Types for Detecting XML Query-Update Independence

Federico Ulliana

Université Paris Sud

Laboratoire de Recherche en Informatique

THÈSE DE DOCTORAT

présentée par : **Federico Ulliana**

Types for Detecting XML Query-Update Independence

THÈSE DIRIGÉE PAR

Bidoit-Tollu Nicole	*Professeur, Université Paris Sud*
Colazzo Dario	*Maître de Conférence, HDR, Université Paris Sud*

RAPPORTEURS

Senellart Pierre	*Maître de Conférence, HDR, Telecom ParisTech*
Schmitt Alan	*CR, HDR, INRIA Rennes*

EXAMINATEURS

Manoussakis Yannis	*Professeur, Université Paris Sud*
Talbot Jean-Marc	*Professeur, Université de Provence*

Abstract

In the last decade XML became one of the main standards for data storage and exchange on the Web. Detecting XML query-update independence is crucial to efficiently perform data management tasks, like those concerning view-maintenance, concurrency control, and security. This thesis presents a novel static analysis technique to detect XML query-update independence, in the presence of a schema. Rather than types, the presented system infers *chains* of types. Each chain represents a path that can be traversed on a valid document during query/update evaluation. The resulting independence analysis is precise, although it raises a challenging issue: recursive schemas may lead to infer infinitely many chains. This thesis presents a sound and complete approximation technique ensuring a finite analysis in any case, together with an efficient implementation performing the chain-based analysis in polynomial space and time.

Résumé

Pendant la dernière décennie, le format de données XML est devenu l'un des principaux moyens de représentation et d'échange de données sur le Web. La détection de l'indépendance entre une requête et une mise à jour, qui a lieu en absence d'impact d'une mise à jour sur une requête, est un problème crucial pour la gestion efficace de tâches comme la maintenance des vues, le contrôle de concurrence et de sécurité. Cette thèse présente une nouvelle technique d'analyse statique pour détecter l'indépendance entre requête et mise à jour XML, dans le cas où les données sont typées par un schéma. La contribution de la thèse repose sur une notion de type plus riche que celle employée jusqu'ici dans la littérature. Au lieu de caractériser les éléments d'un document XML utiles ou touchés par une requête ou mise à jour en utilisant un ensemble d'étiquettes, ceux-ci sont caractérisés par un ensemble de chaînes d'étiquettes, correspondants aux chemins parcourus pendant l'évaluation de l'expression dans un document valide pour le schéma. L'analyse d'indépendance résulte du développement d'un système d'inférence de type pour les chaînes. Cette analyse précise soulève une question importante et difficile liés aux schémas récursifs: un ensemble infini de chaînes pouvant être inférées dans ce cas, est-il possible et comment se ramener à une analyse effective donc finie. Cette thèse présente donc une technique d'approximation correcte et complète assurant une analyse finie. L'analyse de cette technique a conduit à développer des algorithmes pour une implantation efficace de l'analyse, et de mener une large série de tests validant à la fois la qualité de l'approche et son efficacité.

Contents

Chapter 1

Introduction

In the last decade, XML has established itself as one of the main standards for data storage and exchange on the Web. Along with XML, a family of languages for querying and manipulating XML data have been devised, such as W3C standards XPath, XQuery and XSLT languages [SCF+07]. The recent W3C standardization of an update facility for XML [RCD+11] attracted a big deal of attention by the database community. Many old questions related to update optimization, already recognized as of crucial importance for relational databases, are open again for XML. Three central issues related to XML updates are view-maintenance, concurrency, and security.

View-maintenance Materialized views are precomputed queries from the database, that are used to expedite query answering. Materialized views have been shown to improve query evaluation performance by up to several orders of magnitude. View-maintenance is the problem of propagating into a materialized view the changes in the base data made by updates. View maintenance is a non-trivial process. Incremental re-computations of the view can take time proportional to the database size just to determine that no operation has to be done on the view and, in general, the problem is non-trivial even for XPath based queries and updates [BGMM09].

Concurrency It is well known that, for ensuring high performances in the execution of concurrent query and update expressions, it is fundamental to evaluate them *in parallel*. Parallel evaluations, with a proper level of safety, can be ensured once one is able to define concurrency control mechanisms, locking schemes and schedulers which guarantee serializability of transactions on XML documents. All of this could be achieved, for instance, by means of a *commutativity* analysis, able to check if two expressions can be safely executed on a database independently of their order. Such an analysis would permit also to optimize logical plans for query languages with side effects, as shown in [GORS08].

Security Security views are a well studied framework to provide controlled access to data XML [FCG04]. The idea is that each user-group is provided with an XML view

consisting of all and only the information that the users of that group are authorized to access or to update. It is of crucial importance to verify that updates coming from a user-group are executed respecting access control policies.

A way to entail an advance of technology in all such contexts, is to provide an efficient solution to the problem of detecting *XML query-update independence*. Query-update independence holds when a query and an update over a database do not interact, and therefore the update does not alter query result.

How can independence detection help tasks of view-maintenance, concurrency and security control? Intuitively, when a view is specified as a query, being able to determine that there is no interaction between a query and and update, allows to skip the whole view maintenance phase. Concerning concurrency control, if a query and an update do not interact, then they commute, meaning that they can be safely executed in any order, or even in parallel. Concerning security views, when these specify the part of data that cannot be accessed or modified, independence can be used to ensure that updates can be executed without violating the security policy expressed by the views.

A query and an update are independent when the query result is not affected by update execution, on any possible input database. In all contexts where detecting query-update independence is of crucial importance, benefits are amplified when query-update independence can be checked *statically*. In order to be useful, every static analysis technique must be sound: if query-update independence is statically detected, then independence does hold. The inverse implication (completeness) cannot be ensured in the general case, since static independence detection is undecidable (see [BC09a]). This means that if a static analyzer is used, for instance, in a view maintenance system, sometimes views are re-materialized after updates even if not needed, because the analysis has not been smart enough to statically detect a view-update independence. Useless view re-materialization frequently occurs if a static analyzer with low precision is adopted. This can lead to great waste of time, since view materialization cost can be proportional to the database size.

High precision of static independence analysis can be ensured by taking into account schema information. In many contexts, schemas are defined by users, mainly by means of the DTD or XML Schema languages, while in other contexts quite precise schemas, in the form of a DTD, can be automatically inferred, by using accurate and efficient existing techniques like the one proposed by Bex et al. in [BNSV10]. Schema-based detection of XML query-update independence has been recently investigated. The state of the art technique has been presented by Benedikt and Cheney in [BC09a]. This technique infers from the schema the set of node types traversed by the query, and the set of node types impacted by the update. The query and the update are then deemed as independent if the two sets do not overlap. This technique is effective since the static analysis *i*) is able to manage a wide class of XQuery queries and updates, *ii*) can be performed in a negligible time, and *iii*) as a consequence, even on small documents, can avoid expensive query re-computation when independence wrt an update is detected. However, the technique has

some weaknesses. As illustrated in [BC09a], in some cases, independence is not detected by the static analysis, due to some over-approximation made by the type inference rules.

Contributions

In this dissertation we propose a novel schema-based approach for detecting XML query-update independence. Differently from [BC09a, Che08, CGMS06], our system infers sequences of labels, hereafter called *chains*, to perform a static analysis of the data accessed by the queries and the updates. Intuitively, for each node that can be selected by a query/update path in a schema instance, the system infers a chain recording *i*) all labels that are encountered from the root to the node and *ii*) the order of traversal. This information is at the basis of a precise static independence analysis.

The main contribution of this work is a precise algorithm to detect independence for a query-update pair `q-u` knowing that documents are valid wrt a DTD `d`. It strongly relies on the following developments.

- Chain-based independence for queries and updates, a static notion, is the foundation of our algorithm: starting from the set of all possible chains associated with a DTD, our inference system extracts subsets of chains for the query and the update, capturing the navigation through valid documents made by the evaluation of the query and the update, respectively. Our inference system is cautiously specified for dealing with all XPath axes. Chain-based independence is the result of the absence of overlapping pair of chains for the query and the update. Our inference system is formally proved to be *sound*.

- A major step of our work concerns recursive schemas, for which chain-based independence analysis may cripplingly involve to deal with an infinite number of chains. Our technique enabling the restriction of the analysis to *finite* subsets of query and update chain is a key contribution, and the core of our algorithm is the resulting finite analysis. It is proved to be *equivalent* to the infinite analysis

- Aiming at designing a tractable analysis, we show that by using a DAG-based representation of inferred chains, the finite analysis can run in polynomial space and time. Our technique has been carefully implemented, and extensive tests have been performed to validate our claim of precision and efficiency. Concerning precision, our results show that our technique outperforms [BC09a] to a large extent.

This work has been published in the *International Conference of Very Large Databases 2012* [BTCU12], while preliminary versions have been presented in the *26ème Journée des Bases de Données Avancées 2010* [BTCU10a] and in the *International Formal Methods Workshop 2010* [BTCU10b].

The thesis is organized as follows. Chapter 2 introduces basic definitions about XML, schemas, queries and updates. Chapter 3 revises the state-of-the-art for detecting XML

query-update independence, and provides motivations for the work here presented. Chapter 4 presents our static independence analysis based on schemas and chains. Chapter 5 describes how the chain analysis can be efficiently implemented. Chapter 6 presents experiments validating our method. Chapter 7 discusses extensions, while Chapter 8 draws conclusions and discusses future perspectives.

Chapter 2

Preliminaries

In this chapter we define the formal framework on top of which we build our work. We begin by presenting XML and DTDs, as the data model and the type language of semi-structured databases. Then, we present the XML query and update languages that we consider, together with their formal semantics, which is used to prove the correctness of our analysis. We conclude by defining the XML query-update independence problem.

2.1 XML

XML [BPSM+06] is a W3C-recommended language for representing data on the Web. XML has been designed to facilitate information exchange across different systems, thus overcoming the limits of its predecessor HTML, that was purposely conceived for displaying. XML is capable of wrapping many different kinds of data, because the data is represented in a textual form, organized hierarchically in a tree, and the document structure is self-describing by means of markup tags.

An XML Document is written as a unicode text with markup tags and other meta-information representing all document nodes. The most important ones we consider are element and textual nodes. As an example, consider the XML document t_1 in Figure 2.1. It is a fragment of a bibliography, containing a single book, with its own title and author. Textual nodes here are written as the text they represent, for instance "`B. Pierce`". Element nodes are denoted by pairs of open and closing markup-tags, for instance `<book></book>`, and are possibly nested inside other elements.

An XML document in its textual form must be *well-formed*. This essentially means that it defines a tree structure, the labels of open and closing tags match, and are also properly nested, so that the closing tag of a node appears always before the close-tag of its parent. The document t_1 is well-formed.

The abstract model underlying XML documents is tree-based. In its textual form, an XML document looks like the linearization of a tree. In this work, we represent an

```
<bib>
  <book>
    <title>
      Types and Programming Languages
    </title>
    <author>
      <name>B. Pierce</name>
    </author>
  </book>
</bib>
```

$$\sigma = \begin{cases} l_{t_1} \leftarrow \texttt{bib}[\, l_1 \,] \\ l_1 \leftarrow \texttt{book}[\, l_2, l_3 \,] \\ l_2 \leftarrow \texttt{title}[\, l_2' \,] \\ l_2' \leftarrow \text{“}\texttt{Types and ...}\text{”} \\ l_3 \leftarrow \texttt{author}[\, l_3' \,] \\ l_3' \leftarrow \texttt{name}[\, l_3'' \,] \\ l_3'' \leftarrow \text{“}\texttt{B. Pierce}\text{”} \end{cases}$$

Figure 2.1: Document t_1 and store (σ, l_{t_1})

XML document as a *store* σ, which is an environment associating each *node location* l with either an element node $\texttt{a}[L]$ or a text node “$\texttt{txt}$”. In $\texttt{a}[L]$, $\texttt{a}$ is the element *tag* (or label), while $L = (l_1, \ldots, l_n)$ is the ordered sequence of *children* locations in σ. We denote by $dom(\sigma)$ the set of locations of σ. A tree is a pair

$$t = (\sigma, l_t)$$

where l_t is the root location. With a little abuse of notation, we denote by $dom(t)$ the domain of t, which consists of l_t together with all locations connected to l_t. We denote by $\sigma@l$ the *tree* t in σ rooted at l. We denote by $t \sim t'$ two isomorphic trees t and t'; they may differ only in terms of names of node locations. The empty sequence of locations is denoted by $()$. The concatenation of two sequences of locations L and L' is denoted by $L \cdot L'$, and defined such that $L \cdot () = () \cdot L = L$.

In Figure 2.1 we draw the store associated to the XML document t_1; this store is a driving example for most of the definitions provided in the chapter. As already mentioned, each element tag of t_1 corresponds to a single node in the store, and vice versa. As a subtlety, notice that each node is referenced by a location. A location can be simply seen as the identifier of the node.

The XML data model has many features, but in this work we consider only element and textual nodes, since they are the most relevant ones. Most of the missing features, e.g., string attribute nodes, can be straightforwardly encoded in our model. We consider tree instances without document-node [BPSM+06] (an unlabeled node which is the parent of the root). Other mechanisms such as key referencing (ID/IDREF) require to extend our model, by adding further constraints. We discuss extensions in Chapter 7.

2.2 Schema Languages

The concept of *schema* is standard in databases: it is a set of constraints defining a collection of data. XML documents are self-describing, hence a schema is an optional feature of XML databases. Nevertheless, any data collection is likely to follow a structure, and schemas are a very powerful means to express structural constraints. In all the context where a schema is available, it can be exploited to ensure *safe* and *efficient* XML processing [CGS11].

Many formalisms for writing XML schemas have been proposed, with the notable distinction of Document Type Defition (DTD) and XML Schema (XSD) as W3C standards, and RelaxNG as OASIS standard [CM01]. In this work we will focus on the widely used DTDs. DTDs embody all of the main features of XML schema languages, whilst still allowing to keep the formalization concise. The extension of DTDs to more powerful languages such as XSD, by means of *extended DTDs* [PV00] will be also discussed.

A DTD essentially consists of a list of declarations of elements. As an example, consider the schema `bibliography.dtd` reported in Figure 2.2. The declaration of elements follows a specific syntax like <!`ELEMENT bib (book*)` > stating that an element tag is defined, that the name of the element is `bib`, and that the content of the element is modeled by the regular expression `book*`. The reserved word `#PCDATA` denotes the string type.

DTDs can be formally defined along the line of [MLMK05].

Definition 2.2.1 (DTD)**.** *A DTD is a 3-tuple* $(\mathsf{d}, \mathsf{s}, \Sigma)$ *where*

- Σ *is a finite alphabet for element tags, denoted by* $\mathsf{a}, \mathsf{a}_2, \mathsf{a}'$*;*
- $\mathsf{s} \in \Sigma$ *is a non-recursive root-type;*
- d *is a function from* Σ *to regular expressions over* $\Sigma \cup \{\mathsf{String}\}$*, where* String *denotes the string base-type.*

For convenience, in this work we assume that the root-type of a DTD is non-recursive. This implies that the root-element tag is never re-used inside a valid document, as happens in most practical scenarios. As we will discuss in Chapter 4, Section 4.4.2, this assumption is made for the sake of formalizations of the finite analysis, and is not restrictive. From now on we will use only the d component to specify a DTD, and we will write Σ_{S} to denote the set $\Sigma \cup \{\mathsf{String}\}$.

In Figure 2.2 we show how a DTD is encoded in our formalism. The verbose declarations are denoted in our formalism in a simpler way, by associating each element tag with a regular expression modeling the element tag content.

An XML database compliant with all of the structural constraints defined by a schema is said to be *valid* against the schema.

<ELEMENT bib (book*)>	bib	←	book*
<ELEMENT book (title, author+)	book	←	title, author+
<ELEMENT title (#PCDATA)>	title	←	String
<ELEMENT author (name)>	author	←	name
<ELEMENT name (#PCDATA)>	name	←	String

Figure 2.2: DTD `bibliography.dtd` and the schema model d_1

Definition 2.2.2 (Validity)**.** *A tree $t = (\sigma, l_t)$ is valid wrt a DTD $(\mathsf{d}, \mathsf{s}, \Sigma)$, denoted by $t \in \mathsf{d}$, if and only if there exists a mapping $\lambda : dom(t) \mapsto \Sigma_\mathsf{S}$ such that*

- $\lambda(l_t) = \mathsf{s}_\mathsf{d}$
- $\lambda(l) = \mathsf{String}$ *implies that $\sigma(l)$ is a text node*
- $\lambda(l) = \mathsf{a}$ *implies that $\sigma(l)$ is an element-node $\mathsf{a}[L]$, and that the word $\lambda(L)$ is generated by the regular expression $\mathsf{d}(\mathsf{a})$, i.e., $\lambda(L) \in \mathsf{Lang}(\mathsf{d}(\mathsf{a}))$*

where $\mathsf{Lang}(\mathsf{r})$ denotes the language generated by the regular expression r.

As an example, the document t_1 in Figure 2.1 is valid wrt the DTD d_1 in Figure 2.2, because there exists a mapping λ such that

- for the root l_{t_1}, we have $\lambda(l_{t_1}) = \mathsf{bib}$
- for all textual nodes, namely l'_2 and l''_3, we have $\lambda(l'_2) = \lambda(l''_3) = \mathsf{String} \in \mathsf{Lang}(\mathsf{String})$
- for all the sequences of children locations, namely $(l_1), (l_2, l_3)$ and (l'_3) we have

$$\begin{array}{rcll} \lambda(l_1) & = & \mathsf{book} & \in \mathsf{Lang}(\mathsf{book}*) \\ \lambda(l_2, l_3) & = & \mathsf{title\ author} & \in \mathsf{Lang}(\mathsf{title}, \mathsf{author}+) \\ \lambda(l'_3) & = & \mathsf{name} & \in \mathsf{Lang}(\mathsf{name}) \end{array}$$

A recursive DTD d may not have any XML tree t such that $t \in \mathsf{d}$. This is because the content description of some element a in d is *non-terminating*, i.e., there exists no finite subtree rooted at a that satisfies all constraints of d. A DTD d may also feature the content description of some elements that are *unreachable* from the root. No subtree of a valid tree is typed with an unreachable type definition, because there is a unique root type, and each element type definition is unique.

One can determine whether an element type a in d is non-terminating or unreachable in linear time (in the size of d). These problems can be reduced to the emptiness problem for context-free grammars, which can be solved in linear time [HU00]. To simplify the discussion, in the sequel we assume that all element types in a DTD are terminating, and that all types in Σ_S are reachable from the root.

DTDs capture all of the main features of XML schema languages. The principal restriction a DTD imposes is that each element tag definition must be *unique*. This is partially relaxed in XSDs, where a tag can have multiple type definitions, but at a specific condition: two type definitions for the same element tag, are never used in the same element content description. This is the *single type* restriction.

To illustrate, a legal fragment of XSD containing two element tag definitions that cannot be expressed by DTDs, is the following one.

```
<xs:element name = "author">
  <xs:complexType>
    <xs:element name = "name"  type = "AuthorName"/>
  </xs:complexType>
</xs:element>

<xs:element name = "publisher">
  <xs:complexType>
    <xs:element name = "name"  type = "PublisherName"/>
  </xs:complexType>
</xs:element>
```

Notice that XSDs and DTDs have different syntax (XSDs are written as XML documents). The command `xs:element` defines a new element tag, while `xs:complexType` is used here to define the type structure of the element tag. In the example, two element tags are defined, namely `author` and `publisher`. Both have a common child element, `name`, specifying the name of the author and that of the publisher.

In DTDs, the definition of `name` must be unique while, in XSDs, we can use multiple type definitions. In this case, the schema refers to distinct type definitions AuthorName and PublisherName (defined elsewhere in the schema) depending on wether `name` is nested in `author` and `publisher`, respectively. What is powerful here is that the two types can define completely different structures of the `name` element.

