whose label (defined by λ) is in R_ρ. Next, μ' is defined by modifying μ as follows. Let $a \in \Sigma$ and $a' \in \Sigma'$ such that $\sigma(a') = a$ or $\sigma(a') = n \in N$ and $\lambda(n) = a$. We modify $\mu(a')$ by eliminating some disjuncts and changing others. Intuitively, a disjunct is eliminated if it is incompatible with $\mu_\rho(a)$. Specifically, a disjunct α in $\mu(a')$ is eliminated if at least one of the following holds:

- there exists b^ω occurring in $\mu_\rho(a)$ such that $\omega \in \{1, +\}$ and there is no $b_1^{\omega_1}$ occurring in α such that $\sigma(b_1) = b$ or $\sigma(b_1) = n \in N$ and $\lambda(n) = b$;
- there exists b^ω occurring in $\mu_\rho(a)$ such that $\omega \in \{1, ?\}$ and there is more than one $b_1^{\omega_1}$ as above, with $\omega_1 = 1$;
- there exists b_1^1 occurring in α such that there is no b^ω occurring in $\mu_\rho(a)$ such that $\sigma(b_1) = n \in N$ and $\lambda(n) = b$.

A disjunct α that is not eliminated is modified in order to conform to $\mu_\rho(a)$. This is done as follows. If b^ω occurs in $\mu_\rho(a)$ where $\omega \in \{1, ?\}$ and there exists $b_1^{\omega_1}$ with $\omega_1 = 1$ occurring in α such that $\sigma(b_1) = n \in N$ and $\lambda(n) = b$, then all b_2^* occurring in α for which $\sigma(b_2) = b$ are eliminated from α. If $b^\omega \in \mu_\rho(a)$, $\omega \in \{1, ?\}$, but the above condition does not hold, then replace the unique b_2^* occurring in α for which $\sigma(b_2) = b$ by b_2^ω; note that uniqueness of b_2 follows from (3) in the definition of unambiguity, since labels that do not appear on data nodes cannot have multiple specializations in the missing information part. If b^+ occurs in $\mu_\rho(a)$ and there is no $b_1^{\omega_1}$ with $\omega_1 = 1$ occurring in α such that $\sigma(b_1) = n \in N$ and $\lambda(n) = b$, then replace the unique b_2^* occurring in α for which $\sigma(b_2) = b$ by b_2^+. Again, uniqueness of b_2 follows from (3) in the definition of unambiguity. Finally, if b_1^* occurrs in α and there is no b^ω occurring in $\mu_\rho(a)$ such that $\sigma(b_1) = b$ or $\sigma(b_1) = n \in N$ and $\lambda(n) = b$, then remove b_1^* from α. It is easily seen that $\mathbf{T}'$ satisfies the statement of the theorem. □

Complexity. Algorithm *Refine* can be used to incrementally refine the information acquired by successive query-answer pairs $(q_1, A_1) \ldots, (q_n, A_n)$. Although each incremental step can be done in PTIME, the size of the incomplete tree may become exponential in the overall sequence of query-answer pairs, as illustrated in Example 3.2 below.

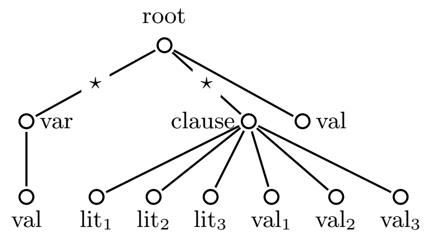
All of our algorithms on incomplete trees have PTIME complexity. However, since the incomplete trees themselves can become exponential with respect to the sequence of query-answer pairs from which they are constructed, the algorithms we developed have, in the worst case, exponential complexity with respect to the overall sequence. One might legitimately wonder if this fact is due to the particular representation system we have chosen. The answer turns out to be negative: We can prove lower-bounds independent of the representation system. We illustrate this type of result with the possible and certain prefix question, shown in Theorem 2.8 to be in PTIME with respect to the incomplete tree.

Theorem 3.6 *Let τ be a tree type over a fixed alphabet Σ, and $\langle q_i, A_i\rangle$, a sequence of ps-query-answer pairs, $1 \leq i \leq n$. Let T be a data tree over Σ and N a set of nodes:*

(i) *it is* NP*-hard to determine whether T is the prefix relative to N of some $T' \in rep(\tau)$ such that $A_i = q_i(T')$, $1 \leq i \leq n$.*

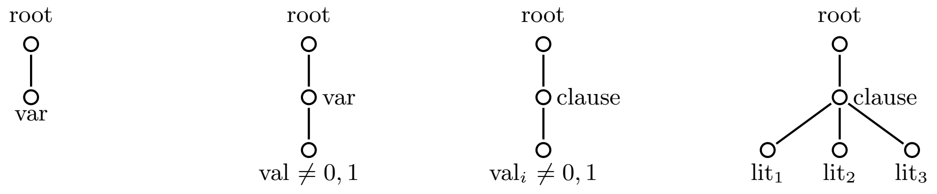
(ii) *it is* CO-NP*-hard to determine whether T is a prefix relative to N of every tree T' as in (i), up to node identifiers.*

Proof: The proof of (i) is by reduction from 3-SAT. Let $\mathcal{C}$ be a set of clauses, each with three literals. The input tree type is:



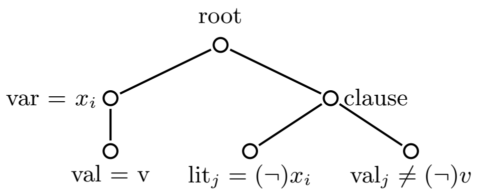
Each clause subtree is supposed to encode a complete clause in $\mathcal{C}$ (i.e. a disjunction of 3 literals). The values of the lit nodes will be literals (x_i or $\neg\ x_i$). The val$_i$ node is supposed to hold a value for literal lit$_i$, $i = 1, 2, 3$.

Now ask these queries:

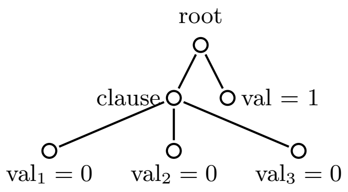


returns n nodes $x_1 \cdots x_n$ (all variables)	empty answer	$i = 1, 2, 3$ empty answer	returns the clauses in C

where the answers are as described. So we know there is one value 0 or 1 per variable, that the encoding of the clauses is right, and the value of each literal is 0 or 1. Next, for each variable x_i, value v (0 or 1), literal $(\neg)x_i$ and compatible value $(\neg)v$ for the literal, ask this query:

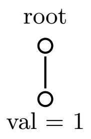


and suppose all answers are empty. This means that all literals in clauses have the right values, given the valuations for the variables. Next, ask the query:



and suppose all answers are empty. This means that *val* cannot equal 1 unless all clauses have at least one literal that is 1. It follows that $\mathcal{C}$ is satisfiable if and only if it is possible for *val* to equal 1.

Thus checking that the tree



is a possible prefix (relative to $N = \emptyset$) given the above sequence of query-answer pairs is equivalent to checking that $\mathcal{C}$ is satisfiable. The result follows.

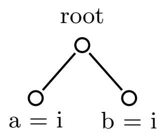
The proof of (ii) is similar and is omitted. □

Remark 3.7 *One might wonder if a matching* NP *upper bound can be established for the problem (i) shown* NP*-hard in Theorem 3.6, and similarly a* CO-NP *upper bound for (ii). This can indeed be done, as discussed in Remark 3.11.*

3.2 Avoiding the exponential blowup

The exponential blowup of incomplete trees is illustrated by the following example.

Example 3.2 Consider ps-queries q_i, $1 \leq i \leq n$, of the form:



whose answers are empty. The incomplete tree constructed for these queries by Algorithm *Refine* yields a set of 2^n specialized types, one for each choice of inequalities $a \neq i$ or $b \neq i$ for $i \in [1, n]$.

We consider two ways of avoiding the exponential blow-up of incomplete trees:

1. by allowing conjunctions of disjunctions of multiplicity atoms in the definition of the incomplete tree, and

2. by restricting the ps-queries.

We also propose two heuristics for dealing with incomplete trees that become too large, regardless of the complexity-theoretic bound.

We first discuss extensions of incomplete trees with conjunction, and restrictions of the ps-queries.