A fragment of XSD that does not respect the single type restriction is the following.

```
<xs:element name = "person">
  <xs:complexType>
    <xs:choice>
      <xs:element name = "name"  type = "AuthorName"/>
      <xs:element name = "name"  type = "PublisherName"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Here `xs:choice` says that only one between the elements is the list can be chosen. In particular, one between the two `name` element definitions can be used. This definition is

illegal in XML Schema, since it is illegal to have two element types with the same name (i.e., `name`) but different types (i.e., AuthorName, PublisherName) in the same regular expression. Indeed, this breaks the single type restriction, and it is not expressible in DTDs nor in XSDs. Nevertheless it is expressible in RelaxNG.

A formalism that goes beyond that of DTDs and captures the expressivity of XML Schemas is that of *Extended DTDs* (EDTDs).

Definition 2.2.3 (Extended DTD and Validation [PV00])**.** *An Extended DTD is a triple* $(\Sigma, \mathsf{d}', \mu)$ *where*

- d' *is a DTD defined over the tag alphabet* Σ'
- $\mu : \Sigma'_{\mathsf{S}} \mapsto \Sigma_{\mathsf{S}}$ *is a function, that is the identity on input* String

A tree $\mathcal{A}$ *over* Σ *is* valid *wrt the extended DTD* $(\Sigma, \mathsf{d}', \mu)$ *if and only if there exists a tree* t' *over* Σ' *such that* $\mu(t') = t$ *and* t' *is valid wrt the DTD* d'*.*

The crux of the definition is to interpret the two tag alphabets Σ and Σ' in a way such that Σ represents all element tags found in the tree instance, while Σ' contains a different tag for each type definition in the schema. In case of a tag with multiple definitions, such as `name` in the last example, we would have $\Sigma \supseteq \{\texttt{name}\}$ and $\Sigma' \supseteq \{\mathsf{AuthorName}, \mathsf{PublisherName}\}$ with $\mu(\mathsf{PublisherName}) = \texttt{name}$ and $\mu(\mathsf{AuthorName}) = \texttt{name}$. The validation of the tree t over Σ wrt the extended DTD is then defined in terms of validation of a tree t' over Σ' wrt a DTD over a "type" alphabet Σ'.

It is worth noticing that EDTDs do not make any assumption on the use of types in the regular expressions, and hence they are strictly more expressive than XML Schemas, and roughly correspond to RelaxNG (without interleaving and counting operators). In this work, we will develop a static analysis relying on DTDs, and in Chapter 7 we will show that it can be lifted to EDTDs very easily.

XML *types* are founded on regular tree grammars [HVP05]. Indeed, a schema language (e.g., DTD) roughly corresponds to a class of regular tree grammars [MLMK05]. This is needed for designing typed XML query and update language, where types of input and output expressions can be checked and type safety possibly ensured.

However, our ultimate goal is to develop an *access analysis* based on schema informations, which is a problem radically different from that of type-checking. To avoid any confusion, in this work we adopt the following definition of XML type.

Definition 2.2.4 (XML Type)**.** *Let* d *be a DTD, a type* $\tau \in \Sigma_{\mathsf{S}}$ *stands for the set of nodes* $\mathcal{A}$ *such that*

- *if* τ *is the label* a *then* $\mathcal{A}$ *is the set of all nodes labeled with* a *belonging to some tree* $t \in \mathsf{d}$
- *if* τ *is* String *then* $\mathcal{A}$ *is the set of all textual nodes belonging to some tree* $t \in \mathsf{d}$

From this definition, it follows that the *tag* of each element definition in a DTD is a type. As an example, consider the bibliographic DTD d_1. The type author stands for all nodes labeled as `author` belonging to a tree valid wrt the DTD d_1.

A DTD induces a type dependency graph, defined as follows.

Definition 2.2.5 (Dependency graph)**.** *Let* d *be a DTD, the dependency graph of* d*, denoted by* $\mathcal{G}_\mathsf{d}$*, is a directed graph defined as* $\mathcal{G}_\mathsf{d} = (\Sigma_\mathsf{S}, \Rightarrow_\mathsf{d})$.

The vertex-set of $\mathcal{G}_\mathsf{d}$ coincides with Σ_S, which is the tag-alphabet of the schema together with the String type. The edge-set of $\mathcal{G}_\mathsf{d}$ is $\Rightarrow_\mathsf{d}$, therefore two (immediately) reachable types in Σ_S are two adjacent vertex in $\mathcal{G}_\mathsf{d}$. The root of the graph is s_d. The graph is connected because all types in Σ_S are reachable from s_d. A DTD d is recursive if and only if $\mathcal{G}_\mathsf{d}$ has a cycle.

Proposition 2.2.6. *Let* $\mathcal{G}_\mathsf{d}$ *be a type dependency graph with* n *vertices and* m *edges.*

i) $n{-}1 \leq m \leq n(n{-}1)$

ii) if $m > n(n{-}1)/2$ *then* $\mathcal{G}_\mathsf{d}$ *is cyclic*

Proof. i) If $\mathcal{G}_\mathsf{d}$ has n vertices, the number of edges in $\mathcal{G}_\mathsf{d}$ varies from $n{-}1$, because the graph is connected, to $n(n{-}1)$, because all types can be recursive, except for the root that we assumed to be non-recursive. *ii)* An *acyclic* dependency graph of n vertices with the maximum number of edges, is such that only one type (the root) is connected with at most $n{-}1$ types, only one type is connected with at most with $n-2$ types, and so on until a single type with no children (e.g., String). This is $\sum_{i=1}^{n}(i-1) = n(n{-}1)/2$. □

We call *saturated* a DTD d that maximizes the number of edges in the dependency graph $\mathcal{G}_\mathsf{d}$. Saturated DTDs are defined as follows.

Definition 2.2.7. *(Saturated DTDs) A recursive DTD* d *is said to be saturated if* $\mathcal{G}_\mathsf{d}$ *has* n *vertexes and* $n(n{-}1)$ *edges. A non-recursive DTD* d *is said to be saturated if* $\mathcal{G}_\mathsf{d}$ *has* n *vertexes and* $n(n{-}1)/2$ *edges.*

An example of a saturated and non-recursive DTD and relative dependency graph is depicted in Figure 2.3. The DTD d_2 features 5 elements. The root type a_1 is defined in terms of all other types $\mathsf{a}_2, \ldots, \mathsf{a}_5$, the type a_2 is defined in terms of all other types except the root and itself, namely $\mathsf{a}_3, \ldots, \mathsf{a}_5$, and so on until the singleton type a_5 that has empty content. Note that no edge can be added in $\mathcal{G}_{\mathsf{d}_2}$ without introducing recursion.

$$
\begin{array}{lcl}
\mathsf{a}_1 & \leftarrow & \mathsf{a}_2, \mathsf{a}_3, \mathsf{a}_4, \mathsf{a}_5 \\
\mathsf{a}_2 & \leftarrow & \mathsf{a}_3, \mathsf{a}_4, \mathsf{a}_5 \\
\mathsf{a}_3 & \leftarrow & \mathsf{a}_4, \mathsf{a}_5 \\
\mathsf{a}_4 & \leftarrow & \mathsf{a}_5
\end{array}
$$

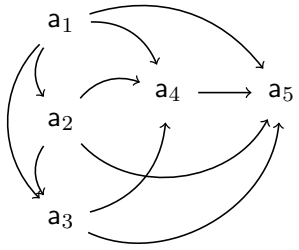


Figure 2.3: DTD d_2 and saturated type-dependency graph $\mathcal{G}_{\mathsf{d}_2}$

Inference and Derivations

Throughout this dissertation we will describe several inference systems for checking properties of XML queries and updates. Inference systems have been nicely formalized in [Gra03].

Definition 2.2.8 (Inference System)**.** *Let $\boldsymbol{J}$ be a set of predicates called judgments. An inference system $\boldsymbol{I}$ is a set of rules defined over judgments. A rule R is of the form*

$$\frac{\Upsilon}{\beta}$$

where $\Upsilon \subseteq \boldsymbol{J}$ is a set of premises and $\beta \in \boldsymbol{J}$ is a judgment called the goal of the rule.

An inference system proves properties (ground judgments) by repeated backward application of inference rules, in Prolog style. The whole proof is called derivation tree, that is a tree whose nodes are judgements and edges denote the application of inference rules. In particular, the root of the derivation tree is the property one aims to prove and the leafs are axioms. A set of rules is said to be deterministic if for each judgement created, at most one rule can be applied. This property is necessary to ease the derivation of an algorithm from the rules. In this dissertation, we will present only deterministic inference systems.

2.3 Query and Update Languages

XQuery is the XML query language standardized by the W3C [SCF+07]. XQuery stands to XML databases as SQL stands to relational databases. Nevertheless, the two languages are deeply different, because they are conceived for different data-models (trees vs. relations), and different deployments (the web vs. mainframes applications).

XQuery and SQL are both declarative languages. This is a crucial requirement for any query language in order to ensure logical and physical data independence. XQuery is a Turing-complete language, while SQL (without advanced features) is not. This just means that XQuery provides sufficient - perhaps, superabundant - expressive query power.

XQuery and SQL are both typed languages, this means that a type representing the result of an expression can be inferred. XQuery is a functional language, while SQL is not. This means that XQuery expressions are functions that can be nested or composed, and that the evaluation of each expression always returns a value with no side effect.

To illustrate, consider the following XQuery expression, where `y` is a free variable bound to the root of a bibliographic XML document and `x` is a bound variable of the query.

$$\texttt{for x in y}//\textit{book}\ \texttt{return}\ (\texttt{x}/\textit{title}, \texttt{x}/\textit{author})$$

The expression iterates over book nodes in the document. At each iteration step, it binds the current node to the variable `x`, and then it returns a *sequence* of title and author of the book.

Being declarative, XQuery allows to retrieve all books in the document, by invoking the XPath navigation $\texttt{y}//\textit{book}$, independently from how they are effectively stored and retrieved.

Being functional, expressions can be composed in a sequence, to form the return clause, or nested to form the whole iterative construct.

Being typed, it is possible to infer the regular expression type $(\mathsf{title}*, \mathsf{author}*)*$, to capture the fact that the result of the query is a possibly empty sequence of title and author lists.

XQuery is built on top of XPath. XPath is a language designed for navigating tree structures, along the axes of ancestors, descendants and siblings of a node. XQuery wraps-up XPath expressions, by introducing programming-language style constructs such as iteration, let binding, conditional expression, sequence construction, element construction, type-switching, etc.

The W3C working group has published a rich documentation on the XML query and update languages [SCF+07, RCD+11]. There is a large amount of details to consider in a rich language such as XQuery, hence it is a common practice to focus on a subset of the language that capture the most interesting constructs, so as to make the formal development intelligible yet effective.

In this work we deal with a large fragment of XQuery considered also in related approaches [BC09a, BC10], which is defined by the grammar of Table 2.1.

We comment on the main constructs of the language.

(Empty) The empty query is an expression performing no operation.

(Concatenation) A concatenation of queries is used to build new sequences of trees.

`q`	::=	`()`	(EMPTY)
		| `q,q`	(CONCATENATION)
		| <*a*>`q`</*a*>	(ELEMENTNODE)
		| "`txt`"	(TEXTNODE)
		| `x/axis::`ϕ	(STEP)
		| `for x in q return q`	(FOR)
		| `let x := q return q`	(LET)
		| `if (q) then q else q`	(IF)
`axis`	::=	`self`	(AXES)
		| `child`	
		| `descendant`	
		| `descendant-or-self`	
		| `parent`	
		| `ancestor`	
		| `ancestor-or-self`	
		| `preceding-sibling`	
		| `following-sibling`	
ϕ	::=	*a*	(NODE TESTS)
		| `text()`	
		| `node()`	

Table 2.1: Query Language

(Node Element and Node Text Construction) The importance of node element and text constructions stems from the fact that they are the only query expression that extend the input store with new data. Their use is twofold. They can be used in order to re-structure nodes of the base data, and then present it to the user (e.g. output a node changing the order of its children). They can be used by update expressions, in order to create new forests to be added to the store (e.g., insert a fresh textual node "`txt`" in a given position).

(XPath Steps) Navigational capabilities are at the heart of all query languages for semi-structured and, in general, for hierarchical data. Our language grammar includes downward, upward and sibling XPath navigational axis. This is a large fragment, considered also by related studies such as [BC09a, BC10].

Preceding and following axes, are not included. They can be easily encoded by standard rewriting As we will show later, this can be done in a both correct and complete way. Therefore, the language we consider simulates all XPath navigational axes.

(Iteration, Let binding and Conditional) Iteration (For), let binding (Let) and conditional expression (If) represent the most powerful constructs of the language, in terms of expressivity. This set of constructs subsumes the standard FLWR XQuery fragment [SCF+07], that features the where clause instead of the conditional if. Indeed, while the where clause nested in a let or for expression can be simulated by using an if-then-else expression, the converse is not always true since the where clause can express only an if-then condition. Therefore the language is strictly more expressive than its FLWR version.

For the sake of presentation, we adopt some minor syntactic deviation from the W3C specification [SCF+07]. In XQuery variables are always prefixed by $, in order to avoid ambiguity during the parsing phase. However, in this work variables will be always clear from the context and thus, we avoid to prefix them with a reserved symbol. Also, element construction is denoted by <*a*>`q`</*a*>, without enclosing the inner query `q` into brackets, as defined in [SCF+07].

XPath node tests are denoted by ϕ. In the grammar, a stands for a tag name. We will use $/\phi$ as a shortcut for $/\texttt{child}::\phi$ while $//\phi$ is a (strict) descendant navigation that, according to the W3C, stands for

$$/\texttt{descendant-or-self}::\texttt{node}()/\texttt{child}::\phi$$

In the following we will use `step` to indicate an XPath step. The node filter `node()` is often denoted by the wildcard $*$. XPath expressions of the form

$$\texttt{x}/\texttt{step}_1/\texttt{step}_2/\ldots/\texttt{step}_n$$

u	::=	()	(EMPTY)
		| u, u	(CONCATENATION)
		| for x in q return u	(FOR)
		| let x := q return u	(LET)
		| if (q) then u_1 else u_2	(IF)
		| delete q_0	(DELETE)
		| rename q_0 as a	(RENAME)
		| insert q pos q_0	(INSERT)
		| replace q_0 with q	(REPLACE)
pos	::=	into (as first | as last)?	(POSITIONS)
		| after	
		| before	

Table 2.2: Update Language

are supported by the grammar by mean of iteration, as follows.

$$\begin{array}{l}\texttt{for } x_1 \texttt{ in } x/step_1 \texttt{ return} \\ \quad \texttt{for } x_2 \texttt{ in } x_1/step_2 \texttt{ return} \\ \qquad \vdots \\ \qquad \texttt{for } x_n \texttt{ in } x_{n-1}/step_n \texttt{ return } x_n/\texttt{self} :: \texttt{node}()\end{array}$$

Recently the W3C standardized an update language for XML called XQuery Update Facility [RCD+11]. This includes a set of commands that can be used to modify instances of the XML data model, thus introducing *effects* to the query language.

As for queries, update commands are declarative and typed. This means that it is possible to modify an XML document, independently from how this is stored or retrieved, and also to compute a type describing the modified document. However, XML updates are not written in a functional way. They are not true functions with a return value that can be arbitrarily composed or nested.

An example of XML update is the following.

$$\texttt{insert } < book/ > \texttt{ into } x/bib$$

which inserts a fresh node labeled as *book* below the root labeled with *bib*.

XQuery Update Facility is built on XQuery and XPath, that are indeed used to fill the arguments of update commands.

The subset of XQuery Update Facility [RCD+11] which we consider is defined in Table 2.2. It is essentially composed of two parts, one of programming constructs shared with the query language, and one of commands modifying the input store.

(Basic Programming) Like for queries, updates can be composed sequentially or can be composed by means of let/for statements, where only the return part can contain update operations. The conditional expression allows to apply different updates, depending on the boolean value of a conditional query.

(Side-effects) In this work we consider all main constructs of the XQuery Update Facility, namely node deletion, renaming, insertion and replacement. We consider only replace update affecting *node* locations. The extension to the replace node-value command is straightforward [1], as well as that to the remaining update commands. Other extensions to the update language are discussed in Chapter 7.

2.4 Formal Semantics

In this section we give the formal semantics of XQuery and XQuery Update Facility. This will be used in the proofs to show the correctness of our independence analysis. We begin by defining the semantics of XPath steps as they are the building blocks of both languages.

2.4.1 Semantics of XPath Steps

Tree relations The semantics of axis navigation and text filtering is defined in [DFF+10]. If σ is a store, the binary parent-child relation among store locations is denoted by $Child_\sigma \subseteq dom(\sigma) \times dom(\sigma)$, and $(l, l') \in Child_\sigma$ when $l = a[L \cdot l' \cdot L'] \in \sigma$. We denote by $Parent_\sigma$ the inverse of $Child_\sigma$, by $Descendant_\sigma$ the transitive closure of $Child_\sigma$, and by $Ancestor_\sigma$ the inverse of $Descendant_\sigma$. Finally, $Self_\sigma$ denotes the identity function on $dom(\sigma)$. The binary relation that holds between a location and its immediate-right-sibling is denoted by $NextSibling_\sigma \subseteq dom(\sigma) \times dom(\sigma)$. We have that $(l, l') \in NextSibling_\sigma$ when $l_a \leftarrow a[L \cdot l \cdot l' \cdot L'] \in \sigma$, for some a. We denote by $FollowingSibling_\sigma$ the transitive closure of $NextSibling_\sigma$, and by $PrecedingSibling_\sigma$ the inverse of $FollowingSibling_\sigma$. If R is one of the binary relations defined above, provided a location l, we write $R(l)$ for the set of locations l' such that $(l, l') \in R$. For instance $Child_\sigma(l)$ denotes the set of children of l and $FollowingSibling_\sigma(l)$ the set of following sibling nodes of l. Of course, it holds $Self_\sigma(l) = \{\ l\ \}$. Finally $docOrder_\sigma \subseteq dom(\sigma) \times dom(\sigma)$ denotes the partial document-order among store locations, that is total for locations belonging to the same tree, but is undefined for locations belonging to distinct trees of the store. If R_σ is a relation over σ defined above, we denote by $R_t \subseteq R_\sigma$ its subset concerning all and only the locations belonging to the tree $t = (\sigma, l_t)$.

[1] `replace value of` q_0 `with q` becomes `let x :=` q_0 `return replace x/`$text()$ `with q`

Axis Semantics The semantics of an XPath step is given by the composition of the semantics of a navigational axis `axis` and that of a test filtering ϕ. The semantics of a navigation axis `axis` wrt a store σ and an input location l, is denoted by $[\![\texttt{axis}]\!]^l_\sigma$ and defined as follows.

$$
\begin{aligned}
[\![\texttt{self}]\!]^l_\sigma &\stackrel{def}{=} \mathit{Self}_\sigma(l)\\
[\![\texttt{child}]\!]^l_\sigma &\stackrel{def}{=} \mathit{Child}_\sigma(l)\\
[\![\texttt{descendant}]\!]^l_\sigma &\stackrel{def}{=} \mathit{Descendant}_\sigma(l)\\
[\![\texttt{descendant-or-self}]\!]^l_\sigma &\stackrel{def}{=} \mathit{Descendant}_\sigma(l) \cup \mathit{Self}_\sigma(l)\\
[\![\texttt{parent}]\!]^l_\sigma &\stackrel{def}{=} \mathit{Parent}_\sigma(l)\\
[\![\texttt{ancestor}]\!]^l_\sigma &\stackrel{def}{=} \mathit{Ancestor}_\sigma(l)\\
[\![\texttt{ancestor-or-self}]\!]^l_\sigma &\stackrel{def}{=} \mathit{Ancestor}_\sigma(l) \cup \mathit{Self}_\sigma(l)\\
[\![\texttt{following-sibling}]\!]^l_\sigma &\stackrel{def}{=} \mathit{FollowingSibling}_\sigma(l)\\
[\![\texttt{preceding-sibling}]\!]^l_\sigma &\stackrel{def}{=} \mathit{PrecedingSibling}_\sigma(l)
\end{aligned}
$$

The semantics of a test filtering wrt a store σ and an input location l, is denoted by $[\![\phi]\!]^l_\sigma$ and defined by the following equations.

$$
\begin{aligned}
[\![\texttt{a}]\!]^l_\sigma &\stackrel{def}{=} \{\, l \mid l \leftarrow \texttt{a}[L] \in \sigma \,\}\\
[\![\texttt{text()}]\!]^l_\sigma &\stackrel{def}{=} \{\, l \mid l \leftarrow \text{“}\texttt{txt}\text{”} \in \sigma \,\}\\
[\![\texttt{node()}]\!]^l_\sigma &\stackrel{def}{=} \{\, l \mid l \in \sigma \,\}
\end{aligned}
$$

Notice that this rules output either input location, when it satisfies the filtering condition, or the empty set.

The following examples illustrate the above definitions. Consider the store σ in Figure 2.1.

$$
\begin{aligned}
[\![\texttt{descendant-or-self}]\!]^{l_{t_1}}_\sigma &= \mathit{dom}(\sigma)\\
[\![\texttt{following-sibling}]\!]^{l_2}_\sigma &= \{l_3\}\\
[\![\texttt{text()}]\!]^{l'_2}_\sigma &= \{l'_2\}\\
[\![\texttt{author}]\!]^{l'_2}_\sigma &= \varnothing
\end{aligned}
$$

An XPath axis `axis` is said to be downward if it is either `child`, `descendant` or `descendant-or-self`, backward if it is `parent`, `ancestor` or `ancestor-or-self`, and horizontal if it is `preceding-sibling` or `following-sibling`.

Step Semantics An XPath step is evaluated by composing axis navigation and step filtering. An axis navigation results in a set of locations $\mathcal{L}$. This set is explicitly casted into a sequence L that respects document order. We model this by using an abstract predicate $order_\sigma(\mathcal{L}) = L$ that holds iff (i) $l \in \mathcal{L} \iff l \in L$ and (ii) $L = (L_1 \cdot l \cdot l' \cdot L_2) \implies (l, l') \in docOrder_\sigma$. Finally, L is filtered according to ϕ. Provided that $[\![\texttt{axis}]\!]^l_\sigma = \mathcal{L}$ and $order_\sigma(\mathcal{L}) = (l_1, \ldots, l_n)$ step semantics is defined as follows.

$$[\![\texttt{axis} :: \phi]\!]^l_\sigma \stackrel{def}{=} ([\![\phi]\!]^{l_1}_\sigma, [\![\phi]\!]^{l_2}_\sigma, \ldots, [\![\phi]\!]^{l_n}_\sigma)$$

When $[\![\phi]\!]^l_\sigma = \varnothing$ it is treated as the empty sequence.

Notice that this definition slightly differs from the one in [SCF+07] where sequences obtained by the evaluation of backward and preceding-sibling axes are returned in inverse document order. Of course, this detail has no impact on our static analysis.

The following examples illustrate the above definition. Consider the store σ in Figure 2.1.

$$\begin{aligned}
[\![\texttt{descendant-or-self} :: \texttt{node()}]\!]^{l_{t_1}}_\sigma &= dom(\sigma) \\
[\![\texttt{descendant-or-self} :: \texttt{text()}]\!]^{l_{t_1}}_\sigma &= (l'_2, l''_3) \\
[\![\texttt{following-sibling} :: \texttt{author}]\!]^{l_2}_\sigma &= (l_3)
\end{aligned}$$

Notice that $[\![\texttt{axis} :: \phi]\!]^l_\sigma$ produces a sequence of nodes without duplicates.

2.4.2 XQuery Semantics

Query semantics is specified in [DFF+10], while we are highly inspired by the succinct and elegant formalization made in [BC09b], from which we borrow some notions that are needed for our own presentation.

Query semantics is denoted by the following judgment

$$\sigma, \gamma \vDash \texttt{q} \Rightarrow \sigma', L$$

meaning that the execution of the query $\texttt{q}$ over σ outputs a sequence of locations L and a new store σ', including σ plus new elements built by $\texttt{q}$. The dynamic environment γ binds each free variable of $\texttt{q}$ to a sequence of locations L in σ, this is denoted by $\gamma[\texttt{x} \mapsto L]$ or simply by $\gamma[\texttt{x} \mapsto l]$ when $L = (l)$.

Query semantics is reported in Table 2.3. In the rules we also make use of two auxiliary judgements $\vDash^{\mathsf{copy}}$ and $\vDash^\star$ that model the copy of a sequence of elements and the iteration over a sequence of elements. These are defined by rule (SQ-Copy) and (SQ-Iter), respectively.

$$\frac{}{\sigma, \gamma \vDash () \;\Rightarrow\; \sigma, ()} \text{(SQ-Empty)}$$

$$\frac{\sigma, \gamma \vDash \mathtt{q}_1 \;\Rightarrow\; \sigma_2, L_1 \quad \sigma_2, \gamma \vDash \mathtt{q}_2 \;\Rightarrow\; \sigma_3, L_2}{\sigma, \gamma \vDash \mathtt{q}_1, \mathtt{q}_2 \;\Rightarrow\; \sigma_3, L_1 \cdot L_2} \text{(SQ-Concat)}$$

$$\frac{\sigma, \gamma \vDash \mathtt{q} \overset{\text{copy}}{\Rightarrow} \sigma_2, L \quad l \notin dom(\sigma_2)}{\sigma, \gamma \vDash \texttt{<}a\texttt{>}\mathtt{q}\texttt{</}a\texttt{>} \;\Rightarrow\; \sigma_2[l := a[L]], l} \text{(SQ-Elt)} \qquad \frac{l \notin dom(\sigma)}{\sigma, \gamma \vDash \text{“}\mathtt{txt}\text{”} \;\Rightarrow\; \sigma_2[l := \text{“}\mathtt{txt}\text{”}], l} \text{(SQ-Text)}$$

$$\frac{\begin{array}{c}\sigma, \gamma \vDash \mathtt{q} \;\Rightarrow\; \sigma_0, L_0 \\ (L', \sigma_1) = copy(L_0, \sigma_0) \qquad \sigma' = \sigma_0 \cup \sigma_1\end{array}}{\sigma, \gamma \vDash \mathtt{q} \overset{\text{copy}}{\Rightarrow} \sigma', L'} \text{(SQ-Copy)}$$

$$\frac{\sigma, \gamma \vDash \mathtt{q}_1 \;\Rightarrow\; \sigma_2, L \quad \sigma_2, \gamma[\mathtt{x} \mapsto L] \vDash^{\star} \mathtt{q}_2 \;\Rightarrow\; \sigma_3, L'}{\sigma, \gamma \vDash \texttt{for x in } \mathtt{q}_1 \texttt{ return } \mathtt{q}_2 \;\Rightarrow\; \sigma_3, L'} \text{(SQ-For)}$$

$$\frac{}{\sigma, \gamma[\mathtt{x} \mapsto ()] \vDash^{\star} \mathtt{q} \;\Rightarrow\; \sigma, ()} \text{(SQ-IterBase)} \qquad \frac{\begin{array}{c}\sigma, \gamma[\mathtt{x} \mapsto l] \vDash \mathtt{q} \;\Rightarrow\; \sigma_2, L_1 \\ \sigma_2, \gamma[\mathtt{x} \mapsto L] \vDash^{\star} \mathtt{q} \;\Rightarrow\; \sigma_3, L_2\end{array}}{\sigma, \gamma[\mathtt{x} \mapsto l \cdot L] \vDash^{\star} \mathtt{q} \;\Rightarrow\; \sigma_3, L_1 \cdot L_2} \text{(SQ-Iter)}$$

$$\frac{\sigma, \gamma \vDash \mathtt{q}_1 \;\Rightarrow\; \sigma_2, L \quad \sigma_2, \gamma[\mathtt{x} \mapsto L] \vDash \mathtt{q}_2 \;\Rightarrow\; \sigma_3, L'}{\sigma, \gamma \vDash \texttt{let x := } \mathtt{q}_1 \texttt{ return } \mathtt{q}_2 \;\Rightarrow\; \sigma_3, L'} \text{(SQ-Let)}$$

$$\frac{\sigma, \gamma \vDash \mathtt{q} \;\Rightarrow\; \sigma_2, l \cdot L \quad \sigma_2, \gamma \vDash \mathtt{q}_1 \;\Rightarrow\; \sigma_3, L_1}{\sigma, \gamma \vDash \texttt{if (q) then } \mathtt{q}_1 \texttt{ else } \mathtt{q}_2 \;\Rightarrow\; \sigma_3, L_1} \text{(SQ-If1)}$$

$$\frac{\sigma, \gamma \vDash \mathtt{q} \;\Rightarrow\; \sigma_2, () \quad \sigma_2, \gamma \vDash \mathtt{q}_2 \;\Rightarrow\; \sigma_3, L_2}{\sigma, \gamma \vDash \texttt{if (q) then } \mathtt{q}_1 \texttt{ else } \mathtt{q}_2 \;\Rightarrow\; \sigma_3, L_2} \text{(SQ-If2)}$$

$$\frac{\gamma(\mathtt{x}) = l \cdot () \quad [\![\mathtt{axis} :: \phi]\!]^l_\sigma = L}{\sigma, \gamma \vDash \mathtt{x/axis} :: \phi \;\Rightarrow\; \sigma, L} \text{(SQ-Step)}$$

Table 2.3: Query evaluation rules

We illustrate the semantics of the language through some running examples. Consider for instance the following iteration, that returns all parents of textual nodes, evaluated over the tree t_1 in Figure 2.1, wrt the environment $\gamma = \{\, \mathtt{x} \mapsto l_{t_1} \,\}$.

$$\frac{\textcircled{1} \qquad\qquad \textcircled{2}}{\sigma, \gamma \vDash \mathtt{x/descendant::text()} \;\Rightarrow\; \sigma, L \qquad \sigma, \gamma[\mathtt{y} \mapsto L] \vDash^{\star} \mathtt{y/parent::node()} \;\Rightarrow\; \sigma, L'}$$
$$\sigma, \gamma \vDash \mathtt{for\ y\ in\ x/descendant::text()\ return\ y/parent::node()} \;\Rightarrow\; \sigma, L'$$

The semantics of iteration is standard and modeled by rule (SQ-For). The resulting sequence of the left subexpression L is bound with variable y and then used for iterating over the right subexpression. Notice that the evaluation does not change the store σ.

Let us explore more in detail the derivation.

① The left branch is derived by applying rule (SQ-Step) as follows

$$\frac{[\![\mathtt{descendant::text()}]\!]^{l_{t_1}}_{\sigma} = (l'_2, l''_3)}{\sigma, \gamma \vDash \mathtt{x/descendant::text()} \;\Rightarrow\; \sigma, (l'_2, l''_3)}$$

The result sequence is composed of all textual nodes that are descendants of the root.

② The right branch is derived by applying rule (SQ-Iter) as follows

$$\frac{\dfrac{[\![\mathtt{parent::node()}]\!]^{l'_2}_{\sigma} = (l_2)}{\sigma, \gamma[\mathtt{y} \mapsto l'_2] \vDash \mathtt{y/parent::node()} \;\Rightarrow\; \sigma, (l_2)} \qquad \dfrac{[\![\mathtt{parent::node()}]\!]^{l''_3}_{\sigma} = (l'_3)}{\sigma, \gamma[\mathtt{y} \mapsto l''_3] \vDash \mathtt{y/parent::node()} \;\Rightarrow\; \sigma, (l'_3)}}{\sigma, \gamma[\mathtt{y} \mapsto (l'_2, l''_3)] \vDash^{\star} \mathtt{y/parent::node()} \;\Rightarrow\; \sigma, (l_2, l'_3)}$$

Here step evaluation is performed for each location bound to y. The results of the evaluations are concatenated and then returned.

The construction of new elements is the only command that extends the input store during query evaluation.

To illustrate, consider the following query listing all books of tree t_1 in Figure 2.1, wrt the environment $\gamma = \{\, \mathtt{x} \mapsto l_{t_1} \,\}$.

$$\frac{\dfrac{\sigma, \gamma \vDash \mathtt{x/descendant::}\mathit{book} \;\Rightarrow\; \sigma, (l_1) \qquad (l_c, \sigma_c) = \mathit{copy}(l_1, \sigma) \qquad \sigma' = \sigma \cup \sigma_c}{\sigma, \gamma \vDash \mathtt{x/descendant::}\mathit{book} \overset{\mathsf{copy}}{\Rightarrow} \sigma', (l_c)} \quad l \notin \mathit{dom}(\sigma')}{\sigma, \gamma \vDash \mathtt{<}\mathit{list}\mathtt{>x/descendant::}\mathit{book}\mathtt{</}\mathit{list}\mathtt{>} \;\Rightarrow\; \sigma'[l := \mathit{list}[l_c]], l_c}$$

In this case rule (SQ-Elt) is applied. First, the inner query selecting all books in the store is evaluated, resulting in the only location l_1. Second, the whole subtree rooted at l_1 is copied to a fresh isomorphic tree $\sigma_c @ l_c$, by means of rule (SQ-Copy). This operation is modeled here by using an abstract predicate $copy(l_1, \sigma)$. The output store σ' is the extension of σ with σ_c. Finally, l_c becomes the child of a fresh node location l representing the element node labeled with `list`.

Rewriting of `preceding` and `following` axes As said before, following and preceding axes are not directly dealt with by our system, but they can be expressed by means of rewriting. We can show that the rewriting is complete, in the sense that it gives the same result set as the standard semantics.

We show the case of the `following` axis, since `preceding` is analogous. Below σ is a fixed store, while l, l_a and l_f are locations belonging to σ. The W3C semantics for the axis is as follows.

$$[\![\texttt{following} :: \phi]\!]^{l}_{\sigma} \stackrel{def}{=} order_{\sigma}(\mathcal{L}) \tag{2.1}$$

where $\mathcal{L}$ is

$$\bigcup_{\substack{l_a \in [\![\texttt{ancestor-or-self} :: \texttt{node}()]\!]^{l}_{\sigma} \\ l_f \in [\![\texttt{following-sibling} :: \texttt{node}()]\!]^{l_a}_{\sigma} \\ l' \in [\![\texttt{descendant-or-self} :: \phi]\!]^{l_f}_{\sigma}}} \{\, l' \,\}$$

Step $\texttt{x/following} :: \phi$ is rewritten in our language in the following way

$$\texttt{x/following} :: \phi \;\approx\; \texttt{q}_\texttt{f}(\phi)$$

where

$$\begin{aligned} \texttt{q}_\texttt{f}(\phi) \;=\; & \texttt{for } \texttt{x}_a \texttt{ in x/ancestor-or-self} :: \texttt{node}() \\ & \quad \texttt{for } \texttt{x}_f \texttt{ in } \texttt{x}_a\texttt{/following-sibling} :: \texttt{node}() \\ & \quad\quad \texttt{return } \texttt{x}_f\texttt{/descendant} :: \phi \end{aligned}$$

Hence, provided that the judgment $\sigma, \texttt{x} \mapsto l \vDash \texttt{q}_\texttt{f}(\phi) \;\Rightarrow\; \sigma, L$ holds, we have that L is

$$\begin{gathered} \prod_{\substack{(l_{a_1}, \ldots, l_{a_n}) = order_{\sigma}([\![\texttt{ancestor} :: \texttt{node}()]\!]^{l}_{\sigma}) \\ (l_{f_{i,1}}, \ldots, l_{f_{i,m}}) = order_{\sigma}([\![\texttt{following-sibling} :: \texttt{node}()]\!]^{l_{a_i}}_{\sigma}) \\ L_{(i,j)} = order_{\sigma}([\![\texttt{descendant-or-self} :: \phi]\!]^{l_{f_{i,j}}}_{\sigma})}} L_{(i,j)} \end{gathered} \tag{2.2}$$

where $\prod$ denotes sequence concatenation, with $i = 1..n$ and $j = 1..m$.

We claim that a location is in the result of (2.1) if and only if it is in the result of (2.2). This holds since the resulting locations of all intermediate steps are the same. Furthermore, if one also wants to enforce that resulting sequences of both (2.1) and (2.2) are output in the same order (that is, document order), it is sufficient to impose that locations in the result of step $[\![\texttt{ancestor} :: \texttt{node()}]\!]^l_\sigma$ in (2.2) are iterated in inverted document order, for instance by imposing $i = n..1$.

2.4.3 XQuery Update Semantics

In this section we formally define the semantics of update expressions.

As a convention in update expressions, we write $\mathtt{q}_0$ for the *target* expression that is retrieving the nodes in the input document that are target of the update (e.g. deleted nodes are target nodes). We simply write $\mathtt{q}$ for the *source* update subexpression, that is retrieving or producing elements to insert in the database by means of insert or replace commands. According to the W3C semantics [RCD+11] the target expression $\mathtt{q}_0$ is required to output a single node otherwise a run time error occurs.

In the W3C specification update evaluation is split into three phases.

1. Creation of an *update pending list* (UPL) of atomic update commands;
2. Execution of a sanity check on this list, and reorder of atomic updates;
3. Application of the UPL on the input store so as to update the document.

An update pending list ω is a sequence of *atomic update commands*. An atomic update commands ι in of the following form.

$$\begin{array}{rcl} \iota & ::= & \texttt{del}(l) \\ & & \mid \texttt{ren}(l, a) \\ & & \mid \texttt{ins}(L, \texttt{pos}, l) \\ & & \mid \texttt{repl}(l, L) \end{array}$$

where l is the *target location*, L the sequence of roots of source elements to be inserted, and `pos` is an insertion position defined as before.

Creation of the UPL The creation of the UPL from an update `u` is denoted as follows

$$\sigma, \gamma \vDash \mathtt{u} \Rightarrow \sigma_\omega, \omega$$

As usual, γ binds u free variables to locations in σ and the store σ_ω that extends σ contains newly created locations potentially used in the UPL ω.

$$\frac{}{\sigma, \gamma \vDash () \;\Rightarrow\; \sigma, \epsilon}\ (\text{SU-EMPTY})$$

$$\frac{\sigma_1, \gamma \vDash \mathtt{u}_1 \;\Rightarrow\; \sigma_2, \omega_1 \quad \sigma_2, \gamma \vDash \mathtt{u}_2 \;\Rightarrow\; \sigma_3, \omega_2}{\sigma_1, \gamma \vDash \mathtt{u}_1, \mathtt{u}_2 \;\Rightarrow\; \sigma_3, \omega_1; \omega_2}\ (\text{SU-CONCAT})$$

$$\frac{\sigma_1, \gamma \vDash \mathtt{q} \;\Rightarrow\; L, \sigma_2 \quad \sigma_2, \gamma[\mathtt{x} \mapsto L] \vDash^{\star} \mathtt{u} \;\Rightarrow\; \sigma_3, \omega}{\sigma_1, \gamma \vDash \mathtt{for\ x\ in\ q\ return\ u} \;\Rightarrow\; \sigma_3, \omega}(\text{SU-FOR})$$

$$\frac{\sigma_1, \gamma \vDash \mathtt{q} \;\Rightarrow\; L, \sigma_2 \quad \sigma_2, \gamma[\mathtt{x} \mapsto L] \vDash \mathtt{u} \;\Rightarrow\; \sigma_3, \omega}{\sigma_1, \gamma \vDash \mathtt{let\ x\ :=\ q\ return\ u} \;\Rightarrow\; \sigma_3, \omega}(\text{SU-LET})$$

$$\frac{\sigma_1, \gamma \vDash \mathtt{q} \;\Rightarrow\; \sigma_2, l \cdot L \quad \sigma_2, \gamma \vDash \mathtt{u}_1 \;\Rightarrow\; \sigma_3, \omega_1}{\sigma_1, \gamma \vDash \mathtt{if\ (q)\ then\ u_1\ else\ u_2} \;\Rightarrow\; \sigma_3, \omega_1}(\text{SU-IF1})$$

$$\frac{\sigma_1, \gamma \vDash \mathtt{q} \;\Rightarrow\; \sigma_2, () \quad \sigma_2, \gamma \vDash \mathtt{u}_2 \;\Rightarrow\; \sigma_3, \omega_2}{\sigma_1, \gamma \vDash \mathtt{if\ (q)\ then\ u_1\ else\ u_2} \;\Rightarrow\; \sigma_3, \omega_2}(\text{SU-IF2})$$

$$\frac{\sigma_1, \gamma \vDash \mathtt{q} \overset{\text{copy}}{\Rightarrow} \sigma_2, L \quad \sigma_2, \gamma \vDash \mathtt{q}_0 \;\Rightarrow\; \sigma_3, l_0}{\sigma_1, \gamma \vDash \mathtt{insert\ q\ pos\ q_0} \;\Rightarrow\; \sigma_3, \mathtt{ins}(L, \mathtt{pos}, l_0)}(\text{SU-INSERT})$$

$$\frac{\sigma_1, \gamma \vDash \mathtt{q}_0 \;\Rightarrow\; \sigma_2, l_0}{\sigma_1, \gamma \vDash \mathtt{delete\ q_0} \;\Rightarrow\; \sigma_2, \mathtt{del}(l_0)}(\text{SU-DELETE})$$

$$\frac{\sigma_1, \gamma \vDash \mathtt{q}_0 \;\Rightarrow\; \sigma_2, l_0 \quad \sigma_2, \gamma \vDash \mathtt{q} \overset{\text{copy}}{\Rightarrow} \sigma_3, L}{\sigma_1, \gamma \vDash \mathtt{replace\ q_0\ with\ q} \;\Rightarrow\; \sigma_3, \mathtt{repl}(l_0, L)}(\text{SU-REPLACE})$$

$$\frac{\sigma_1, \gamma \vDash \mathtt{q}_0 \;\Rightarrow\; \sigma_2, l}{\sigma_1, \gamma \vDash \mathtt{rename\ q_0\ as}\ b \;\Rightarrow\; \sigma_2, \mathtt{ren}(l_0, b)}(\text{SU-RENAME})$$

$$\frac{}{\sigma, \gamma[\mathtt{x} \mapsto ()] \vDash^{\star} \mathtt{u} \;\Rightarrow\; \sigma, \epsilon}\ (\text{SU-ITERBASE})$$

$$\frac{\sigma_1, \gamma[\mathtt{x} \mapsto l] \vDash \mathtt{u} \;\Rightarrow\; \sigma_2, \omega_1 \qquad \sigma_2, \gamma[\mathtt{x} \mapsto L] \vDash^{\star} \mathtt{u} \;\Rightarrow\; \sigma_3, \omega_2}{\sigma_1, \gamma[\mathtt{x} \mapsto l \cdot L] \vDash^{\star} \mathtt{u} \;\Rightarrow\; \sigma_3, \omega_1; \omega_2}\ (\text{SU-ITER})$$

Table 2.4: Rules for evaluating update expressions to pending update lists

$$\frac{\sigma(l') = a[L_1 \cdot l \cdot L_2]}{\sigma \vDash \texttt{del}(l) \rightsquigarrow \sigma[l' := a[L_1 \cdot L_2]]} \text{ (SAU-Delete)}$$

$$\frac{\sigma(l) = a[L]}{\sigma \vDash \texttt{ren}(l, b) \rightsquigarrow \sigma[l := b[L]]} \text{ (SAU-Rename)}$$

$$\frac{\sigma(l') = a[L_1 \cdot l \cdot L_2]}{\sigma \vDash \texttt{ins}(L, \texttt{before}, l) \rightsquigarrow \sigma[l' := a[L_1 \cdot L \cdot l \cdot L_2]]} \text{ (SAU-InsertBefore)}$$

$$\frac{\sigma(l') = a[L_1 \cdot l \cdot L_2]}{\sigma \vDash \texttt{ins}(L, \texttt{after}, l) \rightsquigarrow \sigma[l' := a[L_1 \cdot l \cdot L \cdot L_2]]} \text{ (SAU-InsertAfter)}$$

$$\frac{\sigma(l) = a[L_1 \cdot L_2]}{\sigma \vDash \texttt{ins}(L, \texttt{into}, l) \rightsquigarrow \sigma[l := a[L_1 \cdot L \cdot L_2]]} \text{ (SAU-InsertInto)}$$

$$\frac{\sigma(l) = a[L_1]}{\sigma \vDash \texttt{ins}(L, \texttt{into as first}, l) \rightsquigarrow \sigma[l := a[L \cdot L_1]]} \text{ (SAU-InsertFirst)}$$

$$\frac{\sigma(l) = a[L_1]}{\sigma \vDash \texttt{ins}(L, \texttt{into as last}, l) \rightsquigarrow \sigma[l := a[L_1 \cdot L]]} \text{ (SAU-InsertLast)}$$

$$\frac{\sigma(l') = a[L_1 \cdot l \cdot L_2]}{\sigma \vDash \texttt{repl}(l, L) \rightsquigarrow \sigma[l' := a[L_1 \cdot L \cdot L_2]]} \text{ (SAU-Replace)}$$

$$\frac{}{\sigma \vDash \epsilon \rightsquigarrow \sigma} \text{ (SAU-Empty)} \qquad \frac{\sigma \vDash \omega_j \rightsquigarrow \sigma' \quad \sigma' \vDash \omega_k \rightsquigarrow \sigma'' \quad \{j, k\} = \{1, 2\}}{\sigma \vDash \omega_1, \omega_2 \rightsquigarrow \sigma''} \text{ (SAU-Iter)}$$

Table 2.5: Update pending list application

Sanity Check and Update Reordering In this phase, the sanity checks verify that UPL is consistent. As in [BC09a], we do not model this phase here, and we defer the work to an abstract predicate $sanitycheck(\omega)$. The check enforces for instance that the same node is not a target of multiple rename or replace operators, that text nodes are not the targets of inserts, and that the elements being inserted are rooted at element nodes, as specified in [RCD+11].

Application of the UPL Applying the UPL ω to the input store σ_ω produces the updated store $\sigma_{\mathtt{u}}$. This is denoted by

$$\sigma_\omega \vDash \omega \rightsquigarrow \sigma_{\mathtt{u}}$$

Of course, all locations used in ω are contained in σ_ω.

The composition of all phases of update semantics is denoted by

$$\sigma, \gamma \vDash \mathtt{u} : \sigma_{\mathtt{u}}$$

and it is defined as follows

$$\frac{\sigma, \gamma \vDash \mathtt{u} \Rightarrow \sigma_\omega, \omega \quad sanitycheck(\omega) \quad \sigma_\omega \vDash \omega \rightsquigarrow \sigma_{\mathtt{u}}}{\sigma, \gamma \vDash \mathtt{u} : \sigma_{\mathtt{u}}}$$

As for queries, we explain the semantics of updates through some examples. Let us consider the execution of the update below that deletes all textual nodes in the tree t_1 of Figure 2.1, wrt the environment $\gamma = \{\ \mathtt{x} \mapsto l_{t_1}\ \}$.