Conjunctive incomplete trees. It is possible to prevent the exponential blowup of incomplete trees by allowing conjunctions of disjunctions of multiplicity atoms in type specifications, rather than just disjunction. The meaning of a conjunction of disjunctions of multiplicity atoms is that the tree described must be simultaneously valid with respect to *all* types specified by each conjunct. In terms of automata, this is analogous to allowing alternation rather than just nondeterminism in the control. We refer to incomplete trees augmented with conjunction as *conjunctive incomplete trees.*

When conjunctive incomplete trees are allowed it is possible to simplify Algorithm *Refine* in the following way. The first step, which computes the incomplete tree for $q^{-1}(A)$, remains the same (Lemma 3.2). Recall that the complexity of this step is in $O((|A| + |q|) \cdot |\Sigma|)$. The second step, which performs an intersection of two incomplete trees, is done as described in Lemma 3.3, except that the step of combining disjunctions of multiplicity atoms now becomes trivial: We simply take their conjunction. Thus the intersection of two conjunctive incomplete trees $\mathbf{T}_1$ and $\mathbf{T}_2$ can be done in polynomial time and yields an output of size $O(|\mathbf{T}_1| + |\mathbf{T}_2|)$. Let *Refine*$^+$ be the resulting algorithm. From this discussion we conclude:

Theorem 3.8 *Given a conjunctive incomplete tree* $\mathbf{T}$ *and a ps-query* q *with answer* A*, Algorithm* Refine$^+$ *computes in polynomial time an incomplete tree* $\mathbf{T}'$ *such that* $rep(\mathbf{T}') = rep(\mathbf{T}) \cap q^{-1}(A)$*. Moreover the size of* $\mathbf{T}'$ *is* $O(|\mathbf{T}| + (|A| + |q|) \cdot |\Sigma|)$.

The usefulness of conjunction in incomplete trees is illustrated by Example 3.2. The incomplete information provided by the query-answer pairs can be represented concisely using a conjunctions of n disjunctions:

$$
\begin{array}{lcl}
root & \rightarrow & (a_1^* b^* \;\vee\; a^* b_1^*) \wedge \cdots \wedge (a_n^* b^* \;\vee\; a^* b_n^*) \\
a & \rightarrow & a^* b^* \\
b & \rightarrow & a^* b^* \\
a_i & \rightarrow & a^* b^*,\ 1 \leq i \leq n \\
b_i & \rightarrow & a^* b^*,\ 1 \leq i \leq n \\
cond(a_i) & = & \neq\ i,\ 1 \leq i \leq n \\
cond(b_i) & = & \neq\ i,\ 1 \leq i \leq n \\
cond(a) & = & cond(b) = true
\end{array}
$$

where each a_i and b_i specialize a and b, respectively. As discussed earlier, without conjunction, Algorithm *Refine* yields a disjunction of 2^n multiplicity statements, corresponding to the DNF form of the n conjuncts.

This example can be generalized using the result of Theorem 3.8, as shown next.

Corollary 3.9 *Let $\langle q_i, A_i \rangle_{1 \leq i \leq n}$ be a sequence of ps-query-answer pairs. Let $\mathbf{T}$ be the conjunctive incomplete tree constructed by repeated applications of Algorithm* Refine$^+$ *to the ps-query-answer pairs. Then the size of $\mathbf{T}$ is polynomial in $\langle q_i, A_i \rangle_{1 \leq i \leq n}$.*

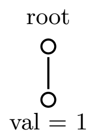
Proof: By induction and Theorem 3.8, one can easily show that the size of $\mathbf{T}$ after query i is $O((|A_1| + |q_1| + \cdots + |A_i| + |q_i|) \cdot |\Sigma|)$. □

The price to pay for the conciseness of conjunctive trees is increased complexity of various manipulations.

To illustrate the increase in the complexity of manipulating conjunctive incomplete trees compared to regular incomplete trees, we consider the key problem of checking non-emptiness of a conjunctive incomplete tree. This problem becomes NP-complete, whereas it is polynomial for usual incomplete trees (see Lemma 2.5).

Theorem 3.10 *Given a conjunctive incomplete tree $\mathbf{T}$, it is* NP*-complete whether $rep(\mathbf{T}) \neq \emptyset$.*

Proof: The NP-hardness of non-emptiness follows immediately from the proof of Theorem 3.6, as shown next. Recall the proof of NP-hardness of (i) in Theorem 3.6. Consider the tree type and query-answer pairs used in the proof. Let $\mathbf{T}_0$ be the conjunctive incomplete tree constructed from the tree type and the query-answer pairs shown there. Next, construct the conjunctive incomplete tree $\mathbf{T}$ for the intersection of $\mathbf{T}_0$ with the simple conditional tree type stating that the answer to the query



is not empty. Clearly, $\mathbf{T}$ is polynomial in $\mathcal{C}$ and $rep(\mathbf{T}) \neq \emptyset$ if and only if $\mathcal{C}$ is satisfiable. This establishes the NP-hardness of checking non-emptiness of conjunctive incomplete trees.

For the upper bound, consider a conjunctive incomplete tree $\mathbf{T} = (N, \lambda, \nu, \tau)$, where $\tau = (\Sigma', R, \mu, \text{cond}, \sigma, \Sigma)$. Let π be a mapping associating to each $\alpha \in \Sigma'$ a choice of one disjunct from each conjunct in $\mu(\alpha)$. Let $\mathbf{T}_\pi$ be the (regular) incomplete tree obtained by defining, for each $\alpha \in \Sigma'$, $\mu(\alpha)$ to be the join of the disjuncts in $\pi(\alpha)$, where the join of disjuncts is defined as the $\bowtie$ operation in the proof of Lemma 3.3. It is easily seen that $rep(\mathbf{T}) \neq \emptyset$ if and only if there exists a mapping π such that $rep(\mathbf{T}_\pi) \neq \emptyset$. Thus, to test whether $rep(\mathbf{T}) \neq \emptyset$, first nondeterministically guess π and construct $\mathbf{T}_\pi$ from $\mathbf{T}$ in polynomial time. Finally, test whether $rep(\mathbf{T}_\pi) \neq \emptyset$ in polynomial time, by Lemma 2.5. This yields the NP upper bound for testing non-emptiness of conjunctive incomplete trees. □

Remark 3.11 *Theorem 3.10 can be used to obtain an* NP *upper bound to (i) in Theorem 3.6, and analogously a* CO-NP *upper bound for (ii). Let τ be a tree type over a fixed alphabet Σ, and $\langle q_i, A_i \rangle_{1 \leq i \leq n}$, a sequence of ps-query-answer pairs. Let T be a data tree over Σ and N a* *set of nodes. To show the* NP *upper bound for (i) in Theorem 3.6, first construct a conjunctive incomplete tree* $\mathbf{T}_1$ *from* $\langle q_i, A_i\rangle_{1\leq i\leq n}$ *and* τ. *This can be done in* PTIME *by Corollary 3.9. Next, construct in* PTIME *an incomplete tree* $\mathbf{T}_2$ *defining the set of trees over* Σ *for which* T *is a prefix relative to* N. *Finally, construct in* PTIME *a conjunctive incomplete tree* $\mathbf{T}$ *such that* $rep(\mathbf{T}) = rep(\mathbf{T}_1) \cap rep(\mathbf{T}_2)$. *Clearly,* T *satisfies (i) if and only if* $rep(\mathbf{T}) \neq \emptyset$. *The latter is in* NP *by Theorem 3.10. Thus, the entire procedure is in* NP, *which provides the* NP *upper bound for (i). The* CO-NP *upper bound for (ii) follows analogously.*

Restricting the queries. A second approach to avoiding the blowup in the size of incomplete trees is to restrict the ps-queries. We exhibit one restriction that, although quite drastic, is likely to be reasonable in many practical situations: the ps-queries consist of a single path (so each node has a single child or is a leaf). We call such ps-queries *linear*. The following lemma shows why the situation is simpler for linear ps-queries.

Lemma 3.12 *Let* $(q_1, A_1), \ldots, (q_n, A_n)$ *be a sequence of linear ps-queries and their answers. It is possible to construct in* PTIME *an incomplete tree representing* $q_1^{-1}(A_1) \cap \cdots \cap q_n^{-1}(A_n)$ *whose size is polynomial in* $|A_1| + |q_1| + \cdots + |A_n| + |q_n|$.

Proof: (sketch) The general idea is the following. Because the queries are linear, the number of conditions describing trees not belonging to the output of a query is linear in the depth of the query, with only one condition per level. Therefore, at each level, the number of tests needed in Algorithm *Refine* remains polynomial in the total number of queries.

Recall from Lemma 3.2 the construction of $q^{-1}(A)$, the incomplete tree representing the set of all data trees T such that $q(T) = A$. In particular, recall the definition of $\bar{\tau}_m$ and $\hat{\tau}_m$ for nodes m of the query tree t_q.

Recall that for m in t_q with children $m_1 \cdots m_l$, we have set $\mu(\hat{\tau}_m) = \bigvee_{1\leq i\leq l} \alpha_i$ where α_i is the multiplicty atom $\bar{\tau}^\star_{m_i}\hat{\tau}^\star_{m_i}\text{else}_i$, which was the only use of disjunction. If q is linear, then $l = 1$, so there is no disjunction. In other words, for a linear ps-query q, $q^{-1}(A)$ contains no disjunction.

Recall now the rest of the construction. Besides the τ_n for $n \in N$, all the other types are mapped to $\star$ by the multiplicity atoms and all types associated to the same label at a given depth (that is $\bar{\tau}_m$ and $\hat{\tau}_m$ for some m) have conditions that give a partition of $\mathbb{Q}$ (unambiguity).