```
for y in x/descendant::text() return delete y
```

The end-to-end semantics of the update consists of generating the UPL of nodes to delete, and then doing the sanity check on the UPL and of applying the update pending list so as to obtain the updated store σ'.

$$\frac{\dfrac{\dfrac{\textcircled{1}}{\sigma, \gamma \vDash \texttt{for y in x/descendant::text() return delete y} \Rightarrow \sigma, (\mathtt{del}(l_2'), \mathtt{del}(l_3''))} \quad sanitycheck(\mathtt{del}(l_2'), \mathtt{del}(l_3''))}{\sigma \vDash \mathtt{del}(l_2'), \mathtt{del}(l_3'') \rightsquigarrow \sigma'} \textcircled{2}}{\sigma, \gamma \vDash \texttt{for y in x/descendant::text() return delete y} : \sigma'}$$

① The derivation of the branch generating the UPL with locations to be deleted is the following.

$$\dfrac{\dfrac{\sigma,\gamma \vDash \texttt{x/descendant::text()} \Rightarrow \sigma,(l'_2,l''_3)}{\sigma,\gamma[\texttt{y}\mapsto l'_2] \vDash \texttt{delete } y \Rightarrow \sigma,(\texttt{del}(l'_2)) \qquad \sigma,\gamma[\texttt{y}\mapsto l''_3] \vDash \texttt{delete } y \Rightarrow \sigma,(\texttt{del}(l''_3))}}{\sigma,\gamma[\texttt{y}\mapsto (l'_2,l''_3)] \vDash^{\star} \texttt{delete y} \Rightarrow \sigma,(\texttt{del}(l'_2),\texttt{del}(l''_3))}}{\sigma,\gamma \vDash \texttt{for y in x/descendant::text() return delete y} \Rightarrow \sigma,(\texttt{del}(l'_2),\texttt{del}(l''_3))}$$

Here the UPL is computed by first evaluating the navigational expression selecting all textual nodes, and then by iterating over the resulting sequence of locations.

② The derivation of the branch applying the UPL is the following.

$$\dfrac{\dfrac{\sigma(l'_3)=\texttt{name}[l''_3] \quad \sigma''=\sigma[l'_3:=\texttt{name}[]]}{\sigma \vDash \texttt{del}(l''_3) \rightsquigarrow \sigma''} \qquad \dfrac{\sigma''(l_2)=\texttt{title}[l'_2] \quad \sigma'=\sigma[l_2:=\texttt{title}[]]}{\sigma'' \vDash \texttt{del}(l'_2) \rightsquigarrow \sigma'}}{\sigma \vDash \texttt{del}(l'_2),\texttt{del}(l''_3) \rightsquigarrow \sigma'}$$

Here the application of the UPL consists of removing the use of the target locations by means of rule (SAU-Iter) and (SAU-Delete). As mentioned before, in this semantics we do not explicitly take into account order of update application, in fact rule (SAU-Iter) specifies that two update pending *lists* can be executed in any order. In order to stress this, in the example, we chose to remove first location l''_3 and then l_2, while the input UPL presents them in inverse order. For instance for all valid sequences of updates targeting the same locations like