With these properties in mind we now turn to the construction of the incomplete tree for $q_1^{-1}(A_1) \cap \cdots \cap q_n^{-1}(A_n)$.

Let $q_i^{-1}(A_i)$ be $(N_i, \lambda_i, \nu_i, \tau_i)$. We construct an incomplete tree (N, λ, ν, τ) representing $q_1^{-1}(A_1) \cap \cdots \cap q_n^{-1}(A_n)$. The construction of N, λ, ν is done as in Lemma 3.3. Notice that $|N| \leq |N_1| + \cdots + |N_n|$. It remains to construct τ.

We prove by induction on n that (i) the size of τ is linear in $|\tau_1| + \cdots + |\tau_n|$, (ii) τ contains no disjunction in multiplicity atoms, (iii) apart from τ_n where $n \in N$, all types are mapped to $\star$ in the multiplicity atoms, and (iv) the conditions associated to the types with the same label in a multiplicity atom define a partition of $\mathbb{Q}$ into intervals.

We already dealt with the case $n = 1$. Assume the claim proven for $n - 1$ and consider a new query q_n. We revisit the proof of Lemma 3.3 that computes the intersection of two unambiguous incomplete trees and show that in the particular case of q_n and $q_1^{-1}(A_1) \cap \cdots \cap q_{n-1}^{-1}(A_{n-1})$, we can derive the inductive properties.

Recall that the algorithm for computing intersection basically computes a new type for each pair of *compatible* types (τ_1, τ_2), one from each incomplete tree. Moreover this new type is reachable from the type of the root if both τ_1 and τ_2 occur at the same depth in the incomplete tree.

Now fix a depth d (and therefore a letter a by linearity of q_n) and consider the set S_d of all compatible types (τ_1, τ_2) for the corresponding letter and depth. Consider all conditions associated to each of these types. By Lemma 2.3 there is a partition I_d of $\mathbb{Q}$ of size linear in $|S_d|$ such that every condition is always true or always false on each interval. For each interval $u \in I_d$ we create a new type τ_u^d.

By linearity of q_n we have constructed a linear number of new types and, for each d, the condition of q_n at depth d is either always true or always false over the interval specified by the condition of any type τ_u^d. Using the truth value of the condition, it is easy to propagate in the multiplicity atom of each τ_u^d the fact that the rest of the query has to verify or not the part of q_n which is below d. $\square$

Heuristics. We next sketch two approaches for dealing with cases when the incomplete tree grows too large to be practical.

The first approach consists of asking a small set of additional queries chosen so as to provide precisely the critical information needed to eliminate some of the unknown information and shrink the incomplete tree. The choice of additional queries may be guided by various heuristics, which can be applied whenever the incomplete tree becomes too large. As shown next, there is a standard choice of additional queries that always keeps the incomplete tree polynomial in size. While the resulting incomplete tree may become in the worst case as large as the entire input data tree, in other cases it can remain much smaller.

Proposition 3.13 *For each input tree T and sequence of query-answer pairs $\langle q_1, A_1\rangle \dots \langle q_k, A_k\rangle$ there exist queries $q'_1 \dots q'_l$, with answers $A'_1, \dots, A'_l$, such that:*

(i) $l \leq O(|q_1| + \cdots + |q_k|)$,

(ii) for each i there exists j such that $|q'_i| \leq |q_j|$,

(iii) the incomplete tree $\mathbf{T}$ *constructed by Algorithm* Refine *for*

$$\langle q_1, A_1\rangle \dots \langle q_k, A_k\rangle, \langle q'_1, A'_1\rangle \dots \langle q'_l, A'_l\rangle$$

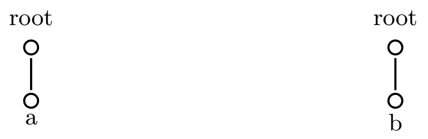
is polynomial in the size of query-answer pairs.

Proof: The main idea is to request a data value as soon as Algorithm *Refine* produces a disjunction using that data value. Providing the actual value eliminates the need for a case analysis represented by the disjunction.

For each query q_i and each node m in its tree pattern, ask the query q_m consisting of the path from root to m, with all conditions set to *true*. Furthermore, query q_m is asked before q_n if n is a child of m in q_i. This retrieves all data nodes with labels mentioned in the queries at a given level of the input tree. When Algorithm *Refine* is run, this results in eliminating from multiplicity atoms all types of the form $\bar{\tau}_m$ and $\hat{\tau}_m$ (because $\bar{\tau}_m$ has associated condition *false* and $\hat{\tau}_m$ is not used because m is a leaf in q_m). Thus, Algorithm *Refine* produces the incomplete tree (N, λ, ν, τ) defined as follows. $N = A_1 \cup \dots A_k \cup A'_1 \cup \dots A'_l$ and λ, ν are defined in the obvious way. The conditional tree type $\tau = (\Sigma', R, \text{cond}, \mu, \sigma, \Sigma)$ contains one type τ_a

for each $a \in \Sigma$ and one type τ_n for each $n \in N$; $\sigma(\tau_n) = \lambda(n)$, $\sigma(\tau_a) = a$, $\text{cond}(n) = \nu(n)$, and $\text{cond}(\tau_a) = \textit{true}$, for $a \in \Sigma$ and $n \in N$. Finally, μ is defined as follows. For $a \in \Sigma$, $\mu(\tau_a) = \text{all}^\star$. If $n \in N$ and n is a leaf, then $\mu(\tau_n) = \text{all}^\star$. If n is not a leaf, let $n_1 \dots n_k$ be its children and $\mu(\tau_n) = \tau_{n_1} \dots \tau_{n_k}\text{else}_n^\star$ where else_n contains $\tau_a^\star$ for all $a \in \Sigma$ such that a is not a label of any of the children of n. Clearly, the incomplete tree above is polynomial in the size of the query-answer pairs. □

Example 3.3 Consider again the queries in Example 3.2, for which the incomplete tree constructed by Algorithm *Refine* is exponential in the query-answer pairs. The following two additional queries result in a polynomial-size incomplete tree.



Intuitively, these queries settle the choice of conditions that previously led to the exponential-size incomplete tree. Note however that this does not guarantee that the new incomplete tree is actually smaller than the original, but only that it is polynomial with respect to the extended sequence of query-answer pairs. Thus, this approach may or may not be beneficial.

The second approach for dealing with large incomplete trees is to gracefully loose some of the information in order to shrink the incomplete tree. The idea is illustrated by Example 3.2. Intuitively, the incomplete tree becomes large because it enumerates explicitly the restrictions on the pairs of values for a and b, which are tested repeatedly by the queries. This makes it expensive to maintain the information about the connection between the a and b values. One way around this is to "forget" the costly information on the connection between the values, and only retain the ranges of allowed values for a and for b. In general, the expensive combinations of values can be identified by a scoring system maintained dynamically as queries are asked. This approach can be extended to combinations of type specializations, which may involve constraints on both the data values and structure of the allowed trees. Yet another approach is to replace some of the specialized types by their unspecialized versions. In the worst case, this reverts back to τ. Further exploration of such heuristics is left for future work.

3.3 Querying incomplete trees

We next show how incomplete trees can be used to answer queries. We look at two distinct issues. The first is the classical problem of answering a query using only the information provided by the incomplete tree. The answer is itself an incomplete tree, providing a description of the possible answers. This first issue is addressed in the present section. The second issue, addressed in next section, is using incomplete trees as a guide for a mediator that must decide what new queries should be asked on the source document in order to provide a complete answer to a user query.

We now consider the first of the questions mentioned above. Suppose our knowledge of the world consists of an incomplete tree $\mathbf{T}$. This means that the possible data trees are in $rep(\mathbf{T})$. Suppose we wish to answer a ps-query q. The possible answers for q are the data trees in $q(rep(\mathbf{T}))$. Incomplete trees form a *strong representation* system for ps-queries if the set $q(rep(\mathbf{T}))$ can be described using an incomplete tree for arbitrary q and $\mathbf{T}$. Indeed, we show that such an incomplete tree exists, and denote it $q(\mathbf{T})$. Moreover, $q(\mathbf{T})$ is polynomial with respect to q and $\mathbf{T}$ for fixed alphabet Σ.

Theorem 3.14 *Let Σ be a fixed set of labels. Given an incomplete tree $\mathbf{T}$ and a ps-query q over Σ, one can construct an incomplete tree denoted $q(\mathbf{T})$ such that*

$$rep(q(\mathbf{T})) = \{q(T) \mid T \in rep(\mathbf{T})\} = q(rep(\mathbf{T})).$$

Furthermore, $q(\mathbf{T})$ can be constructed in PTIME *with respect to q and $\mathbf{T}$ (exponential in Σ).*

Proof: The incomplete tree $q(\mathbf{T})$ is constructed by a careful Cartesian product of $\mathbf{T}$ and q.

Let $\mathbf{T}= (N, \lambda, \nu, \tau)$ where $\tau = (\Sigma', R, \mu, \text{cond}, \sigma, \Sigma \cup N)$ be an incomplete tree and $q = \langle t_q, \lambda_q, \text{cond}_q\rangle$ be a ps-query.

Assume for simplicity that q contains no nodes of type $\bar{a}$, a case that is easily handled.

For each node m in t_q, let q_m be the query consisting of the subtree of t_q rooted at m, with the same label and conditions as in q. Let $Cert(m)$ be the set of types on which q_m will *certainly* produce an output, that is the set of types $\tau \in \Sigma'$ such that $q_m(T) \neq \emptyset$ for every $T \in rep(\mathbf{T}_\tau)$, where $\mathbf{T}_\tau$ is the same as $\mathbf{T}$ except that the root set is $\{\tau\}$. Analogously let $Poss(m)$ be the set of types on which q_m will possibly produce an output, that is the set of types $\tau \in \Sigma'$ such that $q_m(T) \neq \emptyset$ for at least one $T \in rep(\mathbf{T}_\tau)$. In can be shown, similarly to the proof of Theorem 2.8, that $Cert(m)$ and $Poss(m)$ can be computed in time polynomial with respect to $\mathbf{T}$ and q.

The incomplete tree $q(\mathbf{T})$ is (N, λ, ν, τ') where $\tau' = (\Sigma'', R', \mu', \text{cond}', \sigma', \Sigma \cup N)$, defined as follows:

- $\Sigma'' = \Sigma' \times nodes(t_q)$;
- $R' = \{\langle \tau, root(t_q)\rangle \mid \tau \in R \cap Poss(root(t_q))\}$;
- $cond'(\langle \tau, m\rangle) = cond(\tau) \wedge cond_q(m)$;
- $\sigma'(\langle \tau, m\rangle) = \sigma(\tau)$.

The mapping μ' is defined as follows. Let $\langle \tau, m\rangle$ be in Σ'' where $m \in nodes(t_q)$, and let $m_1 \ldots m_k$ be the children of m in t_q. Then $\mu'(\langle \tau, m\rangle)$ consists of the following multiplicity atoms, obtained by modifying or discarding multiplicity atoms in $\mu(\tau)$. If $\alpha \in \mu(\tau)$ and there exists a child m_i of m such that α contains no τ^ω where $\tau \in Poss(m_i)$, then α is discarded. Otherwise, α is modified as follows. First, if τ^ω is in α and there is no m_i with the same label as τ such that $\tau \in Poss(m_i)$, then τ^ω is eliminated from α.

At this stage, α is of the form $\alpha_1 \ldots \alpha_k$ where α_i consists of all τ^ω where τ has the same label as m_i. Let α' be obtained by replacing, in each α_i, τ^ω by $\langle \tau, m_i\rangle^\omega$, yielding $\alpha'_1 \ldots \alpha'_k$.

We now focus on the multiplicities of the types. For each m_i the query requires at least one node with the same label. The incomplete tree may have several types τ for that label with different multiplicities, and each τ may be in $Poss(m_i)$ or in $Cert(m_i)$. We take this into account as follows. First, for each $(\tau')^\omega = \langle \tau, m_i\rangle^\omega$ in α'_i, we replace $(\tau')^1$ by $(\tau')^?$ and $(\tau')^+$ by $(\tau')^\star$ for each $\tau \in Poss(m_i) - Cert(m_i)$. Secondly we require that at least one type per label occurs at least one. Let us say that α'_i is possibly empty if α'_i contains only types with multiplicity in $\{?, \star\}$. Conceptually, we replace each possibly empty α'_i by the disjunction of all multiplicity atoms obtained by replacing one of the multiplicities ? or $\star$ by 1 or $+$, respectively.

The new set of multiplicity atoms α' for $\mu'(\langle\tau, m\rangle)$ generated by α corresponds to the disjunctive normal form of the $\alpha'_1 \ldots \alpha'_k$ modified as above. Note that this construction generally yields a set of multiplicity atoms exponential in Σ.

It is straightforward to verify that the constructed $q(\mathbf{T})$ satisfies the statement. □

As a very useful consequence, we can decide if a ps-query q can be fully answered using the information available after a sequence of query-answer pairs. More specifically, let us call an incomplete tree *reachable* if it is obtained by repeated applications of Algorithm *Refine* from a sequence of query-answer pairs, further modified to capture the initial tree type, as done in the proof of Theorem 3.5.

Reachable incomplete trees have several useful properties. In particular, each type corresponding to a data node has multiplicity one, and the subtree of the data nodes is a prefix of every data tree in $rep(\mathbf{T})$. Let us refer to this subtree as the data tree of $\mathbf{T}$, denoted T_d. The above question can now be formulated as follows: does T_d contain enough information to fully answer a new ps-query q? The answer is provided next.

Corollary 3.15 *Let Σ be a fixed set of labels. Let $\mathbf{T}$ be a reachable incomplete tree over Σ, with data tree T_d, and q a ps-query over Σ. It is decidable in* PTIME *(exponential in Σ) whether q can be fully answered using $\mathbf{T}$, i.e. whether for every $T \in rep(\mathbf{T})$, $q(T) = q(T_d)$.*

Proof: We first compute the incomplete tree $q(\mathbf{T})$, as outlined in the proof of Theorem 3.14. Next, we have to verify that all types τ used in $q(\mathbf{T})$ that do not specialize a data node are empty. The required tests of emptiness can be done in PTIME with respect to $q(\mathbf{T})$ (see Lemma 2.5). □

Remark 3.16 *As an important side effect, Corollary 3.15 in combination with Theorem 3.4 provide a way to check if a ps-query q can be answered using the views provided by a sequence of ps-query-answer pairs. The problem of answering queries using views has been studied recently in other contexts (see related work).*

There are several important variants of the query answering problem with incomplete information, such as deciding if certain facts are *certain* or *possible* in the answers to a given query. The following is an immediate consequence of Theorems 2.8 and 3.14.

Theorem 3.17 *Let Σ be a fixed set of labels. Given an incomplete tree $\mathbf{T}$, a ps-query q, and a data tree T over Σ, it can be checked in* PTIME *(exponential in Σ) whether T is a certain prefix or whether T is a possible prefix of $q(\mathbf{T})$.*

From Theorem 3.17 we derive immediately the following interesting corollary:

Corollary 3.18 *Let Σ be a fixed set of labels. Given an incomplete tree $\mathbf{T}$ and a ps-query q over Σ, the following can be checked in* PTIME *(exponential in Σ):*

(possible non-emptiness) $q(T) \neq \emptyset$ *for* some *tree $T \in rep(\mathbf{T})$; and,*

(certain non-emptiness) $q(T) \neq \emptyset$ *for* every *tree $T \in rep(\mathbf{T})$.*

3.4 Guiding mediators

We consider here the following problem: Suppose we have partial information about the input document(s) specified as an incomplete tree, and the user poses a query against the virtual input document. If we are lucky, we may be able to provide the complete answer to the query using the information available. Otherwise, additional queries may have to be generated against the input document to obtain the information needed to fully answer the query. The incomplete tree can be used as a guide to generate such queries. We assume that the generated queries further explore the input document starting from the nodes already available. We refer to such queries as *local*. In order to generate local queries intelligently, we must determine which sources possibly (or certainly) contain information relevant to the query. Corollary 3.18 states that these questions can be effectively answered.

Example 3.4 (**More catalog queries**) We illustrate the use of incomplete trees to answer queries. Continuing with the catalog Example 2.1, suppose the following query arrives following Query 1 and Query 2:

- Query 3: find the name, price and pictures of all cameras costing less than $100 and having at least one picture.

Clearly, we can answer this query fully using just the information available locally. More interestingly, suppose we ask the query:

- Query 4: list all cameras.

While we are not able to provide the complete answer, we can do the following:

1. provide the complete list of cameras that are less than $200 or have a picture;

2. tell the user that there may be more cameras (that are expensive and have no pictures).

Thus, we can provide an incomplete answer to the query given the knowledge available, without accessing the data source for further information.

If accessing the source is possible and desired, we can augment our result by issuing the following query:

- Query 5: find the name and price of cameras that cost at least $200.

We now consider in more detail the generation of local queries. We show that we can always efficiently compute a set of local queries that collect the additional information allowing to answer a given ps-query. More formally, let $\mathbf{T}$ be a reachable incomplete tree with data tree T_d and q a ps-query. A *local* ps-query is an expression of the form $p@n$ where p is a ps-query and n is a node in T_d. The query returns the answer to p on the subtree of the full input tree rooted at n. Consider a set L of local queries against $\mathbf{T}$. We say that L *completes* $\mathbf{T}$ *relative to* q if for each $T \in rep(\mathbf{T})$, $q(T)$ equals $q(T')$ where T' is obtained by extending each node n of T_d for which $p@n \in L$, with $p@n(T)$.

Obviously, the query q itself posed at the root is always a trivial completion of $\mathbf{T}$ relative to q. The point in using local queries is to avoid doing the work already done by previous queries. Therefore, we would like the completion L to avoid, as much as possible, retrieving nodes that already exist in $\mathbf{T}$. We would also like the following properties: (i) that the answers to distinct queries in L not overlap, and, (ii) that L should not contain queries that always return empty answers on the possible input trees. If L has properties (i) and (ii), we call it *non-redundant.* The following shows that non-redundant completions can always be found.