```
for x in y//a
return (delete x, rename x as b)
```

the semantics allows to apply non-deterministically the atomic delete and rename commands in any order. Whatever order of application is chosen, the result of the update is to delete all nodes labeled as a. Notice that once atomic updates are collected, we may have that a location is both target of an insert and of a rename. In fact, the sanity check just enforces that no location is target of multiple rename or replace.

Another interesting point to stress is that for deleted and replace updates, no node is destroyed in the store. In fact, if $\sigma,\gamma \vDash \texttt{u} \Rightarrow \sigma_\omega,\omega$ and $\sigma_\omega \vDash \omega \rightsquigarrow \sigma_\texttt{u}$ then $dom(\sigma) \subseteq dom(\sigma_\texttt{u})$. For a tree $t{=}(\sigma,l_t)$, $\texttt{u}(t)$ denotes the tree $(\sigma_\texttt{u}@l_t,l_t)$. Note that $dom(\sigma) \subseteq dom(\sigma_\texttt{u}@l_t)$ may not hold anymore, since $\sigma_\texttt{u}@l_t$ does not keep locations disconnected to the root l_t after the update.

To see how the input store is extended by updates such as insert and replace, we consider the following expression on the tree t_1 in Figure 2.1, wrt the environment $\gamma = \{\, \mathtt{x} \mapsto l_{t_1} \,\}$. The update inserts a new tree listing all books as a child of the root.

$$\dfrac{\dfrac{\textcircled{1}}{\sigma, \gamma \vDash \texttt{insert <list>x/descendant :: book</list> into x/self :: bib} \Rightarrow \texttt{ins}(L, \texttt{into}, l_{t_1})} \quad \mathit{sanitycheck}(\texttt{ins}(L, \texttt{into}, l_{t_1})) \quad \dfrac{\textcircled{2}}{\sigma \vDash \texttt{ins}(L, \texttt{into}, l_0) \rightsquigarrow \sigma'}}{\sigma, \gamma \vDash \texttt{insert <list>x/descendant :: book</list> into x/self :: bib} : \sigma'}$$

We explore a possible derivation tree of the update semantics, by means of an example.

① The derivation for the branch generating the UPL is the following.

$$\dfrac{\dfrac{\sigma, \gamma \vDash \texttt{x/self :: bib} \Rightarrow \sigma, (l_{t_1}) \quad \dfrac{\sigma, \gamma \vDash \texttt{x/descendant :: book} \Rightarrow \sigma_1, l_1 \quad (l_c, \sigma_c) = \mathit{copy}(l_1, \sigma_1) \quad \sigma' = \sigma_1 \cup \sigma_c}{\sigma, \gamma \vDash \texttt{<list>x/descendant :: book</list>} \overset{\text{copy}}{\Rightarrow} \sigma', l_c}}{\sigma, \gamma \vDash \texttt{insert <list>x/descendant :: book</list> into x/self :: bib} \Rightarrow \texttt{ins}(l_c, \texttt{into}, l_{t_1})}$$

Here the UPL is composed by a single atomic insert update command. The arguments of the atomic update are taken from the outputs of the *target* and the *source* query. From one hand, the target query simply returns the root of the document. From the other hand, the source query returns the copy of a fresh tree listing all books.

② The derivation concerning the application of the UPL is the following.

$$\dfrac{\sigma(l_{t_1}) = \texttt{bib}[l_1] \qquad \sigma' = \sigma[l_{t_1} := \texttt{bib}[l_1, l_c]]}{\sigma \vDash \texttt{ins}(l_c, \texttt{into}, l_{t_1}) \rightsquigarrow \sigma'}$$

Here the root location receives one more child. In this example we chose to insert the new location in last position. It is worth noticing that when position `into` is specified, our semantics, as well as the W3C one, allows to insert the node non-deterministically, in any position.

Assumptions on the Query Language In order to simplify the formal treatment, we do some assumptions on the query language.

(Node Element Construction) We assume that node element construction <*a*>q</*a*> is not used in the left-hand side expression of a for/let-expression. This restriction is met by a very large class of queries used in practice, while queries like

$$\texttt{let x :=} \ \texttt{<}a\texttt{>q</}a\texttt{>} \ \texttt{return} \ \texttt{<}b\texttt{>x</}b\texttt{>}$$

can be rewritten by simple variable substitution into

$$\texttt{<}b\texttt{><}a\texttt{>q</}a\texttt{></}b\texttt{>}$$

(Text Element Construction) Given that a constant query is independent of any update, we assume that "txt" is either nested in a for expression.

Updates Preserving the Schema We assume also that the updates are preserving the schema, in the sense that they do not introduce new chains. Delete updates that break schema constraints are however allowed. As we show later in Chapter 3, schema evolution is an issue that goes beyond the scope of detecting independence.

2.5 Query-Update Independence

We now formalize query-update independence. To this end we need a notion of equivalence over XML stores.

Definition 2.5.1 (Equivalence)**.** *Given two stores σ and σ', two locations $l \in \sigma$ and $l' \in \sigma'$ are said to be* value equivalent*, written $(\sigma, l) \cong (\sigma', l')$, iff the two trees $\sigma@l$ and $\sigma'@l'$ are isomorphic (they possibly differ only in terms of locations). We write $(\sigma, L) \cong (\sigma', L')$ to denote the value equivalence on location sequences $L = (l_1, \ldots, l_n)$ and $L' = (l'_1, \ldots, l'_n)$, with $l_i \in \sigma$ and $l'_i \in \sigma'$, and holding iff $(\sigma, l_i) \cong (\sigma', l'_i)$ for $i = 1..n$.*

Definition 2.5.2 (Independence: $\texttt{q} \perp\!\!\!\perp \texttt{u}$, $\texttt{q} \perp\!\!\!\perp_\texttt{d} \texttt{u}$)**.** *Let σ be a store and γ a variable environment over σ.*

A query q *and an update* u *are said to be independent wrt (σ, γ) if*

$$\sigma, \gamma \vDash \texttt{q} \Rightarrow \sigma_\texttt{q}, L_\texttt{q} \qquad \sigma, \gamma \vDash \texttt{u} : \sigma_\texttt{u} \qquad \sigma_\texttt{u}, \gamma \vDash \texttt{q} \Rightarrow \sigma'_\texttt{q}, L'_\texttt{q}$$

implies $(\sigma_\texttt{q}, L_\texttt{q}) \cong (\sigma'_\texttt{q}, L'_\texttt{q})$.

Also, q *and* u *are independent, written* $\texttt{q} \perp\!\!\!\perp \texttt{u}$*, iff they are independent for any pair (σ, γ).*

Finally, q *and* u *are independent wrt the DTD d, written* $\texttt{q} \perp\!\!\!\perp_\texttt{d} \texttt{u}$*, iff for every tree $t = (\sigma, l_t) \in d$ and γ, they are independent for (σ, γ).*

As a natural consequence of the fact that XML data are typed by a schema, we assume that our independence analysis is run in a context where all data remain consistent wrt the schema after each update. In case an update triggers schema evolution, then a larger task of schema maintenance has to be carried on. This task may imply existing views (queries) to be reformulated in order to be correct wrt the new schema, and thus it is likely to exclude any other kind of schema-based analysis until its completion.

Conclusions

In this chapter we presented the formal framework on top of which we build our work. We presented XML databases, schemas, query and update languages, and defined the problem we study: XML query update independence. In the next chapter we will revise the state of the art for detecting independence via static analysis.

Chapter 3

Query Update Independence: State of the Art

In this chapter we present the state of the art techniques for checking XML query-update independence. In the first part, we report on the theoretical background that is relevant to detecting independence via static analysis, and outline the intractability of the problem for practical cases. In the second part, we present a family of approximate algorithms developed for checking independence that, being sound but not complete, try to ensure a tractable analysis.

3.1 Decidability and Complexity of Exact Static Independence Analysis

Former studies by Benedikt and Cheney [BC09a] and by Raghavachari and Shmueli [RS06] provided the theoretical background of checking independence via static analysis. These works show that independence essentially inherits its computational difficulty from more fundamental problems such as query *equivalence* and *containment*. Because of this, the problem is challenging.

The following results give a flavor on how checking independence ranges from undecidability to tractability. Below, boolean atomic queries and updates are expressions without aggregates and tests on data values returning either the empty sequence or a constant value [BK06].

Theorem 3.1.1 ([BC09a])**.** *Independence is undecidable for XQuery queries and updates.*

Theorem 3.1.2 ([BC09a])**.** *For boolean atomic XQuery queries and updates, independence is decidable but non-elementary.*

Theorem 3.1.3 ([RS06])**.** *For queries and updates defined by using XPath expressions featuring descendant navigation, * wildcard operator and predicates, independence is NP-Complete.*

Theorem 3.1.4 ([RS06])**.** *Independence is polynomial for queries and updates defined by downward XPath expressions using at most two of the following features: descendant axis, * wildcard and predicates.*

In the following, we discuss the main steps of the proofs of the above theorems, with the aim of giving more details on the links between independence, equivalence and containment.

The main observation for relating independence with equivalence suggested in [BC09a] is that we can solve independence if we can solve the equivalence problem for *compositions* of queries and updates. Equivalence means that two programs always agree on their output, once evaluated on the same input database. That is, a query $\mathtt{q}$ is independent from an update $\mathtt{u}$ if the equivalence $\mathtt{q} \equiv \mathtt{q} \circ \mathtt{u}$ holds, provided that the composition $\mathtt{q} \circ \mathtt{u}$ and the notion of equivalence $\equiv$ are properly defined. Results on the decidability of independence follow from this idea, as we now illustrate.

General Undecidability of Independence Benedikt and Cheney showed that independence is undecidable with a reduction to the satisfiability problem for queries expressed as first order logic formulas over data trees (corresponding to XQuery expressions) [BC09a]. It is well known that satisfiability of first order logic is undecidable, and this is also true in the signature of trees. With signature of trees, we mean the binary and unary relations that allow to define trees (see [BK09] for more details). The idea is that we could solve independence if we had solved satisfiability of first order logic over data trees. Fix an XQuery query $\mathtt{q}$, and define a query update pair in the following way

$$\mathtt{q}' = \mathtt{if}\ (/root/a)\ \mathtt{then\ q\ else}\ () \qquad \mathtt{u} = \mathtt{insert}\ \texttt{<}a\texttt{/>}\ \mathtt{into}\ /root$$

Then,

$$\mathtt{q} \text{ is unsatisfiable for all } \sigma, \gamma \quad \text{if and only if} \quad \mathtt{q}' \text{ is independent of } \mathtt{u} \text{ for all } \sigma, \gamma$$

In fact, query $\mathtt{q}'$ outputs constantly the empty sequence (before and after $\mathtt{u}$) on any σ, γ, if and only if $\mathtt{q}$ is not satisfiable on any σ, γ. Since the satisfiability problem for XQuery queries is undecidable, we conclude that independence for XQuery queries and updates is generally undecidable. This is stated by Theorem 3.1.1.

Decidability of Boolean Independence for Core Atomic XQuery Benedikt and Cheney showed also that independence is decidable for a relevant subset of XML queries and updates [BC09a], as stated by Theorem 3.1.2. Again, the result is obtained from decidability of satisfiability of first order logic over trees. This result is interesting because it sheds light on the links between independence and query equivalence.

Each step of the proof is quite dense, and will be detailed next.

1. Define independence using a weaker notion of equivalence.

2. Restrict the query and update languages.

3. Show that, with such restrictions, queries and updates correspond to *first order interpretations*.

4. Show that first order interpretations can be composed and tested to be equivalent, thus conclude decidability of independence.

1) Define independence using a weaker notion of equivalence.

Several notions of equivalence can be used to define independence (i.e., $\mathtt{q} \equiv \mathtt{q} \circ \mathtt{u}$), notably, boolean equivalence, node equivalence or subtree equivalence. They naturally differ on the test they do. *Boolean* equivalence checks if the results of two expressions always agrees on non-emptiness. *Node* equivalence checks if node ids in result sequences are isomorphic. *Subtree* equivalence checks if forests of the two expressions are isomorphic. Decidability of boolean query equivalence has been showed in [BK06] for the class of atomic queries. Differently, decidability of node-equivalence and subtree-equivalence are still open problems. Consequently, in [BC09a] the authors studied the analogous problem of *boolean independence*. Boolean independence just checks if the result of a query changes from empty to non empty or vice-versa, after update. Now, for testing boolean independence it is sufficient to translate a query `q` in a boolean-query of the form `if (q) then` 1 `else` 0.

2) Restrict the query and update language.

Benedikt and Cheney [BC09a] considered XML queries and updates without aggregates and tests on data values, which is originally referred as Core Atomic XQuery in [BK06]. This is roughly equivalent to the language defined in Section 2.3. Because we want to check boolean independence, we will consider Boolean Core Atomic XQuery queries of the form `if (q) then` 1 `else` 0, and Core Atomic XQuery updates built on such expressions

3) Show that queries and updates correspond to first order interpretations.

Benedikt and Koch [BK06] showed that Boolean Core Atomic XQuery queries of XQuery are given by first-order queries in the signature of trees. First order interpretations are formulas that define the output structure in function of a given input. First order interpretations model queries. The crucial point is the following. It can be shown (see [BC09a]) that the addition of updates to Core Atomic XQuery still yields first order interpretations. This means that, while in general it is true that updates are more expressive than queries, this is not the case for Core Atomic XQuery. For this fragment queries and updates have the same expressive power, and they precisely correspond to first order interpretations.

4) Conclude decidability of independence.

Since first-order interpretations are closed under composition (Theorem 5.1 of [BK09]), the composition of a query and an update is again a first order interpretation. So the

goal now is to test equivalence of first order interpretations. In the special case of a boolean query, a first-order interpretation is just a first-order sentence (a formula with no variables). First-order sentences can be translated into tree automata. Equivalence of automata obtained by first order sentences can be decided. Therefore, equivalence of first order interpretations for boolean queries can be decided. Hence, equivalence of a query against the composition of a query and an update can be decided. Hence, independence can be effectively decided. It is worth noticing that it is critical for decidability that data value comparisons are disallowed.

Benedikt and Cheney also give the complexity of the problem when the tag alphabet is fixed. This captures the case where a schema is available. In this case, because the satisfiability problem for first order logic over labeled trees (with fixed alphabet) is non-elementary [Vor96] we obtain that the independence problem in the presence of the schema is non-elementary. As outlined in [Ben10], decidability of equivalence, and hence independence, when considering powerful notions of equivalence such as node and subtree equivalence is still an open question.

NP-Completeness for XPath$(//, *, [])$ fragment Another contribution to the study of XML independence comes from the work of Raghavachari and Shmueli [RS06]. This work investigates the complexity of detecting *conflicting* query and update expressions. A query and an update conflict when a node accessed by the query is also impacted by the update. This is a dual notion of independence, that indeed holds in the absence of conflicts. Detecting conflicting expressions has been proved NP-complete for XML queries and updates expressed in the downward fragment of XPath featuring * wildcard and branching, and a notion of delete/insert node-conflict reminiscent of node-equivalence (are the ids of resulting sequences isomorphic?). NP-completeness follows from a reduction to the complement of the containment problem for XPath. Actually, it follows from the dual non-containment problem. The non-containment problem for XPath asks wether two expressions disagrees on the emptiness of the result, for some trees. In brief, it turns out that detecting this property is enough to detect if the result of a query may change after update. For instance, by taking an instance for which query result is empty and adding some data so as to get a non-empty query result. This operation is possible if the expressions are not in the containment relation. Updates are defined in terms of queries, hence solving non-containment between query-paths and update-paths would solve the problem of determining if the expressions conflict. Miklau and Suciu previously showed that non-containment is NP-hard [MS04], that in turn determines the hardness of independence.

Polynomial fragments of XPath$(//, *, [])$ Along the line of the previous result, Ravagachari and Shmueli also showed that by further restricting the query and update language polynomial time algorithms for independence can be given [RS06]. This is stated by Theorem 3.1.3. This holds for queries and updates written with path expressions belonging to a subset of XPath$(//, *, [])$ which is employing at most two features among

descendant navigation "//", * wildcard and branching "[]". The authors in [RS06] showed the relationship with query containment of XPath expressions for such languages [MS04, AYCLS01]. This shows that tractable algorithms for independence can be given only by substantially restricting the query and update language. These subsets of XPath are quite restrained and may not support realistic XQuery workloads.

The results we discussed show that independence is not tractable in practical cases. Its general undecidability and hardness suggest that in order to do static analysis one is obliged to proceed with *approximate* algorithms, trading completeness with tractability, and then devise analysis that can be useful in practice. This is what motivated recent research efforts towards approximate techniques discussed next.

3.2 Approximate Algorithms

The hardness of independence leads to attack the problem with approximate solutions. Approximate algorithms are sound but not complete methods, that is, they either correctly detect an independence or conservatively raise a conflict. As we will see throughout this dissertation, approximate solutions can be extremely effective by exploiting structural properties of queries and updates.

The state-of-the-art techniques for detecting independence reformulate the problem in terms of an intersection problem. Precisely, testing independence is made by checking whether the data needed by a query intersects with the data affected by an update. The techniques now decompose the problem in two parts: 1) providing an abstract representation of the data that is accessed by the query and impacted by the update, and 2) defining the intersection test. Before illustrating the state-of-the-art, we discuss the problem of providing an abstract representation of the data and show a link with XML projection.

The problem of computing a subset of the input database that is accessed by a query, and that is sufficient for its execution, is known as XML data-projection [MS03, BCCN06]. This problem has been studied also for updates in [BBC+11]. The set of nodes involved by query/update execution is usually captured via static analysis. XML projection has developed the ideas of using either *XPath navigational paths* or *XML types* (when a schema is available) in order to abstract over the collection of data. These ideas gave birth to a series of works showing how projection can be used to optimize query execution [MS03, BCCN06] or for locally updating a database [BBC+11]. Indeed, all approaches in the current literature dealing with independence [GRS08, BC09a, BC10, BTCU12] are likely to follow these ideas. However, substantial technical differences arise between methods for projection and independence, as we will see next.

In the following, we present the state-of-the-art techniques for independence in detail. We begin by presenting the analysis based on XML types [BC09a] which are the closest to our work since they use schema information. Then we present the analysis based on XPath paths [GRS08] and then the approach with destabilizers [BC10].

3.2.1 Schema-based independence analysis for XML updates [BC09a]

This work is the first to propose a static analysis for detecting independence, in the presence of a schema. The idea at the basis of the schema-based analysis of [BC09a] is to infer *type-names* of nodes that are accessed by a query and impacted by an update, and then check that these do not overlap. The main contribution of [BC09a] is a type system that features very low complexity and which is able to detect a consistent number of independencies statically.

The technique targets a fragment of XQuery that is essentially the one presented in Section 2.3. Mild differences comes from the fact that queries can navigate on constructed data, and updates do not need to preserve the schema.

The independence analysis relies on three inference systems.

(RES) An inference system for typing the *result* (**RES**) of a query expression. The judgment $\mathtt{d}, \Gamma \vdash_{\mathbf{(RES)}} \mathtt{q} : \tau$ stands for a relation between the schema $\mathtt{d}$, the static variable environment Γ, the query $\mathtt{q}$ and the set of types τ where τ contains at least all types of nodes, defined in $\mathtt{d}$, that can be in the result sequences of $\mathtt{q}$.

(SAC) An inference system for typing the data *accessed* by a query, that is called static access cover (**SAC**). The judgment $\mathtt{d}, \Gamma \vdash_{\mathbf{(SAC)}} \mathtt{q} : \tau$ is a relation between the schema $\mathtt{d}$, the static variable environment Γ, the query $\mathtt{q}$ and the set of types τ where τ contains at least all types of nodes, defined in $\mathtt{d}$, that are needed to evaluate $\mathtt{q}$.

(IMP) An inference system for typing the data *impacted* by the update (**IMP**). The judgment $\mathtt{d}, \Gamma \vdash_{\mathbf{(IMP)}} \mathtt{q} : \tau$ is a relation between the schema $\mathtt{d}$, the static variable environment Γ, the query $\mathtt{u}$ and the set of types τ where τ contains at least all types of nodes, defined in $\mathtt{d}$, that are updated by $\mathtt{u}$.

We now illustrate through some examples how these three systems work, and how type inference is done.

Consider the bibliographic DTD $\mathtt{d}_3$ and the following query and update pair, for which we want to verify independence.

$\mathtt{q} = \mathtt{for\ x\ in\ doc/descendant::}\mathit{book}\ \mathtt{return\ x/}\mathit{author}$ $\qquad$ $\mathtt{u} = \mathtt{delete\ doc//}\mathit{title}$

We always assume that the context of the first navigational step of a query or update is the schema root type, thus in all of the examples of this chapter we assume that Γ is a static environment such that $\Gamma(\mathtt{doc}) = \{\ \mathtt{bib}\ \}$. The derivation tree for the inference of types accessed by $\mathtt{q}$ is the following.

$$\begin{aligned} \mathsf{bib} &\leftarrow \mathsf{book}* \\ \mathsf{book} &\leftarrow \mathsf{title}, \mathsf{author}+, \mathsf{publisher}+ \\ \mathsf{author}, \mathsf{publisher} &\leftarrow \mathsf{name} \\ \mathsf{title}, \mathsf{name} &\leftarrow \mathsf{String} \end{aligned}$$

Figure 3.1: DTD d_3

$$\dfrac{\dfrac{\textcircled{1}}{\mathsf{d}_3, \Gamma \vdash_{\mathbf{(SAC)}} \mathtt{doc/descendant} :: \mathit{book} : \tau_1} \quad \dfrac{\textcircled{2}}{\mathsf{d}_3, \Gamma \vdash_{\mathbf{(RES)}} \mathtt{doc/descendant} :: \mathit{book} : \tau_2} \quad \dfrac{\textcircled{3}}{\mathsf{d}_3, \Gamma[\mathtt{x} \mapsto \tau_2] \vdash_{\mathbf{(SAC)}} \mathtt{x}/\mathit{author} : \tau_3}}{\mathsf{d}_3, \Gamma \vdash_{\mathbf{(SAC)}} \mathtt{for\ x\ in\ doc/descendant} :: \mathit{book}\ \mathtt{return\ x}/\mathit{author} : \tau_1 \cup \tau_3}$$

The set of types accessed by `q` is the union of types accessed by its subexpressions, i.e. $\tau_1 \cup \tau_3$. Types are inferred by calling system **SAC** as shown by derivations ① and ③. What is inferred for the query in τ_1 and τ_3? Because `doc/descendant` $::$ *book* selects all book nodes in a store, in τ_1 we expect to find *book* and all of its ancestor types defined in d_1. Because `x`/*author* selects all author children of nodes bound to `x`, and `x` is only bound to book nodes, in τ_3 we expect to find types *book* and *author*. Derivation ② shows also the role of the inference system **RES** in the process. It is used in order to provide types for variable `x` in the subsequent **SAC** derivation.

By exploring the next level of derivation we can understand more peculiarities of type inference.

① The judgment $\mathsf{d}_3, \Gamma \vdash_{\mathbf{(SAC)}} \mathtt{doc/descendant} :: \mathit{book} : \tau_1$ is developed as follows

$$\dfrac{\Gamma(\mathsf{doc}) = \{\ \mathsf{bib}\ \} \qquad \mathsf{d}_3, \Gamma \vdash_{\mathbf{(RES)}} \mathtt{doc/descendant} :: * : \Sigma_{\mathsf{S}}^{\mathsf{d}_3} \setminus \{\ \mathsf{bib}\ \}}{\mathsf{d}_3, \Gamma \vdash_{\mathbf{(SAC)}} \mathtt{doc/descendant} :: \mathit{book} : \Sigma_{\mathsf{S}}^{\mathsf{d}_3}}$$

The judgment infers the static access cover for `doc/descendant` $::$ *book*. The inference result is the whole set of types of d_3. This is obtained as a union of the root type (that is the context type for the step `descendant` $::$ $*$), together with all descendant types of the root (that type the result of the step `descendant` $::$ $*$). This last one is obtained by calling the **RES** system.

The first peculiarity of the system **SAC** we can observe is how the static access cover is computed. As said before, a sound static access cover would contain types for *book* nodes and their ancestors. To accomplish this, the navigational step `descendant` $::$ *book* is approximated by replacing the *book* label with the $*$ wildcard (denoting any node). This of course makes the analysis sound but imprecise,

because all schema types are inferred, and thus also all types unrelated with the step. As a consequence of this, the query is likely to be deemed dependent wrt almost all updates. We stress that this is always the case, for all queries starting with a descendant navigation.

② The judgment $\mathsf{d}_3, \Gamma \vdash_{\mathbf{(RES)}} \texttt{doc/descendant} :: \mathit{book} : \tau_2$ is developed as follows.

$$\frac{\Gamma(\texttt{doc}) = \{\mathsf{bib}\} \qquad \tau_2 = \{\ \mathsf{book} \mid \mathsf{book} \text{ is a descendant type of } \mathsf{bib} \text{ in } \mathsf{d}_3\ \}}{\mathsf{d}_3, \Gamma \vdash_{\mathbf{(RES)}} \texttt{doc/descendant} :: \mathit{book} : \{\ \mathsf{book}\ \}}$$

It infers types for the result of $\texttt{doc/descendant} :: \mathit{book}$, this time for providing input to the next inference step. The derivation stresses that schema information is used in order to check if type book is a descendant of type bib. If this is not the case then $\tau_2 = \varnothing$.

③ The judgment $\mathsf{d}_3, \Gamma[\texttt{x} \mapsto \tau_2] \vdash_{\mathbf{(SAC)}} \texttt{x}/\mathit{author} : \tau_3$ is developed in the following way

$$\frac{\mathsf{d}_3, \Gamma[\texttt{x} \mapsto \{\ \mathsf{book}\ \}] \vdash_{\mathbf{(RES)}} \texttt{x}/* : \{\ \mathsf{title}, \mathsf{author}, \mathsf{publisher}\ \}}{\mathsf{d}_3, \Gamma[\texttt{x} \mapsto \{\ \mathsf{book}\ \}] \vdash_{\mathbf{(SAC)}} \texttt{x}/\mathit{author} : \{\ \mathsf{book}, \mathsf{title}, \mathsf{author}, \mathsf{publisher}\ \}}$$

This infers the static access cover for $\texttt{x}/\mathit{author}$. The result is constituted by the union of types bound to variable x with the types of nodes in the result of step $\texttt{x}/*$.

Once again we observe that the child step is approximated, by replacing the label *author* with $*$. This means that all children types of book are inferred namely title, author and publisher. Once again, the typing is sound but is not precise since while type author is needed by step $\texttt{x}/\mathit{author}$, types title and publisher are not. Analogously to the previous case, the query is likely to be deemed as dependent wrt all updates impacting a node related to book.

Once types are inferred for the query, the update is analyzed.

The types of a node impacted by u are inferred by the system **IMP** as follows.

$$\frac{\mathsf{d}_3, \Gamma \vdash_{\mathbf{(RES)}} \texttt{doc}//\mathit{title} : \{\ \mathsf{title}\ \} \qquad \mathsf{d}_3, \Gamma[\texttt{x} \mapsto \{\ \mathsf{title}\ \}] \vdash_{\mathbf{(RES)}} \texttt{x/parent} :: * : \{\ \mathsf{book}\ \}}{\mathsf{d}_3, \Gamma \vdash_{\mathbf{(IMP)}} \texttt{delete doc}//\mathit{title} : \{\ \mathsf{book}\ \}}$$

The set of types deleted by the update is computed in two steps. First, system **RES** infer types for nodes effectively deleted, that are those returned by the target expression $\texttt{doc}//\mathit{price}$. Second, a parent navigation for such types is performed. This is the second peculiarity of the system, and it is meant to state that a node whose children are modified by the update is considered as impacted by the update as well. For the DTD d_3, this means that the data type impacted by the update is book. As a consequence of this, the

update is deemed dependent with all queries accessing book or one of its children. The same approximation is done for rename, insert and replace updates.

After type inference is done, for the analysis to be sound, all descendant types of nodes in the result of the query are also added to the set of types accessed by the query. For instance, because author types the root of a subtree output by q, all descendant types of author, namely name and String, are included in the result of the analysis.

Once type inference is done for q and u, providing sets of accessed and updated types τ_{q} and τ_{u}, static independence holds iff

$$\tau_{\mathsf{q}} \cap \tau_{\mathsf{u}} = \varnothing$$

Another peculiarity of this method is that by basing an analysis on type-names, even if type inference is precise, independence cannot be detected in many cases. In fact, when a type is *reused* in several definitions, it may be difficult, or even impossible, to disambiguate it.

To illustrate, consider again the DTD d_3, the environment Γ, and the following pair of query and update expressions.

$$\mathtt{q} = \mathtt{doc}//\mathit{author}/\mathit{name} \qquad \text{and} \qquad \mathtt{u} = \mathtt{delete\ doc}//\mathit{publisher}/\mathit{name}$$

Assume that the type name is inferred for both expressions. This type captures both nodes returned by the query and nodes impacted by the update, that are labeled with *name*. However, the sets of accessed and updated nodes here are *disjoint*, because the query accessed nodes labeled with *name* nested into nodes labeled with *author* while the update deletes nodes labeled with *name* nested into nodes labeled with *publisher*. [1] By using sets of type names, without recording any information on ancestor types, independence cannot be detected in many cases such as this one.

So far, we have seen a running example of the independence analysis based on types, and the use of inference system **RES**, **SAC** and **IMP**. Now we discuss how this technique supports updates that do not preserve the schema.

As said before, not assuming schema preservation is a nice property of the type-based analysis. However, when flows of updates (possibly violating the initial schema) are evaluated this property alone does not ensure the success of the analysis: one needs to maintain the schema after each update which does not preserve the schema [2]. Indeed, a flow of updates may change completely the structure of the schema thus making the static analysis unsound.

[1] Recall that if more than one node labeled with *publisher* is selected by the update path then the update has to be encoded by means of iteration, otherwise a run-time error is raised.

[2] A sound method for determining if an update preserves a schema and for maintaining the schema can be found in [BC09b]

To see this, consider the following update renaming all labels of the schema, provided the DTD $\mathtt{d}_3$ and the environment Γ.

$$\begin{aligned}\mathtt{u}_1 = \ & \mathtt{for\ x\ in\ doc}//\mathit{book}\\ & \mathtt{rename\ x\ as}\ \text{“}\mathit{livre}\text{”}\\ & \mathtt{rename\ x}/\mathit{title}\ \mathtt{as}\ \text{“}\mathit{titre}\text{”}\\ & \mathtt{rename\ x}/\mathit{author}\ \mathtt{as}\ \text{“}\mathit{auteur}\text{”}\\ & \mathtt{rename\ x}//\mathit{name}\ \mathtt{as}\ \text{“}\mathit{nom}\text{”}\end{aligned}$$

How does the analysis detect that a query like $\mathtt{q} = \mathtt{doc}//\mathit{livre}$, whose result is empty over any document valid wrt the DTD, is now drawing some data from this renaming? This case is covered since, while computing the static access cover of the query, system **SAC** approximates $\mathtt{doc}//\mathit{livre}$ with $\mathtt{doc}//*$. This makes the analysis sound, despite the fact that $\mathtt{u}_1$ does not preserve the schema. Indeed, in the intersection of type-names inferred from $\mathtt{q}$ and $\mathtt{u}_1$ is non-empty.

At this point, if the DTD $\mathtt{d}_3$ is not maintained, the analysis may turn to be unsound. Consider an update $\mathtt{u}_2$ possibly conflicting with $\mathtt{q}$ over the (invalid) document instance obtained after $\mathtt{u}_1$ such as

$$\mathtt{u}_2 = \mathtt{delete\ doc}//\mathit{livre}$$

Because no definition of *livre* is found in $\mathtt{d}_3$, the set of types inferred from $\mathtt{u}_2$ over $\mathtt{d}_3$ is empty, and thus independence of $\mathtt{q}$ and $\mathtt{u}_2$ is assumed, while it should not. Therefore, after an update breaking schema constraints the schema needs to be maintained if one aims at a sound analysis.

This short discussion shows that preservation of schema constraints is a problem orthogonal to that of independence. Schema evolution has been studied in [BC09b, CGM11] and it is out of the scope of our work.

We conclude this section by discussing simplicity and complexity of the type analysis of [BC09a] which are the most important features of the system. The approximations made by the type system discussed are motivated by avoiding a complex analysis and providing a simple inference system. The advantages of a simple type system are that it can be easily understood, implemented and proved correct. Proofs of the type systems have been provided and worst case complexity studied. The independence analysis runs in $O((|\mathtt{d}|^2 + |\mathtt{q}|)^2 + |\mathtt{u}|)$ and it turns out to be very fast in practice.

3.2.2 Commutativity analysis for XML updates [GRS08]

Another approach for detecting independence can be drawn from the commutativity analysis of XML updates proposed by Ghelli, Rose and Siméon in [GRS08]. This technique relies on an extension of the path-based analysis proposed in [MS03] that overapproximates the nodes accessed and modified by a given expression, by means of simple XPath expressions for which disjointness is decidable.

This work on commutativity distinguishes itself by two main aspects. First, it studies commutativity of updates, a different problem from that of independence. Second, and more importantly, this study considers XML languages with an iterative semantics, thus different from languages with a snapshot semantics like the one introduced in Section 2.3. As we discuss next, these two differences can be actually levelled out since commutativity allows to detect independence and the path-based approach suits to both kinds of semantics.

We discuss first the links between independence and commutativity. Roughly speaking, independence asks wether the result of a query is preserved after updating, while commutativity asks if the application order of two updates is irrelevant. Deciding commutativity allows to decide independence. Conceptually, this is true because updates are written in terms of queries. This implies that, an instance of the independence problem for `q`, `u` can be reduced to a commutativity one by rewriting `q` as a new update $\mathtt{u}' =$ `replace q with q`, and then check that $\mathtt{u}'$ commutes with `u`. This reduction also shows that checking commutativity is at least as hard as checking independence.

The commutativity analysis of [GRS08] has been developed for query languages with side-effects (i.e., updates) such as of XQuery! [GRS06] and XQueryP [CCF+06]. These languages have a so called *iterative* semantics, they support immediate update application and explicit left-to-right evaluation order. This is a main difference with the languages considered in [BC09a, BC10, BTCU12], that have a *snapshot* semantics. Iterative languages have a strict left to right evaluation order, while languages with snapshot semantics reorder their updates before application: insert and replace are executed first, then rename and finally delete. Iterative languages allow to see the effects of updates during evaluation.

While iterative languages seem more intuitive, they have a more involved semantics. This is because any expression can read data that it previously inserted. To illustrate, a (possibly) non terminating update such as the following can easily be written in an iterative language.

$$\texttt{for x in doc//}name\ \texttt{return}\ \ \texttt{insert}\ \texttt{<}name\texttt{/>}\ after\ \texttt{x}$$

Assuming that the base data contains at least one node labeled with *name* (that is also a descendant of the node bound to `doc`), the expression keeps on extending the iterating sequence of *name* nodes forever.

Much more machinery effort has to be done in order to prove the soundness of the commutativity analysis for a language with iterative semantics, with respect to languages with snapshot semantics. Many expressions that do not commute in iterative languages do commute in snapshot language (simply because they do not have the same effect), so an analysis for an iterative language is likely to work also for a language with snapshot semantics. Of course, it is possible to define a more general (and less precise) static analysis that works for both kinds of languages.

We proceed now with the description of the path-based analysis, thus illustrating its characteristics.

The path-based analysis relies on a system of *simple path* inference. Simple paths are expressions belonging to the fragment of XPath composed by downward and upward navigation, plus the wildcard *. Predicates, as well as data values and sibling axis, are not allowed.

The analysis relies on an inference system **(PATH)** for inferring the simple paths of an expression. The judgment $\Gamma \vdash_{\textbf{(PATH)}} \mathtt{q} : (\mathbf{rt}, \mathbf{ac}, \mathbf{up})$ is a relation between the static variable environment Γ, the query $\mathtt{q}$ and the sets of paths $(\mathbf{rt}, \mathbf{ac}, \mathbf{up})$ where $\mathbf{rt}$ is the set of path leading to returned subtrees of $\mathtt{q}$, $\mathbf{ac}$ is the set of path representing nodes accessed but possibly not returned by $\mathtt{q}$, and $\mathbf{up}$ is the set of path representing nodes that are updated by $\mathtt{q}$.

Distinguishing among these three kinds of paths allows to capture two main aspects of the language. First, side-effects, by distinguishing between update paths from the rest. Second, query semantics, by distinguishing between accessed and return paths.

Note that here the analysis is supported by a single inference system, and most importantly, the analysis does not depend on an input schema. In the following examples the steps navigating from the root of a tree are written without an explicit variable.

To illustrate the (**PATH**) inference system, we consider again the following query-update pair.

$\mathtt{q} = \mathtt{for\ x\ in\ /descendant} :: \mathit{book}\ \mathtt{return\ x}/\mathit{author} \qquad \mathtt{u} = \mathtt{delete}\ //\mathit{price}$

Inference of return, used and update paths for $\mathtt{q}$ is derived in the following way.

$$\dfrac{\dfrac{\textcircled{1}}{\Gamma \vdash_{\textbf{(PATH)}} \mathtt{/descendant} :: \mathit{book} : (\mathbf{rt}, \mathbf{ac}, \mathbf{up})} \qquad \dfrac{\textcircled{2}}{\Gamma[\mathtt{x} \mapsto \mathbf{rt}] \vdash_{\textbf{(PATH)}} \mathtt{x}/\mathit{author} : (\mathbf{rt}', \mathbf{ac}', \mathbf{up}')}}{\Gamma \vdash_{\textbf{(PATH)}} \mathtt{for\ x\ in\ /descendant} :: \mathit{book}\ \mathtt{return\ x}/\mathit{author} : (\mathbf{rt}',\ \mathbf{ac} \cup \mathbf{ac}',\ \mathbf{up} \cup \mathbf{up}')}$$

Accessed and updated paths inferred for the whole expression are those inferred from each subexpression only. Return paths for the whole expression are naturally those on the right subexpression. Next we will see how the static environment Γ helps in reconstructing navigational paths by following variable bindings.

Notice that the rule for path extraction introduces a bit of imprecision since all accessed and updated paths of the left subexpressions are kept in the final result, independently of the right subexpression. This means that, for instance, when the right branch of the iteration query performs no operation, and thus the query does nothing,

the set of inferred paths for the whole expression may *not* be empty, while it should. This is the case for the following query

$$\texttt{for x in } //\textit{book} \texttt{ return ()}$$

for which path $//book$ is inferred, despite the fact that inferring an empty set of returned and accessed path would still be sound.

Let us explore further derivations and comment on other peculiarities of the analysis.

① The judgment $\Gamma \vdash_{\textbf{(PATH)}} \texttt{/descendant} :: \textit{book} : (\mathbf{rt}, \mathbf{ac}, \mathbf{up})$ is developed as follows

$$\frac{\mathbf{rt} = \{\texttt{/descendant} :: \textit{book}\}}{\Gamma \vdash_{\textbf{(PATH)}} \texttt{/descendant} :: \textit{book} : (\mathbf{rt}, \mathbf{rt}, \varnothing)}$$

In this case path inference is a simple path extraction. Notice that return and accessed paths coincide, and of course no update path is inferred.

② The judgment $\Gamma[\texttt{x} \mapsto \mathbf{rt}] \vdash_{\textbf{(PATH)}} \texttt{x}/\textit{author} : (\mathbf{rt}', \mathbf{ac}', \mathbf{up}')$ is developed as follows

$$\frac{\Gamma[\texttt{x} \mapsto \mathbf{rt}] \vdash_{\textbf{(PATH)}} \texttt{x} : (\mathbf{rt}, \varnothing, \varnothing) \qquad \mathbf{rt}' = \mathbf{rt} \times \{/\textit{author}\} \qquad \mathbf{ac}' = \textit{prefixes}(\mathbf{rt}')}{\Gamma[\texttt{x} \mapsto \mathbf{rt}] \vdash_{\textbf{(PATH)}} \texttt{x}/\textit{author} : (\mathbf{rt}', \mathbf{ac}', \varnothing)}$$

Here the system first reconstructs simple paths by following variable bindings. Indeed, the path $\texttt{/descendant} :: \textit{book}$ that is bound to $\texttt{x}$ is combined in a cartesian product with step $/\textit{author}$ in order to infer $\mathbf{rt}' = \{\ \texttt{/descendant} :: \textit{book}/\textit{author}\ \}$. Accessed paths are defined as prefixes of inferred paths. Of course, no update path is inferred since the operation has no side-effect.

The application of the rule may lead to infer exponentially many paths for some rare expressions such as the following one.

$$\texttt{for}\ \ \texttt{x}_1\ \texttt{in}\ (\texttt{x}_0/a\,,\ \texttt{x}_0/b)\ \ \ldots\ \ \texttt{x}_n\ \texttt{in}\ (\texttt{x}_{n-1}/a\,,\ \texttt{x}_{n-1}/b)\ \ \texttt{return}\ \texttt{x}_n$$

this expression would produce exactly 2^n return paths and $O(n2^n)$ accessed paths. This kind of expressions are however very rare in practice.

Now that paths are inferred from the query, we focus on the update. Path inference for the update expression is derived as follows

$$\frac{\Gamma \vdash_{\textbf{(PATH)}} //\textit{price} : (\{\ //\textit{price}\ \}, \varnothing, \varnothing) \qquad \mathbf{ac} = \{\ //\textit{price}\ \} \qquad \mathbf{up} = \{\ //\textit{price}\ \} \times \{\ \texttt{/descendant-or-self} :: *\ \}}{\Gamma \vdash_{\textbf{(PATH)}} \texttt{delete}\ //\textit{price} : (\varnothing, \mathbf{ac}, \mathbf{up})}$$

Here nodes deleted by the expression are captured by combining return paths of *//price* with a `descendant-or-self` :: $*$ step, denoting that whole subtrees rooted at prices are deleted. Return paths for the target query become also accessed paths for the whole expression. No return path is inferred for the whole update.

Notice that the path analysis precisely capture only the deleted subtrees without including their parent, as is done by the type analysis.

Another characteristics of the analysis is how updates paths are extracted from insert and replace commands. To illustrate, consider the following update.

`insert` *//author* `into` */site/book*

Paths for the update are extracted in the following way.

$$\frac{\Gamma \vdash_{\textbf{(PATH)}} \textit{//author} : (\mathbf{rt}_1, \mathbf{ac}_1, \mathbf{up}_1) \qquad \Gamma \vdash_{\textbf{(PATH)}} \textit{/site/book} : (\mathbf{rt}_2, \mathbf{ac}_2, \mathbf{up}_2) \quad \mathbf{ac} = \mathbf{rt}_1 \times \{\texttt{/descendant-or-self} :: *\} \quad \mathbf{up} = \mathbf{rt}_2/ \times \{\texttt{descendant} :: *\}}{\Gamma \vdash_{\textbf{(PATH)}} \texttt{insert } \textit{//author } \texttt{into } \textit{/site/book} : (\varnothing\,,\, \mathbf{ac}_1 \cup \mathbf{ac}_2 \cup \mathbf{ac}\,,\, \mathbf{up}_1 \cup \mathbf{up}_2 \cup \mathbf{up})}$$

Accessed paths extracted from the two subexpressions become accessed paths for the whole update. Moreover, return paths for the source expression *//author* are combined with a self or descendants axis, and become accessed paths for the whole update, mimicking the fact that an entire subtree is copied.

Update paths extracted from the subexpressions (if any) become update paths for the whole update. Notice that the set of *inserted* paths is inferred by combining path */site/book* with a descendant step, mimicking that the whole subtree rooted at book is updated. This of course is sound, but not complete. For instance, the update is deemed as non-commuting with query */site/book/title*, while it does.

Once return, accessed and update paths are inferred from the expressions, an intersection test is run. The subset of XPath considered for inference includes downward and backward navigational axes, and * wildcard. Branching is excluded. For this language intersection is decidable. While results in [Hid03] show that the intersection problem for downward XPath expressions is NP-complete, [HKL05] approach shows that intersection is solvable in quadratic time for the downward fragment only.

One of the most important features of the path analysis, is that it works independently without requiring a schema. However, in all the cases where schema is available, schema constraints cannot be easily plugged into the path-based analysis. Soundness of the path-based commutativity analysis has been proved in [GRS08], and also its benefits have been showed when used for optimizing logical plans of queries with side-effects [GORS08].

3.2.3 Destabilizers and independence of XML updates [BC10]

Destabilizers [BC10] are a generalization of the idea of statically approximating the set of accessed nodes found in some prior work [BCCN06, MS03, GRS08] and then testing intersection, combined with the concrete use of the data. The destabilizer framework can work in principle for any language and data-model, in [BC10] they are formally implemented for XML queries and updates.

The formal definition of destabilizers is quite technical, thus here we provide just a gentle introduction to the approach, discuss the problematics in computing destabilizers and outline how they are concretely used.

Given a query $\mathtt{q}$, the destabilizer of $\mathtt{q}$, denoted by $\Delta(\mathtt{q})$, is defined as the set of updates that may change the answer of $\mathtt{q}$. From this follows that an update $\mathtt{u}$ is independent wrt a query $\mathtt{q}$ when $\mathtt{u}$ does not overlap with $\Delta(\mathtt{q})$. In the following we will distinguish between *dynamic* or *static*, and *exact* or *approximate* destabilizers. Dynamic destabilizers are composed by atomic update commands, while static ones are update expressions. Exact destabilizers are *minimal*, in the sense that they produce no false negative (updates that do not change the answer of $\mathtt{q}$), while approximated ones may produce false negatives.

An exact dynamic destabilizer is a minimal set of atomic update commands that can affect the result of a query $\mathtt{q}$ evaluated over the tree t. We recall that an atomic update command is an operation of

$$\mathtt{del}(\_) \quad \mathtt{ins}(\_,\_,\_) \quad \mathtt{ren}(\_,\_) \quad \mathtt{repl}(\_,\_)$$

As an example, consider the document $t = (\sigma, l_t)$ with two nodes

$$\sigma = \begin{cases} l_t & \leftarrow \quad \mathtt{bib}[l_1] \\ l_1 & \leftarrow \quad \mathtt{book}[\,] \end{cases}$$

Let $\mathtt{q}$ be the query */bib/book*. The exact dynamic destabilizer of $\mathtt{q}$ wrt t, denoted by $\Delta^d(\mathtt{q}, t)$, is composed by all atomic update commands targeting either l_t or l_1 in t. The atomic updates that can alter l_t are the followings

$$\mathtt{del}(l_t) \quad \mathtt{ren}(l_t, \text{“}a\text{”}) \quad \mathtt{repl}(l_t, l')$$

where l' is a location not isomorphic to l_t, and a is a string different from *bib*.
The atomic updates that can alter l_1 are the followings.

$$\mathtt{del}(l_1) \quad \mathtt{ins}(L_i, \mathtt{pos}, l_t) \quad \mathtt{ins}(L_i, \mathtt{pos}', l_1) \quad \mathtt{ren}(l_1, \text{“}b\text{”}) \quad \mathtt{repl}(l_1, L_i)$$

where L_i is a non empty sequence of locations containing a node not labeled with *book*, $\mathtt{pos}$ is one of $\{\mathtt{into}, \mathtt{into\ as\ first}, \mathtt{into\ as\ last}\}$, $\mathtt{pos}'$ is one of $\{\mathtt{before}, \mathtt{after}\}$, and b is a string different from *book*.

An update u is independent of a query q if the UPL produced by u over t does not overlap with $\Delta^d(\mathtt{q}, t)$. Consider the update

$$\mathtt{u}_1 = \mathtt{insert}\ \ < new/ >\ \ \mathtt{into}\ \ /bib$$

$\mathtt{u}_1$ over t generates the atomic update $\mathtt{ins}(l_{new}, \mathtt{into}, l_t)$. Because l_{new} is not labeled with *book*, q is independent of $\mathtt{u}_1$. Differently, the update

$$\mathtt{u}_2 = \mathtt{delete}\ \ /bib/*$$

is not independent with q because it generates the atomic update $\mathtt{del}(l_1)$ which also belongs to $\Delta^d(\mathtt{q}, t)$.

Benedikt and Cheney [BC10] illustrate that it is not feasible to deal with dynamic destabilizers as concrete sets of atomic update sequences. Therefore, destabilizers are treated at a symbolic level. The set of nodes involved in atomic updates in $\Delta^d(\mathtt{q}, t)$ is approximated by any update expression targeting those locations. Since updates are defined in terms of queries, it turns out that the destabilizer set is defined as a set of *query* expression targeting all nodes that are arguments of the atomic updates. For q = */bib/book*, the *exact static* destabilizer $\Delta^s(\mathtt{q}, t)$ approximating the dynamic one, is simply

$$\Delta^s(\mathtt{q}, t) = \{\ /bib,\ /bib/book\ \}$$

In developing a static destabilizer, there is a trade-off between the precision (how close it is to the dynamic analog) and the complexity of analyzing the resulting query. The generic static destabilizer defined as above is frequently a significant over-approximation of the runtime destabilizer. For example, a query may be sensitive to update sequences that can only affect a node via a particular update operation. In its simplest form, the static destabilizer cannot capture this aspect. The authors presented a query-rewriting technique that provides a useful, sound approximation. Destabilizers sensitive to update operations (precise for each update), as well as destabilizers sensitive to different equalities (precise for boolean, node and subtree equivalence) are also defined in [BC10].

Benedikt and Cheney showed that calculating an *exact* static destabilizers is not feasible even for a core XML query language, hence the analysis needs to abstract away from some features in order to be effectively executed. This is stated by the following theorem.

Theorem 3.2.1 ([BC10])**.** *There is no elementary time algorithm for constructing an operation-sensitive minimal static destabilizer.*

The formal definition of a destabilizer is reminiscent of a path analysis. The main difference with the path-based approach is that destabilizers also consider the source document for the overlapping test. To illustrate, consider the following query q and update u over the previous document t.

$$\mathtt{q} = \mathtt{for\ x\ in}\ /bib\ \mathtt{return}\ <list>\mathtt{x}/book</list> \qquad \mathtt{u} = \mathtt{delete}\ //book/title$$

The destabilizer for the query is

$$\Delta^s(\mathtt{q}, t) = \{\ /bib\,,\ /bib/book/\mathtt{descendant\text{-}or\text{-}self} :: *\ \}$$

Now, while running the intersection test between u and $\Delta^s(\mathtt{q}, t)$, we consider also the actual data of t.

The intersection tests of the destabilizers approach is more accurate that the one of previous techiques. While the path analysis (and also the type analysis) would exclude independence, the destabilizer analysis correctly deem the independence, because there is no data in the store corresponding to *title* nodes. In fact, the path analysis (and also the type analysis) does not take into account the document. This is because the path analysis would see that the path targeting delete nodes //*book*/*title*/descendant-or-self :: * intersects with path /*bib*/*book*/descendant-or-self :: * (inferred for q).

The source document is taken into account by translating queries into logical formulas, and then transforming the intersection test of query and update paths to a satisfiability problem, solved with SAT-solvers. Several exact and approximate translation are proposed. These allow to trade between precision and efficiency of the analysis. In the following, we limit ourselves to discuss translations, without reporting the formal development, that can be found in [BC10].

We discuss now exact solutions. Queries without element construction can be translated to first-order formulas over the child, descendant and sibling relations, and then disjointness analysis for these queries reduces to satisfiability for first-order formulas over trees FO(Trees). However it follows from results of [Vor96] that there is a non-elementary lower bound on the complexity of satisfiability. The solver used for resolution of this formulas is Yices [DM06]. Another strategy is to convert expression into monadic second order logic (MSO). It follows from results in [SM73] that there is a non-elementary lower bound on the complexity of satisfiability of MSO over trees. The solver used for MSO resolution is MONA [KM11].

We discuss approximate solutions. To check intersection of query expressions on database instances, the authors propose first a reduction to existential second order logic (ESO) which uses as support an arithmetic encoding of the input document. This solution has an overhead that depends on the document to analyze, and this could not be reasonable for large documents. We recall that path intersection is NP-hard in general, even if for downward-only paths, overlapping is exactly decidable in quadratic time.

In [JGL12] Junedi, Genevès and Layïda showed that the efficiency of destabilizer can be enhanced wrt the work of [BC10] by checking satisfiability of logical formulas using their μ-solver [GLS07, GL08]. Their approach is based on translating XPath expressions into μ-calculus formulas. As outlined above, the theoretical complexity of satisfiability on MSO formulas is known to be non-elementary, whereas it is exponential for simplex

based linear arithmetic satisfiability. The complexity of satisfiability on μ-formulas is simple exponential. The resulting approach resulted to be far more competitive wrt solver such as MONA and Yices. An evidence from experiments in [JGL12] is that the dominant inference cost does not necessarily reside in the resolution procedure, but mostly in constructing destabilizers, parsing formulas and initializing the auxiliary libraries for resolution. The time spent in the resolution procedure better reflects the intrinsic difficulty of each problem instance.

In conclusion, the merit of destabilizers is twofold. It is a modular framework that generalizes concepts of all previous approaches to independence. It promotes the practical use of SAT-solvers for intersection analysis. The approach does not require a schema, even if they can be handled by adding schema constraints in the satisfiability test. While former experiments in [BC10] showed some limits of the approach concerning the cost of the destabilizer analysis, recent works of [JGL12] showed that destabilizers can be implemented efficiently.

Conclusions

In this chapter we presented the theoretical background and the state-of-the-art techniques for checking XML query update independence. We discussed theoretical results that show the computational difficulty of the problem. Then we discussed three approximated analysis that, being sound but not complete, try to make the analysis tractable. All approaches distinguish for simplicity, precision and complexity. The type-based analysis [BC09a] requires a schema in order to be executed, it is simple to implement and it runs in polynomial time, but it may be not precise. The path-based analysis [GRS08] works without requiring a schema but when structural constraints are available they cannot be easily integrated. It is also simple to implement but for some expression the path extraction may occur in blowups, and also exact checking of path disjunction turns out to be NP-hard. The destabilizer analysis [BC10] is a modular framework that generalizes all previous ideas, and allow to perform an independence analysis varying precision and complexity of the approach. The method works without schemas, but it can easily integrate structural constraints.