Theorem 3.19 *Let* $\mathbf{T}$ *be a reachable incomplete tree, and* q *a ps-query. One can construct in* PTIME *a set of local queries* L *which forms a non-redundant completion of* $\mathbf{T}$ *relative to* q.

Proof: Consider $\mathbf{T}$ and q. For each node m in the query tree of q, let q_m be the query consisting of the subtree of q rooted at m, with the same conditions as in q. As in the proof of Theorem 3.14, define $Poss(q_m)$ be the set of types τ of $\mathbf{T}$ such that $q_m(T) \neq \emptyset$ for some $T \in rep(\mathbf{T}_\tau)$, where $\mathbf{T}_\tau$ is the same as $\mathbf{T}$ except that the root set is $\{\tau\}$. Recall that $Poss(q_m)$ can be computed in time polynomial with respect to $\mathbf{T}$ and q.

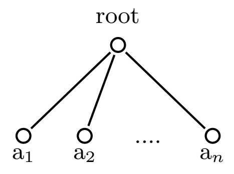
Let T_d be the data tree of $\mathbf{T}$. The completion L is generated recursively as follows. First, initialize L to $q@r$ where r is the root of T_d. Next, recursively apply the following step. Suppose that L contains a set of local queries. Consider a query $p@n$ in L. If the tree of p consists of more than just the root, let the children of the root be $m_1, \ldots, m_k$. Let τ_n be the type corresponding to n in $\mathbf{T}$. Let C be the set of m_i such that some multiplicity atom α in $\mu(\tau_n)$ contains τ_i^ω where $\tau_i \in Poss(p_{m_i})$ and τ_i is not the type of a data node (intuitively, this means that part of the answer to p_{m_i} can be obtained from the missing information under n). Let p_C be the query obtained from p by eliminating the subtrees rooted at nodes m_i not in C. Replace $p@n$ by the following local queries: (i) $p_C@n$ (ii) all queries of the form $p_{m_i}@n_i$ where n_i is a child of n in T_d whose type is in $Poss(p_{m_i})$, for $m_i \notin C$, $1 \leq i \leq k$. The procedure ends when it is no longer possible to replace a query in L. It is easy to see that the resulting set L forms a non-redundant completion of $\mathbf{T}$ relative to q. $\square$

Although non-redundancy of completions is an appealing property, enforcing it clearly comes at a cost. In practice, other parameters are likely to also be taken into account in order to generate efficient completions.

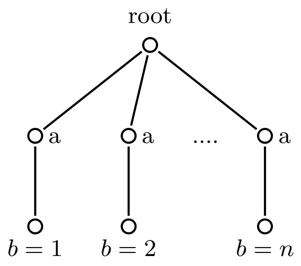
4 Extensions

Our framework for XML documents with incomplete information relies on many limitations and assumptions, such as the availability of persistent node ids, the lack of order, and a very simple query language. In this section we discuss several extensions to our framework, and their impact on handling incomplete information. A comprehensive study of possible extensions is beyond the scope of this paper. Instead, we illustrate the kinds of difficulties that various extensions may introduce. Many of the definitions in this section are informal. We begin by discussing several extensions to ps-queries, and their impact on handling incomplete information.

Branching. Recall that ps-query tree patterns allow just one child with a given label for each node in the pattern. For instance, this disallows a query whose pattern looks simultaneously for a product that has a picture and another that is a camera. Branching allows multiple children with the same label. Incomplete trees remain a strong representation system for ps-queries extended with branching and can be maintained incrementally in PTIME. However, if $\mathbf{T}$ is a reachable incomplete tree and q a ps-query with branching, $q(\mathbf{T})$ may now be exponential with respect to $\mathbf{T}$, even for fixed Σ. For example, if the data tree of $\mathbf{T}$ is (a), where $a_1 \ldots a_n$ are specializations of a, and q is the ps-query (b) with branching,



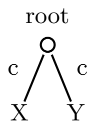
(a)



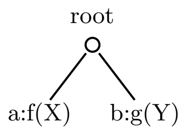
(b)

then the incomplete tree for $q(\mathbf{T})$ has to describe $n!$ possibilities of assigning the n values of b to $a_1, \ldots a_n$.

Branching and constructed answers. Queries with constructed answers consist of a *body* and a *head*. As for ps-queries, the body is a tree pattern. However, the nodes in the pattern are labeled by variables. For each input, the body defines a set of bindings of the variables to input nodes. The head of the query specifies how to construct an answer data tree from the bindings. This is done in the spirit of XML-QL, using Skolem functions. Incomplete trees are no longer a strong representation system for ps-queries with branching and constructed answers. For example, the query:



body



head

produces answers with equal numbers of a's and b's (one a for each binding of X and one b for each binding of Y), which cannot be precisely described by incomplete trees. In fact, the existence of a strong representation system for such queries remains open.

Branching and optional subtrees. Queries with optional subtrees allow labeling some subtrees by "?". The semantics is that a valuation is now a partial mapping, not required to be defined on the nodes of the optional subtrees. For example, this allows requesting cameras and displaying their pictures if they exist. However, a camera is in the answer even if it has no picture. The combination of branching and optional subtrees yields an exponential blowup in complexity for several questions we considered. We illustrate this behavior using a variant of the *certain prefix* question. Note that, by Theorems 3.4, 3.14, and 2.8, we can check in PTIME (for fixed Σ) whether a tree T is a certain or possible prefix for the answers of a ps-query q' on trees compatible with a given input tree type τ and a *single* ps-query-answer pair $\langle q, A\rangle$. For ps-queries extended with branching and optional subtrees, we can show the following for a fixed Σ (of size 4):

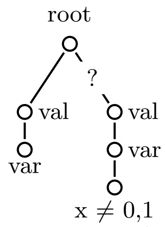
Theorem 4.1 *Given a tree T, a tree type τ, a query-answer pair $\langle q, A\rangle$, and a query q', where q and q' are ps-query with branching and optional subtrees, it is* CO-NP*-hard whether T is a certain prefix for*

$$q'[rep(\tau) \ \cap \ q^{-1}(A)].$$

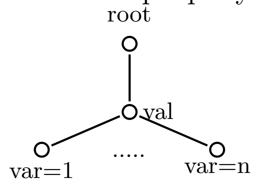
Proof: The proof of CO-NP-hardness is by reduction of validity of DNF formulas with 3 variables per disjunct. Let $\mathcal{D}$ be a set of disjuncts $\{d_1, \cdots, d_k\}$, each with 3 literals, and using variables $x_1, \cdots, x_n$. Consider the input tree type defined by:

$$\begin{array}{lcl} root & \rightarrow & val \\ val & \rightarrow & var^* \\ var & \rightarrow & x \end{array}$$

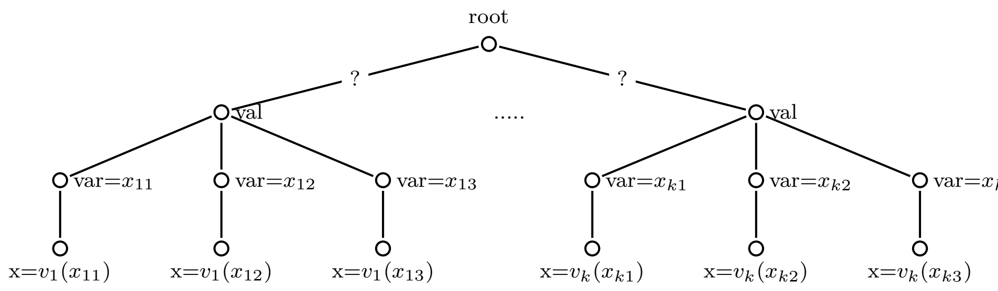
Intuitively, if var $= i$, then its child x represents variable x_i, $1 \leq i \leq n$. Consider the ps-query q with branching and optional subtrees:



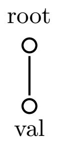
whose answer A is:



This indicates that there is exactly one representative for each x_i and the corresponding x has value 0 or 1. For each disjunct $d_i \in \mathcal{D}$, let x_{i1}, x_{i2}, and x_{i3} be the three variables occurring in d_i and let v_i be the valuation on these variables defined by $v_i(x_{ij}) = 1$ if x_{ij} occurs in d_i, and $v_i(x_{ij}) = 0$ if $\neg\, x_{ij}$ occurs in d_i, $1 \leq j \leq 3$. Now consider the query q':



Clearly, $\mathcal{D}$ is valid if and only if the tree:



is a certain prefix for the answers to q', i.e., for the set $\{q'(T) \mid T \in rep(\tau) \ \cap \ q^{-1}(A)\}$. $\Box$

Note that this complexity lower bound is independent of a particular representation system.