In the next chapters we present our technique for detecting independence based on schemas, showing how it can ensure a precise analysis while still keeping inference polynomial in time and space.

Chapter 4

Independence Analysis based on Schema-Chains

This chapter presents the main contribution of this thesis: a novel method for detecting XML query update independence, in the presence of a schema. We present an inference system for computing the set of navigational paths accessed by queries and updates, called *chains*. We prove the chain-based inference system to be sound and more precise than the related approach based on types. We present a static notion of independence, based on chains, that we show to be decidable, even for recursive schemas which can generate infinite sets of chains.

The chapter is organized in four parts. In the first part, we present a system for chain inference. In the second part, we show the correctness of the type system. In the third part, we present a static notion of independence based on chains. In the fourth part, we show the decidability of the notion of static independence.

4.1 Introduction

As explained in the previous chapter, the type analysis for checking independence defined in [BC09a] is designed so as to infer information about accessed data, by abstracting away from some of the structural properties expressed by queries and updates. This is done in order to keep the systems relatively simple, and to make the inference fast. Actually, the main structural information which is traded is data-nesting, that defines the way in which types interact.

Data-nesting is a synonym of hierarchical information. Together with document order, it entails that the semantics of a subtree depends on its position in the document. Describing a collection of accessed and updated trees with a set of types-names, is equivalent to consider all possible trees with fixed tag alphabet, ignoring any constraint on the hierarchy of data. Unfortunately, in many cases this causes lack of precision because of the tree shape of XML data.

The goal of this work is to improve the precision of the method proposed in [BC09a] by exploiting schema information in a different way. Rather than inferring sets of type-names accessed by the expressions, our goal is to infer *sequences* of accessed types, called *chains*, that essentially correspond to navigational paths traversed by query and update expressions on valid schema instances. Chains will form the basis of an extremely precise independence analysis.

We present now formal definitions about chains. We begin by defining the notion of *type* and *chain* of types for a *node location* belonging to a store.

Definition 4.1.1 (Node Type and Chain)**.** *Given a store σ, the type of a location $l \in dom(\sigma)$, denoted by $\mathsf{type}(l)$ is defined as*

$$\mathsf{type}(l) = \begin{cases} \mathsf{a} & \text{if } \sigma(l) = \mathsf{a}[L] \\ \mathsf{String} & \text{otherwise} \end{cases}$$

The chain associated to location l, denoted by c_l^σ, is defined by

$$\mathsf{c}_l^\sigma = \begin{cases} \mathsf{type}(l) & \text{if } l \text{ has no parent} \\ \mathsf{c}^\sigma_{parent(l)}.\mathsf{type}(l) & \text{otherwise} \end{cases}$$

If $t = (\sigma, l_t)$ and $l \in dom(t)$ then we denote by c_l^t the chain $\mathsf{c}_l^t = \mathsf{c}_l^\sigma$. To illustrate, consider the tree t_2 in Figure 4.1. Here the type of the root location l_{t_2} is $\mathsf{type}(l_{t_2}) = \mathsf{bib}$, and it also coincides with its own chain $\mathsf{c}^\sigma_{l_{t_2}}$, because the root node has no parent. The type of l_3' and l_4' are the same, indeed $\mathsf{type}(l_3') = \mathsf{type}(l_4') = \mathtt{name}$. However their chains differ since

$$\mathsf{c}^\sigma_{l_3'} = \mathtt{bib.book.author.name} \qquad \mathsf{c}^\sigma_{l_4'} = \mathtt{bib.book.publisher.name}$$

Notice that since the store coincides with the tree, we have that $\mathsf{c}_l^\sigma = \mathsf{c}_l^{t_2}$ for all $l \in dom(\sigma)$. As a convention in the notation, we use "." to separate two consecutive labels in a chain. Notice also that this definition considers only *rooted* chains, where the starting symbol is always the root.

We stress that the above definition works for any XML tree and node location, independently from schema constraints. However, dealing with schemas is our ultimate goal. Hence in order to develop a static analysis which is correct we need to ensure that, if a document is compliant with a schema, then all node chains in the document are induced by the schema.

To formalize this property, we define the set of chains induced by a schema relying on a reachability relation between types.

Definition 4.1.2 (Reachability and Chains)**.** *Given a DTD d, we denote by $\mathsf{a} \Rightarrow_\mathsf{d} \mathsf{b}$ the reachability relation that holds for a pair $\mathsf{a}, \mathsf{b} \in \Sigma_\mathsf{S}$ if b occurs in the regular expression $\mathsf{d}(\mathsf{a})$. A chain c over d is a non-empty sequence of labels*

$$\mathsf{a}_1.\mathsf{a}_2 \ldots .\mathsf{a}_n$$

such that $\mathsf{a}_i \Rightarrow_\mathsf{d} \mathsf{a}_{(i+1)}$ for $i = 1 \ldots n-1$.

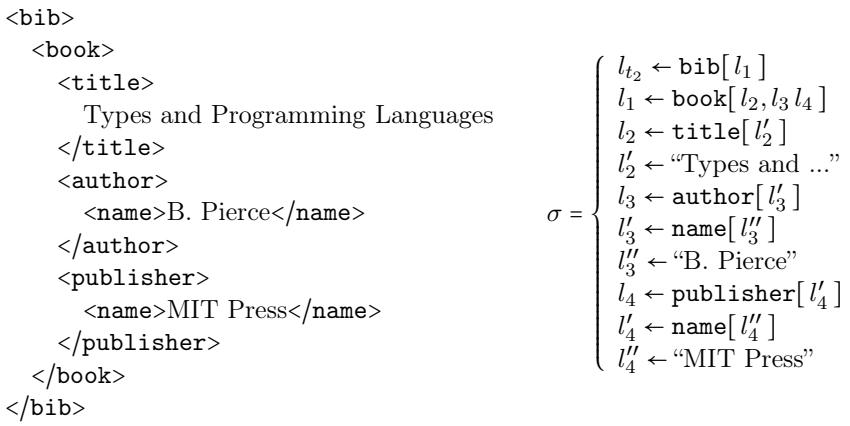


Figure 4.1: Document t_2 and store (σ, l_{t_2})

The set of chains associated with the DTD d is denoted by $\mathsf{C_d}$. Observe that chains in $\mathsf{C_d}$ are of finite length and may start with any DTD symbol. The set $\mathsf{C_d}$ is infinite only if d is a vertical-recursive schema (i.e., some type is defined in terms of the type itself). The empty chain is denoted by ϵ, and we assume, as a convention, that it is never induced by any schema. Given two chains c_1 and c_2, the concatenation of c_1 and c_2 is denoted $\mathsf{c}_1.\mathsf{c}_2$, of course $\mathsf{c}_1.\epsilon = \mathsf{c}_1$. We write $\mathsf{c}_1 \preceq \mathsf{c}_2$ to indicate that c_1 is a prefix of c_2, that is $\mathsf{c}_2 = \mathsf{c}_1.\mathsf{c}$ for some chain c.

As an example of schema chains, consider the DTD d_3 in Figure 3.1. The set $\mathsf{C_{d_3}}$ includes the chains

bib.book.title.String bib.book.author.name.String bib.book.publisher.name.String

and all their non-empty subsequences. For instance, bib and author.name.String are $\mathsf{C_{d_3}}$-chains, while bib.author and book.String are not.

Relation $\Rightarrow_{\mathsf{d}}$ is both sound and complete. Soundness means that for any pair of parent-child locations belonging to a valid tree, there exists a correspondent pair of types in $\Rightarrow_{\mathsf{d}}$. Completeness means that the relation is also minimal, thus removing one pair from $\Rightarrow_{\mathsf{d}}$ we beak soundness. This is stated by the following lemmas. Proofs are reported in Chapter 9.

Lemma 4.1.3 (Soundness of $\Rightarrow_{\mathsf{d}}$)**.** *Let* d *be a DTD and* $t = (\sigma, l_t) \in \mathsf{d}$ *a valid tree. If* $(l, l') \in \mathit{Child}_t$ *then* $\mathsf{type}(l) \Rightarrow_{\mathsf{d}} \mathsf{type}(l')$*.*

Lemma 4.1.4 (Completeness of $\Rightarrow_{\mathsf{d}}$)**.** *For all DTD* d*, if* $\mathsf{a} \Rightarrow_{\mathsf{d}} \mathsf{b}$ *then there exists a valid tree* $t = (\sigma, l_t) \in \mathsf{d}$*, and a pair of locations* $(l, l') \in \mathit{Child}_t$*, such that* $\mathsf{type}(l) = \mathsf{a}$ *and* $\mathsf{type}(l') = \mathsf{b}$*.*

At this point, we can reconcile chains associated to node locations (Definition 4.1.1) and chains induced by a schema (Definition 4.1.2) by stating that, when we consider a document valid against a schema, all node-chains *are* schema-chains.

Proposition 4.1.5. *Given a valid tree* $t \in \mathsf{d}$ *and a location* $l \in dom(t)$ *we have* $\mathsf{c}_l^t \in \mathsf{C}_\mathsf{d}$.

Proof. Immediate by Lemma 4.1.3. □

The converse of this proposition is not true. Not all of the schema-chains are node-chains of a tree, for *all* valid schema instance. This is because some elements may be optional. Therefore, what can be proved is that for all *rooted* schema chain there *exists* a document instance including a node with that chain. This, as a corollary of Lemma 4.1.4.

At this point, we formally defined types and chains for documents and schemas. We have shown that schema chains can be safely used in order to abstract over tree chains (Proposition 4.1.5), as a corollary of validity. The sequel of the chapter is dedicated to define an independence analysis based on schema-chains.

4.2 Chain Inference

We begin by defining chain inference for XPath navigational steps, we then generalize to queries and updates.

4.2.1 Step Chain Inference

The definition of our chain inference system makes the assumption that the inference is made starting from an input set of chains C. Given a DTD d, this set can be either C_d (possibly infinite analysis) or a finite subset of C_d (finite analysis). We would like to stress that assuming a pre-computed chain set is only made to ease the formal presentation. Any reasonable implementation can avoid this, by inferring chains in C on the fly (see Chapter 5).

The first ingredient for query/update chain inference is chain inference for a single XPath step. We first define chain inference for axes, and then for node tests. Axis chain inference aims at inferring all chains that can be generated by axis navigation, in a d instance, starting from a node typed by a chain $\mathsf{c} \in \mathsf{C}$. Chain inference for XPath axes is defined below, where we assume that $\mathsf{c}', \mathsf{c}'' \neq \epsilon$.

$$
\begin{aligned}
\mathbf{A}_\mathsf{C}(\mathsf{c}, \texttt{self}) &\stackrel{def}{=} \{\, \mathsf{c} \,\} \\
\mathbf{A}_\mathsf{C}(\mathsf{c}, \texttt{child}) &\stackrel{def}{=} \{\, \mathsf{c.a} \mid \mathsf{c.a} \in \mathsf{C} \,\} \\
\mathbf{A}_\mathsf{C}(\mathsf{c}, \texttt{descendant}) &\stackrel{def}{=} \{\, \mathsf{c.c'} \mid \mathsf{c.c'} \in \mathsf{C} \,\} \\
\mathbf{A}_\mathsf{C}(\mathsf{c}, \texttt{parent}) &\stackrel{def}{=} \{\, \mathsf{c'} \mid \mathsf{c} = \mathsf{c'.a} \,\} \\
\mathbf{A}_\mathsf{C}(\mathsf{c}, \texttt{ancestor}) &\stackrel{def}{=} \{\, \mathsf{c'} \mid \mathsf{c} = \mathsf{c'.c''} \,\}
\end{aligned}
$$

Chain inference rules strictly mimic XPath semantics of axes. It is worth noticing that all navigations follows from the reachability relation $\Rightarrow_{\mathsf{d}}$. Notice also that

$$\mathbf{A}_{\mathsf{C}}(\mathsf{c}, \mathtt{axis\text{-}or\text{-}self}) = \mathbf{A}_{\mathsf{C}}(\mathsf{c}, \mathtt{axis}) \cup \mathbf{A}_{\mathsf{C}}(\mathsf{c}, \mathtt{self})$$

for $\mathtt{axis} \in \{\mathtt{descendant}, \mathtt{ancestor}\}$.

We illustrate step chain inference over the DTD d_3 in Figure 3.1, with the following examples.

$$\begin{aligned}
\mathbf{A}_{\mathsf{C}_{\mathsf{d}_3}}(\mathsf{bib.book}, \mathtt{child}) &= \{\ \mathsf{bib.book.title} \\
&\quad\ \mathsf{bib.book.author} \\
&\quad\ \mathsf{bib.book.publisher}\ \} \\
\mathbf{A}_{\mathsf{C}_{\mathsf{d}_3}}(\mathsf{bib.book.author}, \mathtt{descendant}) &= \{\ \mathsf{bib.book.author.name} \\
&\quad\ \mathsf{bib.book.author.name.String}\ \} \\
\mathbf{A}_{\mathsf{C}_{\mathsf{d}_3}}(\mathsf{bib.book.author}, \mathtt{parent}) &= \{\ \mathsf{bib.book}\ \}
\end{aligned}$$

Chain inference for sibling axes has a slightly more involved definition. First we need to define a relation between sibling types. This is needed in order to deal with horizontal axis.

Definition 4.2.1 (Sibling-type relation)**.** *Two types* a, b *are said to be siblings wrt a regular expression* r*, denoted by* $\mathsf{a} <_{\mathsf{r}} \mathsf{b}$ *if there exists a word* u *belonging to the language generated by* r *in which a letter* a *occurs before a letter* b*. The relation* $<_{\mathsf{r}}$ *is inductively defined as follows*

$$\begin{aligned}
<_{()} &\stackrel{def}{=} \varnothing \\
<_{\mathsf{a}} &\stackrel{def}{=} \varnothing \\
<_{\mathsf{r}_1|\mathsf{r}_2} &\stackrel{def}{=} <_{\mathsf{r}_1} \cup <_{\mathsf{r}_2} \\
<_{\mathsf{r}_1,\mathsf{r}_2} &\stackrel{def}{=} <_{\mathsf{r}_1} \cup <_{\mathsf{r}_2} \cup\ \mathsf{Sym}(\mathsf{r}_1) \times \mathsf{Sym}(\mathsf{r}_2) \\
<_{\mathsf{r}_1^+} &\stackrel{def}{=} <_{\mathsf{r}_1,\mathsf{r}_1} \\
<_{\mathsf{r}_1^*} &\stackrel{def}{=} <_{\mathsf{r}_1,\mathsf{r}_1}
\end{aligned}$$

where $\mathsf{Sym}(\mathsf{r})$ *is the set of symbols used in the regular expression* r*.*

In the following rules, with a little abuse of notation, given a chain c on d, we use $\mathsf{d}(\mathsf{c})$ to indicate either the regular expression $\mathsf{d}(\mathsf{a})$, when $\mathsf{c} = \mathsf{c}'.\mathsf{a}$, or the empty regular expression ϵ, when $\mathsf{c} = \mathsf{c}'.\mathsf{String}$. Chain inference for preceding/following-sibling axes is defined as follows.

$$\begin{aligned}
\mathbf{A}_{\mathsf{C}}(\mathsf{c}, \mathtt{following\text{-}sibling}) &\stackrel{def}{=} \{\ \mathsf{c}'.\mathsf{b} \in \mathsf{C} \mid \mathsf{c} = \mathsf{c}'.\mathsf{a},\ \ \mathsf{a} <_{\mathsf{d}(\mathsf{c}')} \mathsf{b}\ \} \\
\mathbf{A}_{\mathsf{C}}(\mathsf{c}, \mathtt{preceding\text{-}sibling}) &\stackrel{def}{=} \{\ \mathsf{c}'.\mathsf{a} \in \mathsf{C} \mid \mathsf{c} = \mathsf{c}'.\mathsf{b},\ \ \mathsf{a} <_{\mathsf{d}(\mathsf{c}')} \mathsf{b}\ \}
\end{aligned}$$

It is worth noticing that this sibling typing is very precise, since the sibling relation is evaluated for a particular regular expression. To illustrate, let $\mathsf{r}_1, \mathsf{r}_2$ be two regular expressions in d. It may be that $\mathsf{a} <_{\mathsf{r}_1} \mathsf{b}$ but $\mathsf{a} \not<_{\mathsf{r}_2} \mathsf{b}$, and this difference can be captured using chains. This is not the case for the type analysis in [BC09a] that would consider a and b as sibling types in all cases.

Step chain inference for sibling axes over DTD d_3 is exemplified below.

$$\begin{aligned}
\mathbf{A}_{\mathsf{C}_{\mathsf{d}_3}}(\mathsf{bib.book.title}, \texttt{following-sibling}) &= \{\ \mathsf{bib.book.author} \\
&\quad\ \mathsf{bib.book.publisher}\ \} \\
\mathbf{A}_{\mathsf{C}_{\mathsf{d}_3}}(\mathsf{bib.book.title}, \texttt{preceding-sibling}) &= \varnothing
\end{aligned}$$

As a more involved example of the sibling relation $<_{\mathsf{r}}$ consider for instance

$$<_{\mathsf{a},(\mathsf{b}\,|\,\mathsf{f})+,\mathsf{g}} = \{(\mathsf{b},\mathsf{b}),\ (\mathsf{b},\mathsf{f}),\ (\mathsf{f},\mathsf{b}),\ (\mathsf{f},\mathsf{f}),\ (\mathsf{a},\mathsf{b}),\ (\mathsf{a},\mathsf{f}),\ (\mathsf{a},\mathsf{g}),\ (\mathsf{b},\mathsf{g}),\ (\mathsf{f},\mathsf{g})\}$$

since

$$\begin{aligned}
<_{\mathsf{a},(\mathsf{b}\,|\,\mathsf{f})+,\mathsf{g}} &= <_{\mathsf{a},(\mathsf{b}\,|\,\mathsf{f})+} \ \cup\ <_{\mathsf{g}} \ \cup\ \{\mathsf{a},\mathsf{b},\mathsf{f}\} \times \{\mathsf{g}\} \\
&= <_{\mathsf{a}} \ \cup\ <_{(\mathsf{b}\,|\,\mathsf{f})+} \ \cup\ \{\mathsf{a}\} \times \{\mathsf{b},\mathsf{f}\} \ \cup\ \{\mathsf{a},\mathsf{b},\mathsf{f}\} \times \{\mathsf{g}\} \\
&= <_{(\mathsf{b}\,|\,\mathsf{f}),(\mathsf{b}\,|\,\mathsf{f})} \ \cup\ \{\mathsf{a}\} \times \{\mathsf{b},\mathsf{f}\} \ \cup\ \{\mathsf{a},\mathsf{b},\mathsf{f}\} \times \{\mathsf{g}\} \\
&= <_{\mathsf{b}\,|\,\mathsf{f}} \ \cup\ \{\mathsf{b},\mathsf{f}\} \times \{\mathsf{b},\mathsf{f}\} \ \cup\ \{\mathsf{a}\} \times \{\mathsf{b},\mathsf{f}\} \ \cup\ \{\mathsf{a},\mathsf{b},\mathsf{f}\} \times \{\mathsf{g}\} \\
&= <_{\mathsf{b}} \ \cup\ <_{\mathsf{f}} \ \cup\ \{\mathsf{b},\mathsf{f}\} \times \{\mathsf{b},\mathsf{f}\} \ \cup\ \{\mathsf{a}\} \times \{\mathsf{b},\mathsf{f}\} \ \cup\ \{\mathsf{a},\mathsf{b},\mathsf{f}\} \times \{d\} \\
&= \{\mathsf{b},\mathsf{f}\} \times \{\mathsf{b},\mathsf{f}\} \ \cup\ \{\mathsf{a}\} \times \{\mathsf{b},\mathsf{f}\} \ \cup\ \{\mathsf{a},\mathsf{b},\mathsf{f}\} \times \{\mathsf{g}\} \\
&= \{(\mathsf{b},\mathsf{b}),\ (\mathsf{b},\mathsf{f}),\ (\mathsf{f},\mathsf{b}),\ (\mathsf{f},\mathsf{f}),\ (\mathsf{a},\mathsf{b}),\ (\mathsf{a},\mathsf{f}),\ (\mathsf{a},\mathsf{g}),\ (\mathsf{b},\mathsf{g}),\ (\mathsf{f},\mathsf{g})\}
\end{aligned}$$

It the development we assumed left associativity for union and concatenation, that implies $\mathsf{r}_1, \mathsf{r}_2, \mathsf{r}_3 = (\mathsf{r}_1, \mathsf{r}_2), \mathsf{r}_3$. It is worth noticing that the relation $<_{\mathsf{r}}$ considers all following siblings of a given type, like (a,g), and not just the immediate ones. Also, $<_{\mathsf{a}} = <_{\mathsf{b}} = <_{\mathsf{f}} = <_{\mathsf{g}} = \varnothing$.

The following two lemmas state that the relation $<_{\mathsf{r}}$ is both sound and complete.

Lemma 4.2.2 (Soundness of $<_{\mathsf{r}}$)**.** *Let* d *be a DTD and* $t = (\sigma, l_t) \in \mathsf{d}$ *a valid tree. If* $(l, l') \in \mathit{FollowingSibling}_t$ *and* $\mathsf{c}_l^t = \mathsf{c.a}$ *then* $\mathsf{type}(l) <_{\mathsf{d}(\mathsf{c})} \mathsf{type}(l')$.

Lemma 4.2.3 (Completeness of $<_{\mathsf{r}}$)**.** *For all DTD* d*, If* $\mathsf{a} <_{\mathsf{d}(\mathsf{c})} \mathsf{b}$ *then there exists a valid tree* $t = (\sigma, l_t) \in \mathsf{d}$*, and a pair of locations* $(l, l') \in \mathit{FollowingSibling}_t$*, such that* $\mathsf{type}(l) = \mathsf{a}$ *and* $\mathsf{type}(l') = \mathsf{b}$*, with* $\mathsf{c}_l^t = \mathsf{c.a}$ *and* $\mathsf{c}_{l'}^t = \mathsf{c.b}$.

Rules for node-test chain inference are easier and defined below.

$$\begin{aligned}\mathsf{T_C}(\,\mathsf{c},\mathtt{node}()\,) &\stackrel{def}{=} \{\,\mathsf{c}\,\}\\ \mathsf{T_C}(\,\mathsf{c},a\,) &\stackrel{def}{=} \{\,\mathsf{c}\mid \mathsf{c}=\mathsf{c}'.\,\mathsf{a}\,\}\\ \mathsf{T_C}(\,\mathsf{c},\mathtt{text}()\,) &\stackrel{def}{=} \{\,\mathsf{c}\mid \mathsf{c}=\mathsf{c}'.\,\mathsf{String}\,\}\end{aligned}$$

Composition of axis and node-filters is noted as

$$\mathsf{T_C}(\,\mathsf{A_C}(\mathsf{c},\mathtt{axis}),\phi\,)$$

that stands for

$$\bigcup_{\mathsf{c}'\in\mathsf{A_C}(\mathsf{c},\mathtt{axis})} \mathsf{T_C}(\,\mathsf{c}',\phi\,)$$

To illustrate, consider the following examples over DTD d_3

$$\mathsf{T}_{\mathsf{C}_{\mathsf{d}_3}}(\,\mathsf{A}_{\mathsf{C}_{\mathsf{d}_3}}(\mathsf{bib.book.author},\mathtt{descendant}),\mathtt{text}()\,) \;=\; \{\,\mathsf{bib.book.author.name.String}\,\}$$

$$\mathsf{T}_{\mathsf{C}_{\mathsf{d}_3}}(\,\mathsf{A}_{\mathsf{C}_{\mathsf{d}_3}}(\mathsf{bib.book.title},\mathtt{following\text{-}sibling}),\mathit{author}\,) \;=\; \{\,\mathsf{bib.book.author}\,\}$$

In this section we defined chain inference for XPath steps, the building blocks of our analysis. We will define next chain inference for query and update expressions. It is worth noting that in the above examples we considered only *rooted* chains, where the first symbol is the root type. This is because these chains are the ones that make sense for the access analysis, despite the fact that all definitions work also for non-rooted chains.

4.2.2 Query Chain Inference

The analysis of query and update chains requires to further structure the inference process, in order to better describe the behavior of query and update expression, and prepare for a more precise independence analysis. For this aim, we introduce several classes of chains that we use to capture accessed and updated data.

As output of inference, our system produces chains of different kinds. The classification resembles that of Marian and Simeon in [MS03] for query path extraction, and of Ghelli et al. in [GRS08] for update path extraction, and is needed in order to reflect different way a query manipulates nodes during its evaluation. Reflecting this distinction is crucial for soundness and precision of the independence analysis.

In our framework, a query chain belongs to one of the following three disjoint classes.

- Return chains type input document nodes (*return nodes*) that are roots of subtrees outputted by the query. All descendants of a return node are in the query result, thus a return chain c implicitly embodies these descendants. Now, if a change made by an update $\mathtt{u}$ targets a *return node* or some of its ancestors or descendants, query-update independence is not guaranteed.

- Used chains type nodes (*used nodes*) belonging to the input document and participating to the query evaluation, without necessarily being part of the result itself. Clearly, if a change of an update $\mathtt{u}$ targets a *used node* or some of its ancestors, then query-update independence is not guaranteed. However, if it targets a used node descendant then independence may still hold.

- Element chains type newly constructed elements; an element chain is of the form $\mathtt{a.c}$, where $\mathtt{a}$ is the tag of the constructed $\mathtt{a}$ element, while $\mathtt{c}$ traverses nodes belonging to the subtree rooted at the $\mathtt{a}$ element. Extracting these chains is important to record the structure of forests built and inserted by update expressions.

For updates, we have one class of chains:

- The purpose of an Update chain, denoted by $\mathtt{c} : \mathtt{c}'$, is twofold: $\mathtt{c}$ types nodes l whose content may be changed by the update and $\mathtt{c}'$, rooted at a child label of l, types descendants of l (either introduced or removed by the update) involved in the changes. For example, given $\mathtt{c} : \mathtt{c}'$, independence is not guaranteed if a query returns an element whose root is typed by $\mathtt{c.a}$, with $\mathtt{a}$ a prefix of $\mathtt{c}'$.

Rules for query chain inference are presented in Table 4.1. These rules prove judgements of the form:

$$\Gamma \vdash_{\mathsf{C}} \mathtt{q} : (\mathsf{r}, \mathsf{v}, \mathsf{e})$$

meaning that starting from a variable environment Γ and a set of chains C, the chain inference for query $\mathtt{q}$ produces the sets r, v and e, containing the return, used and element chains for $\mathtt{q}$, respectively. Γ is a static variable environment that associates each query free-variable $\mathtt{x}$ with a set $\Gamma(\mathtt{x})$ of chains, typing nodes that can be assigned to the variable during query evaluation. When $\mathtt{x}$ is bound to a single chain, we write $\mathtt{x} \mapsto \mathtt{c}$ as a shorthand of $\mathtt{x} \mapsto \{\, \mathtt{c} \,\}$.

All the rules mimic query semantics [DFF+10, BC09b] introduced in Section 2.3. Rule (Empty) returns no chain, as the empty query performs no operation. Rule (Concat) returns the union of return, used and element chains for the two expressions.

Navigational steps are captured by rule (StepF) and (StepUH) dealing with forward and upward/horizontal navigations respectively. Rule (StepF) produces only return chains, by using in turn step chain inference rules defined in the previous section. This is due to the fact that forward navigations only *extend* input chains, without loosing track of visited ancestor types. Differently, rule (StepUH) produces also used chains. These are input chains leading to a non-empty result for the axis inference. These chains are needed for developing a correct access analysis, since return chains produced by an horizontal/upward step may not contain as a prefix the input chain in $\Gamma(\mathtt{x})$ from which they have been generated. Also, these input chains are kept as *used* chains because the descendants of the locations they are typing do not need to be accessed as well.

$$\frac{}{\Gamma \vdash_C () : (\varnothing,\varnothing,\varnothing)}\ (\textsc{Empty}) \qquad \frac{\Gamma \vdash_C q_i : (r_i,v_i,e_i) \quad i{=}1,2}{\Gamma \vdash_C q_1,q_2 : (r_1 \cup r_2, v_1 \cup v_2, e_1 \cup e_2)}\ (\textsc{Conc})$$

$$\frac{\begin{array}{c}\texttt{axis} \in \{\texttt{self},\texttt{child},\texttt{descendant-or-self}\} \\ r_c = \mathsf{T}_C(\mathsf{A}_C(c,\texttt{axis}),\phi) \quad \text{for any } c \in \Gamma(x)\end{array}}{\Gamma \vdash_C x/\texttt{axis}::\phi : (\bigcup_{c\in\Gamma(x)} r_c, \varnothing, \varnothing)}\ (\textsc{StepF})$$

$$\frac{\begin{array}{c}\texttt{axis} \in \{\texttt{parent},\texttt{ancestor},\texttt{ancestor-or-self},\\ \texttt{following-sibling},\texttt{preceding-sibling}\} \\ r_c = \mathsf{T}_C(\mathsf{A}_C(c,\texttt{axis}),\phi) \quad \text{for any } c \in \Gamma(x)\end{array}}{\Gamma \vdash_C x/\texttt{axis}::\phi : (\bigcup_{c\in\Gamma(x)} r_c\ ,\ \bigcup_{\substack{c\in\Gamma(x)\\ r_c\neq\varnothing}} \{c\}, \varnothing)}\ (\textsc{StepUH})$$

$$\frac{\begin{array}{c}\Gamma \vdash_C q_1 : (r_1,v_1,e_1) \\ \Gamma[x\mapsto c] \vdash_C q_2 : (r_c,v_c,e_c) \quad \text{for any } c \in r_1\end{array}}{\Gamma \vdash_C \texttt{for } x \texttt{ in } q_1 \texttt{ return } q_2 : (\bigcup_{c\in r_1} r_c, v_1 \cup \bigcup_{\substack{c\in r_1\\ r_c\cup e_c\neq\varnothing}} (v_c\cup\{c\}), \bigcup_{c\in r_1} e_c)}\ (\textsc{For})$$

$$\frac{\Gamma \vdash_C q_1 : (r_1,v_1,e_1) \qquad \Gamma[x\mapsto r_1] \vdash_C q_2 : (r_2,v_2,e_2)}{\Gamma \vdash_C \texttt{let } x := q_1 \texttt{ return } q_2 : (r_2, r_1\cup v_1\cup v_2, e_2)}\ (\textsc{Let})$$

$$\frac{\Gamma \vdash_C q_i : (r_i,v_i,e_i)\ \ i{=}0..2}{\Gamma \vdash_C \texttt{if } (q_0) \texttt{ then } q_1 \texttt{ else } q_2 : (r_1\cup r_2,\ \bigcup_{i=0..2} v_i \cup r_0, e_1\cup e_2)}\ (\textsc{If})$$

$$\frac{}{\Gamma \vdash_C \text{“}\texttt{txt}\text{”} : (\varnothing,\varnothing,\{\mathsf{String}\})}\ (\textsc{Text})$$

$$\frac{\begin{array}{c}\Gamma \vdash_C q : (r,v,e) \\ e_0 = \{\ a.a.c' \mid c.a \in r, c.a.c' \in \overline{r}\ \} \cup \{\ a.c \mid c \in e\ \} \cup \{\ a \mid r\cup e{=}\varnothing\ \}\end{array}}{\Gamma \vdash_C \texttt{<}a\texttt{>}q\texttt{</}a\texttt{>} : (\varnothing, \overline{r}\cup v, e_0)}\ (\textsc{Elt})$$

Table 4.1: Chain Inference Rules for Queries

The following examples are made on the DTD d_3 in Figure 3.1, and we assume that variable doc is always bound to bib in Γ.

To illustrate rules (StepF) and (StepUH), consider , and the query

$$\texttt{doc}//\textit{title}/\texttt{following-sibling} :: \textit{author}$$

Chain inference for this expression is done by analyzing the two steps in sequence. Rule (StepF) infers chain bib.book.title for the step $\texttt{doc}//\textit{title}$. By applying rule (StepUH) on the step $\texttt{following-sibling} :: \textit{author}$, provided that $\Gamma' = \Gamma \cup \{\ \texttt{x} \mapsto \mathsf{bib.book.title}\ \}$, we obtain the following derivation.

$$\frac{\mathbf{T}_{\mathsf{C}_{\mathsf{d}_3}}(\,\mathbf{A}_{\mathsf{C}_{\mathsf{d}_3}}(\mathsf{bib.book.title}, \texttt{following-sibling}), \textit{author}\,) = \{\mathsf{bib.book.author}\}}{\Gamma' \vdash_{\mathsf{C}_{\mathsf{d}_3}} \texttt{x/following-sibling} :: \textit{author} : (\{\mathsf{bib.book.author}\}, \{\mathsf{bib.book.title}\}, \varnothing)}$$

The rule produces a return chain, namely bib.book.author, that is typing subtrees rooted at author returned by the query, but also a used chain, namely bib.book.title, that is typing locations accessed before getting to the result.

Rules (For) and (Let) are very similar, thus we mainly comment on the former. According to query semantics, rule (For) performs an iteration on the set of *return* chains inferred for q_1 and, at each iteration step, it evaluates q_2. We illustrate this process through an example. Consider the DTD d_3 and the following query

$$\texttt{for x in doc/descendant} :: \textit{name}\ \texttt{return x/parent} :: *$$

The expression iterates on the sequence of document nodes labeled with *name*, and outputs the parent of the each node. Here, chain inference for the left subexpression $\texttt{/descendant} :: \textit{name}$ produces two return chains $\mathsf{c_a}$ = bib.book.author.name and $\mathsf{c_p}$ = bib.book.publisher.name. Accordingly, chain inference for the right subexpression $\texttt{x/parent} :: *$ is done twice, with x bound to $\mathsf{c_a}$ and $\mathsf{c_p}$ respectively.

Moreover, three other operations are necessary in order to perform a correct and precise analysis.

First, *return* chains for q_1 are converted into *used* chains for the whole expression. This is needed because chain inference is a bottom-up process: inside q_1 a path expression is seen as a query producing a *result* (and as such it *locally* produces return chains), while it only selects nodes to be *used* in the outer iteration $\texttt{for x in}\ \mathsf{q}_1\ \texttt{return}\ \mathsf{q}_2$. To illustrate, consider the query

$$\begin{array}{l}\texttt{for x in doc}//\textit{publisher}\ \texttt{return}\\ \quad\texttt{for y in doc}//\textit{name}\ \texttt{return y}\end{array}$$

In this case the return chain bib.book.publisher inferred for the step $//\textit{publisher}$, should not be considered as a return chain for the whole expression, since the descendants of

nodes labeled with *publisher* are not accessed by the expression. Rather, it should be considered as a used chain for the whole expression.

Second, return chains for $\mathtt{q}_1$ that are irrelevant for $\mathtt{q}_2$ are filtered out. To illustrate, consider the query

```
for x in doc//node() return
if (x/title) then x/author else ()
```

Chain inference for the expression doc//node() returns all rooted chains of the schema. If these are all kept in the final result, the analysis is jeopardized since the query would be consider as dependent wrt almost every update. Our system avoids this, by discarding all return chains of doc//node() that do not further satisfy x/*title*. Chain inference for the whole expression only produces the used chain bib.book.title and the return chain bib.book.author.

Third, used chains for $\mathtt{q}_1$ become chains for the whole expression. This operation is needed in order to ensure a sound analysis. Consider the DTD $\mathtt{d}_3$ and following query

```
for x in doc//title/following-sibling::author return x
```

here the left query doc//*title*/following-sibling::*author* makes the system inferring a return chain bib.book.author and a used chain bib.book.title. Now, in order to have a sound analysis, the chain bib.book.title should be inferred as a used chain for the whole expression.

Rule (If) is similar to rule (Concat). It invokes chain inference for each subexpression, and then outputs the proper union of sets inferred for the subexpressions. As an example consider the query

```
for x in doc//book return
if (x/price) then x/author else x/title
```

This case illustrates well that a correct analysis needs the chains inferred for all subexpressions, namely bib.book.price, bib.book.author, bib.book.title. Return chains for the then-branch (bib.book.author) and the else-branch (bib.book.title) become return chains for the whole expression. Differently, the return chain inferred for the boolean conditional-query (bib.book.price) is converted into a used chains for the whole expression, since this is just needed to perform boolean tests on the existence of some nodes, without requiring the whole subtrees.

Element queries <*a*>q</*a*> are dealt with by rule (Elt). In the element construction rule, $\overline{\tau}$ denotes all descendant chains of chains in the set τ wrt C.

$$\overline{\tau} \stackrel{def}{=} \{\, \mathsf{c.c}' \in \mathsf{C} \mid \mathsf{c} \in \tau \,\}$$

This rule infers element chains of the form a.c, where c is obtained from either an element or return chain of q. The rule also infers a set of used chains, composed by i) used

chains of q and *ii*) return chains of q converted into used chains for the whole expression. To this end $\overline{\mathsf{r}}$ is used to extend returned chains of the inner query. This return-to-used chain conversion is needed to correctly handle nested element construction. For instance, consider the following query q = `for x in doc`//*book* `return` <*livre*>q′</*livre*> where

$$\mathsf{q}' = \begin{array}{l} \mathtt{x}/\mathit{title}\ , \\ \mathit{<auteur>} \\ \ \mathtt{x}/\mathit{author}/\mathit{name} \\ \mathit{</auteur>} \end{array}$$

The expression restructures the result of a query selecting some informations on the books in the bibliography, changing both the structure and the label of nodes of a tree labeled with *book*. Element chains for q are inferred in terms of chains for q′. So, element chains for q are livre.title and livre.auteur.name, given that for q′ the return chain is bib.book.title (for x/*title*) and the element chain is auteur.name (for <*auteur*>x/*author*/*name*</*auteur*>). In order to avoid ending up with a wrong element chain for q, i.e., title.name the return chain bib.book.author.name inferred for x/*author*/*name* does not have to belong to the set of return chains for q′. At the same time, we must record the fact that the whole subtree rooted at name is accessed. This is handled by the return-to-used conversion of the return chain bib.book.author.name when inferring chains for <*auteur*>x/*author*/*name*</*auteur*>. The return-to-used conversion extends r, the set of return chains of the inner query x/*author*/*name*, with all possible continuations by means of $\overline{\mathsf{r}}$ and then it converts chains in $\overline{\mathsf{r}}$ to used chains for the whole expression.

It is worth stressing that if we just convert return chains to used ones without the extension $\overline{\mathsf{r}}$, then we loose their semantic property of representing entire subtrees of data. Notice that this extension is needed for the purpose of the formal presentation although any efficient implementation can avoid performing these extensions by using intensional representations.

Rule (Text) deals with expressions building new text nodes. The rule simply infers String as an element chain for the built textual node. With a little abuse of notation, for simplicity, we preferred not to use another class of chains.

In this section we illustrated chain inference rules for for query expressions. As update expressions are defined in terms of queries, we will see, in the next section, that chain inference rules for updates rely on the rules for query chain inference.

4.2.3 Update Chain Inference

Rules for update chain inference are presented in Table 4.2. These rules prove judgments of the form

$$\Gamma \vdash_{\mathsf{C}} \mathtt{u} : \mathsf{U}$$

meaning that starting from a static variable environment Γ and a set of chains C, the chain inference for the update u produces the set of update chains U.

As seen before, update chains are of the form $\mathsf{c} : \mathsf{c}'$. Essentially, the *prefix* c types updated nodes, that are nodes whose children are modified by the update, while the *suffix* c' types modified children or new inserted/replaced descendants.

The most important inference rules are the ones corresponding to delete, rename, insert and replace operations, which are explained in the sequel.

$$\begin{array}{rcl} \mathsf{bib} & \leftarrow & \mathsf{book}* \\ \mathsf{book} & \leftarrow & \mathsf{title}?, \mathsf{author}*, \mathsf{publisher}* \\ \mathsf{author}, \mathsf{publisher} & \leftarrow & \mathsf{name}* \\ \mathsf{title}, \mathsf{name} & \leftarrow & \mathsf{String}? \end{array}$$

Figure 4.2: DTD d_4

The following examples are made on the DTD d_4, and again we assume that variable doc is always bound to bib in Γ. We recall that when the update target query returns a sequence with more than one location, this has to be bound to a variable and then the update command has to applied by means of iteration, otherwise a dynamic error is raised. However for the sake of conciseness, and w.l.o.g., in the following examples we do not make iteration explicit and assume that the update target query returns only one location.

Inference for delete expressions is defined by the (DELETE) rule, which simply puts the separator ":" just before the last symbol of each return chain of the target query. A delete chain $\mathsf{c} : \mathsf{a}$ captures the fact that a node typed by c has a child labeled by a which may be deleted. As an example, consider the update

$$\texttt{delete doc//}\textit{publisher}$$

By applying rule (DELETE) we obtain the following derivation

$$\frac{\Gamma \vdash_{\mathsf{C}_{\mathsf{d}_4}} \texttt{doc//}\textit{publisher} : (\{\mathsf{bib.book.publisher}\}, \varnothing, \varnothing)}{\Gamma \vdash_{\mathsf{C}_{\mathsf{d}_4}} \texttt{delete doc//}\textit{publisher} : \{\mathsf{bib} : \mathsf{book} : \mathsf{publisher}\}}$$

showing how return chains inferred for the target query become update chains.

For rename updates, the process is similar. Rule (RENAME) infers chains $\mathsf{c} : \mathsf{a}$ where a is the tag of the target node before renaming, but it also produces chains $\mathsf{c} : \mathsf{b}$, where b is the tag of the target node after renaming. Recall that textual nodes are never target of rename commands. As an example, consider the DTD d_4 and the rename update

$$\texttt{rename doc//}\textit{author}\ \texttt{as}\ \textit{publisher}$$

By applying rule (RENAME) we obtain the following derivation.

$$\frac{\Gamma \vdash_{\mathsf{C}_{\mathsf{d}_4}} \texttt{doc}//\mathit{author} : (\{\mathsf{bib.book.author}\}, \varnothing, \varnothing)}{\Gamma \vdash_{\mathsf{C}_{\mathsf{d}_4}} \texttt{rename doc}//\mathit{author}\ \texttt{as}\ \mathit{publisher} : \{\mathsf{bib.book} : \mathsf{author}, \mathsf{bib.book} : \mathsf{publisher}\}}$$

In this case the update chains capture both the fact that subtrees rooted at author disappear and that subtrees rooted at publisher appear.

Chain inference for insert-into expressions (position ranges over into, first and last) is specified by the rule (INSERT-1). For any chain $\mathsf{c} : \mathsf{c}'$ inferred, the prefix c is a return chain of the target query q_0 (typing the insertion point), while the suffix c' is either a return or element chain of the source expression q typing a branch of a node element returned by q itself; this element can either be a new one or a sub-element of the input document; in both cases the suffix chain corresponds to inserted data. As an example, consider the DTD d_4 and the update expression `insert` <*name*/> `into doc/self` :: *bib*/*book*. By applying rule (INSERT-1) we obtain the following.

$$\frac{\Gamma \vdash_{\mathsf{C}_{\mathsf{d}_4}} \mathit{<name/>} : (\varnothing, \varnothing, \{\mathsf{name}\}) \quad \Gamma \vdash_{\mathsf{C}_{\mathsf{d}_4}} \texttt{doc/self} :: \mathit{bib}/\mathit{book} : (\{\mathsf{bib.book}\}, \varnothing, \varnothing)}{\Gamma \vdash_{\mathsf{C}_{\mathsf{d}_4}} \texttt{insert}\ \mathit{<name/>}\ \texttt{into doc/self} :: \mathit{bib}/\mathit{book} : \{\mathsf{bib.book} : \mathsf{name}\}}$$

Rule (INSERT-2) is similar, and deals with insert-before/after updates.

The two rules for insert updates rely on the fact that element chains can be extracted in source queries. The fact that we opportunely attach element chains of the source-expression to result chains of the target-expression, allow us to have a precise analysis. It is worth saying that related update analysis of [BC09a] and [GRS08] define as updated *everything* below the parent of the source location. This design choice has a strong impact for this case, since it is likely to make the analysis deem the update as dependent with almost all queries accessing information concerning the book, while it should only raise conflicts on *name* elements.

This last case suggests also that element chains are necessary for a precise independence analysis, when dealing with updates. To illustrate, consider the following update over the DTD in Figure 4.2.