Pebble transducers. It turns out that our framework can be extended to *ordered* trees and very powerful restructuring queries, as long as data joins are not allowed. We illustrate this using the k-pebble tree transducers introduced by Milo, Suciu, and Vianu [35] to model a wide range of XML transformation languages. We provide next an informal description of k-pebble transducers.

A k-pebble transducer takes as input a binary tree and outputs a binary tree, whereas trees in our model are unranked. However, since unranked ordered trees have a standard representation as binary trees [34], this mismatch can be easily overcome. The k-pebble transducer uses up to k pebbles to mark certain nodes in the tree. Transitions are determined by the current node symbol, the current state, and by the existence or absence of the various pebbles on the node. The pebbles are ordered and numbered $1, 2, \ldots, k$. The machine can place pebbles on the root, move them around, and remove them. In order to limit the power of the transducer, the use of pebbles is restricted by a stack discipline: pebbles are placed on the tree in order and removed in reverse order, and only the highest-numbered pebble present on the tree can be moved.

The transducer works as follows. The computation starts by placing pebble 1 on the root. At each point, pebbles $1, 2, \ldots, i$ are on the tree, for some $i \in \{1, \ldots, k\}$; pebble i is called the *current pebble*, and the node on which it sits is the *current node*. The current pebble serves as the head of the machine. The machine decides which *transition* to make, based on the following information: the current state, the symbol under the current pebble, and the presence or absence of the other $i - 1$ pebbles on the current node. There are two kinds of transitions: *move* and *output* transitions. *Move* transitions are of four kinds: they can place a new pebble on the root, pick the current pebble, or move the current pebble in one of the four directions *down-left, down-right, up-left, up-right* (one edge only). If a move in the specified direction is not possible, the transition does not apply. After each move transition the machine enters a new state, as specified by the transition.

An *output* transition emits some labeled node and does not move the input head. There are two kinds of output transitions. In a *binary* output, the machine spawns two computation branches computing the left and right child respectively. Both branches inherit the positions of all pebbles on the input, and do not communicate; each moves the k pebbles independently of the other. In a *nullary* output the node being output is a leaf and that branch of computation halts.

Looking at the global picture, the machine starts with a single computation branch and no output nodes. After a while it has constructed some top fragment of the output tree, and several computation branches continue to compute the remaining output subtrees. The entire computation terminates when all computation branches terminate. It is shown [34, 35] that k-pebble transducers captures the core tree restructuring capabilities of the main XML query languages, including XQuery.

The acceptor analog of the k-pebble transducer is called a *k-pebble automaton*, and is defined in the natural way, as a transducer without output. The languages of ordered binary trees accepted by k-pebble transducers are precisely the regular tree languages [34, 35].

Input tree types can be defined as regular tree languages, represented by k-pebble automata. For such an automaton τ, $rep(\tau)$ is the tree language accepted by τ. The k-pebble automata provide a concise representation system that can be maintained efficiently and stays polynomial in the input type and the entire sequence of query-answer pairs. Indeed, the following is a consequence of known results [34, 35]:

Theorem 4.2 *Let τ be an input type specified by a k-pebble automaton, and $\langle q_1, A_1\rangle, \ldots, \langle q_n, A_n\rangle$ a sequence of query-answer pairs where each q_i is a k-pebble transducer and $A_i \in q_i(T)$, $i \in [1, n]$. There exists a k-pebble automaton τ', computable in* PTIME *from τ and the query-answer sequence, such that*

$$rep(\tau') \;=\; rep(\tau) \;\cap\; q_1^{-1}(A_1) \;\cap\; \cdots \;\cap q_n^{-1}(A_n).$$

Despite the fact that k-pebble automata provide an efficient representation system for a very broad class of restructuring queries, they have several drawbacks. First, the intuitively appealing representation of incomplete information provided by incomplete trees is lost. Second, k-pebble automata are not a *strong* representation system. Indeed, as discussed in [34, 35], $q(rep(\tau))$ is not necessarily a regular tree language for an input type τ and k-pebble transducer q. (This problem already occurred in the simpler setting of branching ps-queries with very simple constructed answers, see above.) Finally, the basic manipulations needed to handle incomplete information have very high complexity, as indicated by the following lower bound result:

Theorem 4.3 *It is non-elementary to determine, for a k-pebble automaton τ, whether $rep(\tau) = \emptyset$.*

The proof, due to Thomas Schwentick [39], uses the fact that testing emptiness of star-free generalized regular expressions[1] is non-elementary [40]. Recall that emptiness of conditional tree types can be tested in PTIME by Lemma 2.5, and this test is a basic step in many of our manipulations.

Remark 4.4 *The k-pebble transducer as initially defined by Milo et al. [34, 35] ignores data values associated with nodes. It is easy to extend the results above to take into account selection conditions on data values, using an extended version of the k-pebble transducer. Intuitively, the extended version augments the basic transducer with the ability to test satisfaction of a condition by a data value. Since there are finitely many equivalence classes of data values with respect to each finite set of conditions, these can be labeled by new alphabet symbols. A classical k-pebble transducer using the extended alphabet can now be used to simulate tests on data values. Theorem 4.2 contiues to hold for the extended k-pebble transducers.*

The next three extensions we consider involve joins on data values. This turns out to be an extremely powerful feature that leads to a dramatic increase in the difficulty of handling incomplete information. Indeed, many of the key questions now become undecidable.

Branching, join on data values, and negation. Negation consists of labeling some subtrees by "$\neg$". With this semantics, a valuation must match positive subtrees and there must be no extension of the valuation matching the negative subtrees. Join on data values allows comparing the data values of different nodes in the pattern of the query (using $=, \neq$). This combination of features leads to undecidability of several questions. For example, given an input tree type and a sequence of query-answer pairs (with branching, data value joins, and negation) it is undecidable whether a new query always has empty answer.

Theorem 4.5 *It is undecidable, given a (nonrecursive) input tree type τ, a sequence*

$$\langle q_1, A_1\rangle, \cdots, \langle q_n, A_n\rangle$$

of query-answer pairs, and a query q, where the q_i and q are ps-queries extended with branching, data value (in)equality, and negation, whether $q(T) = \emptyset$ for all

$$T \in rep(\tau) \cap q_1^{-1}(A_1) \cap \cdots \cap q_n^{-1}(A_n).$$

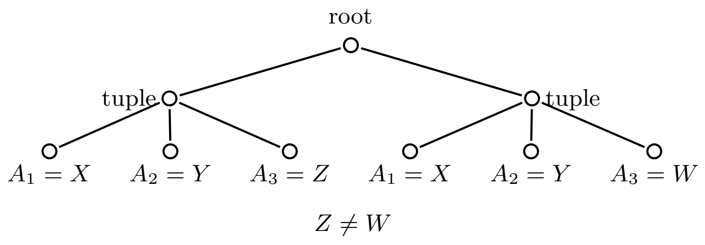
Proof: We use the undecidability of implication for functional and inclusion dependencies. It is known that for some relation R with attributes $A_1, \ldots, A_n$, it is undecidable whether $\Sigma \models \sigma$ where Σ is a finite set of fd's and incd's over R and σ is an fd over R (see [3]). We construct a tree type τ (of fixed depth) that represents relation R:

$$\begin{array}{lcl} root & \rightarrow & tuple^{\star} \\ tuple & \rightarrow & A_1 \ldots A_n \end{array}$$

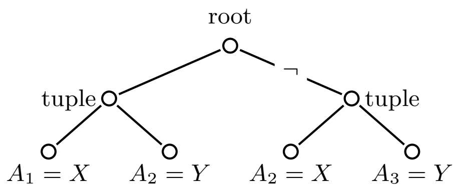
[1]These are expressions using alphabet symbols, union, concatenation, and complement.

For each $\varphi \in \Sigma \cup \{\sigma\}$, we construct a ps-query q_φ with branching, data value (in)equality, and negation, such that for each $T \in rep(\tau)$, $q_\varphi(T) = \emptyset$ if and only if the relation represented by T satisfies φ. Then $\Sigma \models \sigma$ if and only if $q_\sigma(T) = \emptyset$ for all $T \in rep(\tau) \cap \bigcap_{\varphi\in\Sigma} q_\varphi^{-1}(\emptyset)$. This proves Theorem 4.5.

We show by example how to construct the queries q_φ. Suppose φ is a fd, say $A_1A_2 \rightarrow A_3$. Then q_φ is the query:



Now suppose φ is an inclusion dependency, say $R[A_1A_2] \subseteq R[A_2A_3]$. Then q_φ is the query:



□

As an easy variation, it can be shown that it is undecidable whether a query is always non-empty for trees satisfying the input tree type and compatible with the query-answer pairs. It is also undecidable whether a given tree is a possible (certain) prefix for such trees.

Note that these results are independent of any representation system. In fact, they imply that there cannot exist an effective representation system for such queries, for which possible emptiness (or the *possible prefix* question) is decidable.

Branching, join on data values, optional subtrees, and construction. By a reduction similar to the proof of Theorem 4.5, it is possible to show the following:

Theorem 4.6 *It is undecidable, given a data tree T, a non-recursive input tree type τ, a sequence*

$$\langle q_1, A_1\rangle, \ldots, \langle q_n, A_n\rangle$$

of query-answer pairs and a query q, where the q_i and q are ps-queries extended with branching, data value (in)equality, optional subtrees, and constructed answers, whether T is a possible prefix for answers to query q on trees in $rep(\tau) \cap q_1^{-1}(A_1) \cap \ldots \cap q_n^{-1}(A_n)$.

As an aside, Theorems 4.5 and 4.6 highlight an interesting trade-off between negation, and optional subtrees together with constructed answers.

Recursive path expressions and join on data values. This extension allows specifying in a query pattern that a node is reachable from its parent in the pattern tree by a path whose labels spell a word in some regular language. Extending ps-queries with recursive path expressions and tests of (in)equality on data values leads once again to undecidability of various key questions. Indeed, we can show:

Theorem 4.7 *It is undecidable, given an input tree type τ, a sequence*

$$\langle q_1, A_1\rangle, \ldots, \langle q_n, A_n\rangle$$

of query-answer pairs and a query q, where q_i and q are ps-queries extended with recursive path expressions and (in)equality tests on data values, whether $q(T) = \emptyset$ for some

$$T \in rep(\tau) \cap q_1^{-1}(A_1) \cap \ldots \cap q_n^{-1}(A_n).$$

Proof: We use a variant of the problem of checking emptiness of the intersection of two languages defined by context-free grammars (CFGs). Recall that in the classical version, an instance of the problem consists of two CFG's G_1 and G_2 over terminal alphabet $\{a, b\}$, and the question is whether $L(G_1) \cap L(G_2) \neq \emptyset$. This is known to be undecidable [25]. In the variant we use, the input consists again of two CFG's G_1, G_2, for which it is additionally known that

(†) there exist $w_1 \in L(G_1)$ and $w_2 \in L(G_2)$ such that $|w_1| = |w_2|$.

The question is, again, whether $L(G_1) \cap L(G_2) \neq \emptyset$. We call this the *weak* CFG *intersection emptiness problem.* Note that (†) is decidable, by reduction to testing emptiness of the intersection of the *regular* languages $f(L(G_1))$ and $f(L(G_2))$, where f is the homomorphism $f(a) = f(b) = a$. It easily follows that the weak CFG intersection emptiness problem remains undecidable.

Let G_1, G_2 be CFG's with start symbols S_1 and S_2. Without loss of generality we can assume that their non-terminals are disjoint and that each is in Chomsky Normal Form (i.e., productions are of the form $A \to BC$ or $A \to a$ where A, B, C are non-terminals and a is a terminal symbol). In addition, we assume that there are no productions $A \to BC$ and $A \to DB$ (B occurs first in one production of A, and second in another). This situation can be avoided by using different versions of B for the case when B occurs first and when B occurs second. Iterating this step for all non-terminals may result in a quadratic blowup in the number of non-terminals and productions. The purpose of this requirement is to have the names of the children of a non-terminal uniquely determine their order. In particular, this implies that for any non-terminal A there exists a regular expression $r(A)$ such that in every derivation tree of the grammar starting from A, a path matches $r(A)$ if and only if it leads to the rightmost terminal derived from A. A similar regular expression $l(A)$ exists for the leftmost case. We use the notation $r_i(A), l_i(A)$ for these regular expressions for G_i, $i \in [1, 2]$.

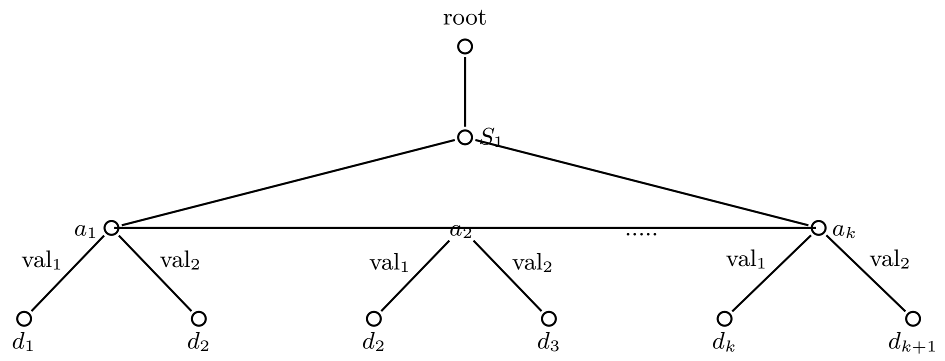
We construct an input tree type τ and queries $q_1, \ldots, q_n$ and q with recursive path expressions and data value (in)equality tests, such that every tree T in $rep(\tau)$ for which $q_i(T) = \emptyset$, $1 \leq i \leq n$, encodes two words $w_1 \in L(G_1)$ and $w_2 \in L(G_2)$ of equal length. In particular, it follows that G_1 and G_2 satisfy (†) and are an instance of the weak intersection emptiness problem. Finally, $q(T) = \emptyset$ if and only if $w_1 = w_2$. It follows that q is possibly empty on trees in $rep(\tau) \cap q_1^{-1}(\emptyset) \cap \ldots \cap q_n^{-1}(\emptyset)$ if and only if $L(G_1) \cap L(G_2) \neq \emptyset$, which is undecidable. This proves the theorem.

We outline the construction of the tree type and queries q_i. The tree type τ consists of:

- $root \to S_1\ S_2$
- the productions of G_1 and G_2
- $a \to val_1\ val_2$

- $b \ \rightarrow \ val_1 \ val_2$.

Note that trees in $rep(\tau)$ consist of a derivation tree of G_1 and a derivation tree of G_2. The terminals a, b additionally have children val_1, val_2 attached to them. Their role is to provide a successor relation on data values, inducing an ordering of the terminal symbols to which they are attached. Thus, a word $a_1 \dots a_k$ derived from S_1 is represented as:



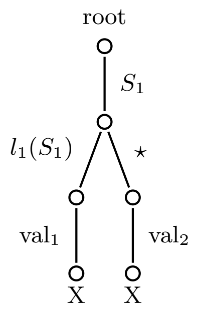
The queries $q_1, \dots, q_n$ ensure that:

1. the data values for the leaves of the subtree rooted at S_1 form a successor relation, and similarly for S_2; and
2. the two sequences of data values at the leaves of the subtrees rooted at S_1 and S_2 are the same.

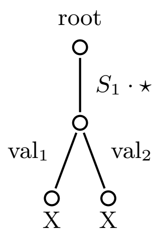
In the graphical representation of queries, we place the labels on edges of the pattern rather than nodes. The labels are regular path expressions, $\star$ being a shortcut for $\Sigma^\star$ (Σ is the set of symbols of the grammars). Variables stand for data values.

The queries whose empty answers ensure (1) above for the subtree rooted at S_1 are as follows:

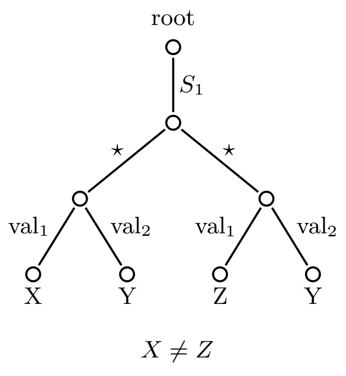
- The leftmost data value is minimal, i.e., it never occurs as the value of val_2:



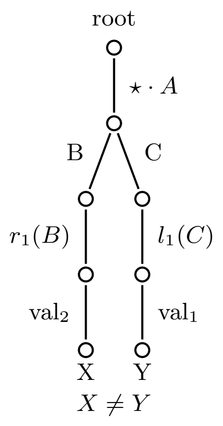
- Sibling nodes val_1 and val_2 have different data values (an element's successor is not itself):



- Distinct elements have distinct successors:

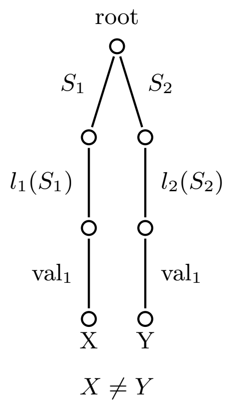


- For adjacent nodes, the val_2 value of the first equals the val_1 value of the second. For each production $A \rightarrow BC$ of G_1, we use the query:

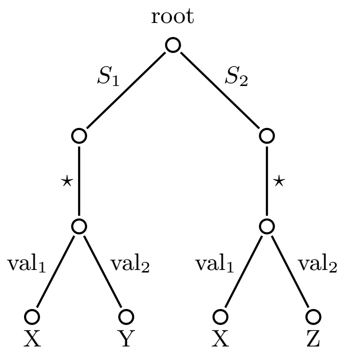


Similar queries ensure (1) for S_2. Item (2) is ensured by emptiness of the following queries:

- The leftmost data values for S_1 and S_2 are the same:

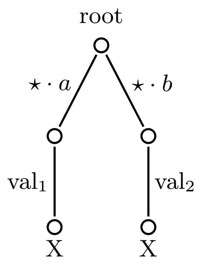


- The rightmost data values in S_1 and S_2 are the same: similar.
- If two nodes in S_1 and S_2 have the same val_1 value, then they have the same val_2 value:



$Y \neq Z$

Thus, emptiness of the queries above ensures that the words w_1 and w_2 corresponding to S_1 and S_2 are indexed by the same data values in the same order (in particular $|w_1| = |w_2|$). Finally, consider the query q:



Its answer is empty if and only if w_1 and w_2 have the same terminal symbol corresponding to each data value index, i.e., $w_1 = w_2$.

□

In particular, the above theorem shows that there can be no effective strong representation system for this class of queries.

Node ids and order. To conclude, we informally discuss the persistent node id assumption, and the issue of order.

Node ids. A significant assumption in our framework is the availability of persistent node ids. In other words, distinct queries against an XML document return nodes with the same id if and only if the nodes are identical. The availability of node ids allows us to enrich the information about a given node (e.g., a product) through consecutive queries, as illustrated in the catalog example. Without ids this is no longer possible in general. Furthermore, the representation system for incomplete information would have to be extended in order to keep track of the various possible ways of matching nodes returned by different queries.

Our assumption that node ids are available is generally dependent on sources providing persistent node ids. Even if this is not generally the case, ids are sometimes available as URLs, element names, known keys, etc. Without ids, our approach can still be used but our expectations would have to be lowered if we wish to keep processing cost down.

Order. The issue of order has many facets. Indeed, it can be considered at various levels:
(1) The input tree may be ordered, as well as the answers to queries. In this case, one would like to preserve in the answer the order of elements from the input.
(2) The source DTD may describe the order of children at each node type, possibly using a

regular expression (as done in full-fledged DTDs).
(3) Queries may use ordering in their selection patterns. For example, a query might request all a elements that occur before some b element. To specify such conditions one could use regular expressions, or perhaps weaker partial order conditions.

Our discussion of (extended) k-pebble transducers shows that some of our framework can be extended in the presence of order, albeit at the cost of high complexity. Intuitively, it is clear that order complicates the handling of incomplete information. As one example, suppose the input is flat and contains a and b elements. Suppose a first query q_1 requests the list of a elements (produced in the order in which they appear in the input) and a second query q_2 asks for the list of b elements. Consider now a third query q_3 that asks for the list of all elements. Can we answer this using the known answers to q_1 and q_2? This depends on the type of the input: If the input is of the form $a^\star b^\star$, then q_3 can be answered (concatenate the answer to q_1 with the answer to q_2); if the input is of the form $(a+b)^\star$, then q_3 cannot be answered using the previous queries, since no information is available on how to interleave the a and b elements. The problem described above is somewhat similar to the issue of persistent ids. Indeed, one way around it is for wrappers of data sources to provide the rank of each element, which allows merging answers to consecutive queries. In the absence of such information, a representation system has to maintain information about partial orders among the elements. Clearly, the order issue raises many interesting questions that need to be further explored.

5 Conclusion

The main contribution of this paper is a simple framework for acquiring, maintaining, and querying XML documents with incomplete information. The framework provides a model for XML documents and DTDs, a simple XML query language, and a representation system for XML with incomplete information. We show that the incomplete information acquired by consecutive queries and answers can be efficiently represented and incrementally refined using our representation system. Queries are handled efficiently and flexibly. They are answered as best possible using the available information, either completely, or by providing an incomplete answer using our representation system. Alternatively, full answers can be provided by completing the partial information using additional queries to the sources, guaranteed to be non-redundant.

Our framework is limited in many ways. For example, we assume that sources provide persistent node ids. Order in documents and DTDs is ignored, and is not used by queries. The query language is very simple, and does not use recursive path expressions and data joins. In order to trace the boundary of tractability, we considered several extensions to our framework and showed that they have significant impact on handling incomplete information, ranging from cosmetic to high complexity or undecidability. This justifies the particular cocktail of features making up our framework, and suggests that it provides a practically appealing solution to handling incomplete information in XML.

Several interesting questions remain to be explored. In our framework, we assume that sources are static. To use our approach when sources are dynamic, we can reinitialize the information about each source to its known DTD, whenever the source changes. A more subtle alternative is to use available information (if any) on the source modification in order to salvage some of the previously accumulated information.

Another interesting issue is how to couple our simple ps-queries used against sources with queries in a more powerful language, asked locally at the Webhouse. This entails being able to decide if a local query can be answered using the information provided by an incomplete tree, and if not, to generate additional ps-queries against the sources needed to answer the query.

References

[1] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for Active XML. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 227–238, 2004.

[2] S. Abiteboul and O. Duschka. Answering queries using materialized views. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 1998.

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

[4] S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. *Theoretical Computer Science*, 78:159–187, 1991.

[5] S. Abiteboul, L. Segoufin, and V. Vianu. Representing and querying XML with incomplete information. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 2001.

[6] A.Bruggemann-Klein, M.Murata, and D.Wood. Regular tree languages over non-ranked alphabets. Unpublished manuscript, 1998.

[7] S. Amer-Yahia, S. Cho, V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2001.

[8] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2004.

[9] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *Proc. Int. Conf. on Database Theory (ICDT)*, 1999.

[10] S. Bose and L. Fegaras. XFrag: A query processing framework for fragmented XML data. In *WebDB Workshop*, 2005.

[11] D. Calvanese, G. De Giacomo, and M. Lenzerini. Semi-structured data with constraints and incomplete information. In *Proceedings of the 1998 Description Logic Workshop (DL'98)*, pages 11–20, 1998.

[12] D. Calvanese, G. D. Giacomo, M. Lenzerini, , and M. Vardi. Rewriting of regular expressions and regular path queries. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 194–204, 1999.

[13] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Vardi. Answering regular path queries using views. In *Int'l. Conf. on Data Engineering*, pages 389–398, 2000.

[14] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Vardi. View-based query processing for regular path queries with inverse. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 58–66, 2000.

[15] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Vardi. Lossless regular views. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 247–258, 2002.

[16] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proc. IEEE Intl. Conf. on Data Engineering*, 1995.

[17] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediator needs data conversion. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1998.

[18] T. Codd. Understanding relations (installment #7). In *FDT Bull. of ACM Sigmod 7*, pages 23–28, 1975.

[19] S. S. Cosmadakis. The complexity of evaluating relational queries. *Inf. and Control*, 58:101–112, 1983.

[20] S. Flesca, F. Furfaro1, S. Greco, and E. Zumpano. Repairs and consistent answers for xml data with functional dependencies. In *International XML Database Symposium*, pages 238–253, 2003.

[21] H. Garcia-Molina, W. Labio, and J. Yang. Expiring data in a warehouse. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1998.

[22] G. Grahne. *The Problem of Incomplete Information in Relational Databases*. Springer-Verlag, Berlin Heidelberg, 1991.

[23] A. Halevy. Theory of answering queries using views. *SIGMOD Record*, 29(4):40–47, 2000.

[24] P. Honeyman, R. Ladner, and M. Yannakakis. Testing the universal instance assumption. *Inf. Proc. Letters*, 10(1):14–19, 1980.

[25] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[26] T. Imieliński and J. Lipski, Witold. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[27] Y. Kanza, W. Nutt, and Y. Sagiv. Queries with incomplete answers over semistructured data. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 1999.

[28] W. Labio, Y. Zhuge, J. L. Wiener, H. Gupta, H. Garcia-Molina, and J. Widom. The WHIPS prototype for data warehouse creation and maintenance. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1997.

[29] A. Levy, A. Mendelzon, D. Srivastava, and Y. Sagiv. Answering queries using views. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 1995.

[30] D. Maier, Y. Sagiv, and M. Yannakakis. On the complexity of testing implications of functional and join dependencies. *J. ACM*, 28(4):680–695, 1981.

[31] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2005.

[32] G. Miklau and D. Suciu. Containment and equivalence for a fragment of xpath. *J. ACM*, 51(1):2–45, 2004.

[33] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. Ngoc. Exchanging intensional xml data. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 289–300, 2003.

[34] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, 2000.

[35] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *J. Comput. Syst. Sci.*, 1(66):66–97, 2003.

[36] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 2000.

[37] A. Rajaraman, Y. Sagiv, and J. Ullman. Answering queries using templates with binding patterns. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 1995.

[38] R. Reiter. A sound and sometimes complete query evaluation algorithm for relational databases with null values. *J. ACM*, 33(2):349–370, 1986.

[39] T. Schwentick, 2000. Personal communication.

[40] L. Stockmeier. *The complexity of decision problems in automata theory and logic.* PhD thesis, Massachusetts Institute of Technology, 1974. Report MAC TR-133, Project MAC.

[41] M. Y. Vardi. The complexity of relational query languages. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 137–146, 1982.

[42] M. Y. Vardi. On the integrity of databases with incomplete information. In *Proc. ACM Symp. on Principles of Database Systems*, pages 252–266, 1986.

[43] C. Zaniolo. Database relations with null values. *Journal of Computer and System Sciences*, 28(1):142–166, 1984.