```
for x in doc//book return
insert <author>q'</author> into x
```

Here the source expression is an element query, for which we infer element chains of the form $\mathsf{author.c}'$, with c' a chain inferred for q'. The update chain $\mathsf{bib.book} : \mathsf{author.c}'$ is obtained by concatenation of the chain bib.book associated with the target expression x, and the chain $\mathsf{author.c}'$. This allows one to conclude independence wrt the query

$$\frac{\Gamma \vdash_C q_0 : (r_0, v_0, e_0) \quad U = \{\, c : a \mid c.a \in r_0 \,\}}{\Gamma \vdash_C \texttt{delete}\ q_0 : U}\ (\textsc{Delete}) \qquad \frac{\Gamma \vdash_C q_0 : (r_0, v_0, e_0) \quad U = \{\, c : a \mid c.a \in r_0 \,\} \cup \{\, c : b \mid c.a \in r_0 \,\}}{\Gamma \vdash_C \texttt{rename}\ q_0\ \texttt{as}\ b : U}\ (\textsc{Rename})$$

$$\frac{\Gamma \vdash_C q : (r, v, e) \quad \Gamma \vdash_C q_0 : (r_0, v_0, e_0) \quad \texttt{pos} \in \{\texttt{into}, \texttt{first}, \texttt{last}\} \quad U = \{\, c : c' \mid c \in r_0,\ c'.c'' \in e,\ c' \neq \epsilon \,\} \cup \{\, c : a.c'' \mid c.a \in r_0,\ c'.a \in r,\ c'.a.c'' \in C \,\}}{\Gamma \vdash_C \texttt{insert}\ q\ \texttt{pos}\ q_0 : U}\ (\textsc{Insert-1})$$

$$\frac{\Gamma \vdash_C q : (r, v, e) \quad \Gamma \vdash_C q_0 : (r_0, v_0, e_0) \quad \texttt{pos} \in \{\texttt{after}, \texttt{before}\} \quad U = \{\, c : c' \mid c.a \in r_0,\ c'.c'' \in e,\ c' \neq \epsilon \,\} \cup \{\, c : b.c'' \mid c.a \in r_0,\ c'.b \in r,\ c.'b.c'' \in C \,\}}{\Gamma \vdash_C \texttt{insert}\ q\ \texttt{pos}\ q_0 : U}\ (\textsc{Insert-2})$$

$$\frac{\Gamma \vdash_C q : (r, v, e) \quad \Gamma \vdash_C q_0 : (r_0, v_0, e_0) \quad U = \{\, c : a \mid c.a \in r_0 \,\} \cup \{\, c : b.c'' \mid c.a \in r_0,\ c'.b \in r,\ c'.b.c'' \in C \,\} \cup \{\, c : c' \mid c.a \in r_0,\ c'.c'' \in e,\ c' \neq \epsilon \,\}}{\Gamma \vdash_C \texttt{replace}\ q_0\ \texttt{with}\ q : U}\ (\textsc{Replace})$$

$$\frac{}{\Gamma \vdash_C u = () : \varnothing}\ (\text{TU-}\textsc{Empty}) \qquad \frac{\Gamma \vdash_C u_1 : U_1 \quad \Gamma \vdash_C u_2 : U_2}{\Gamma \vdash_C u_1, u_2 : U_1 \cup U_2}\ (\text{TU-}\textsc{Concat})$$

$$\frac{\Gamma \vdash_C q : (r, v, e) \quad \Gamma[x \mapsto c] \vdash_C u : U_c \quad \text{for any } c \in r}{\Gamma \vdash_C \texttt{for}\ x\ \texttt{in}\ q\ \texttt{return}\ u : \bigcup_c U_c}\ (\text{TU-}\textsc{For}) \qquad \frac{\Gamma \vdash_C q : (r, v, e) \quad \Gamma[x \mapsto r] \vdash_C u : U}{\Gamma \vdash_C \texttt{let}\ x\ :=\ q\ \texttt{return}\ u : U}\ (\text{TU-}\textsc{Let})$$

$$\frac{\Gamma \vdash_C q : (r, v, e) \quad \Gamma \vdash_C u_i : U_i \quad i = 1, 2}{\Gamma \vdash_C \texttt{if}\ (q)\ \texttt{then}\ u_1\ \texttt{else}\ u_2 : U_1 \cup U_2}\ (\text{TU-}\textsc{If})$$

Table 4.2: Chain Inference Rules for Updates

doc//*title*, whose unique return chain is bib.book.title (forasmuch as *title* element is never a descendant of an *author* element): the update chain is not a prefix of the query chain and vice-versa.

Now, let us do the analysis without considering element chains: for the source expression <*author*>q′</*author*>, the best that can be done is to infer the chain

$$\mathsf{bib.book} : \_$$

telling that something happens beneath book elements. As a consequence, we would not deduce the independence. As said in the introduction, for similar reasons [BC09a] wrongly excludes independence for similar cases.

In the presence of nested element construction, the same remark holds. In the previous example, if q′ is

$$<first>Umberto</first>, <second>Eco</second>$$

then by composing element chains during the inference, we end up with the following update chains bib.book : author.first.String, bib.book : author.second.String, bib.book : author, bib.book : author.first and bib.book : author.second. Indeed, this is necessary to exclude independence wrt the query doc//*author*/*email* (assuming the DTD allows for email elements into author elements).

The rule for replace expressions (Replace) is built on the same principles as (Insert-2) and (Delete) rules. The chains produced by the expression capture the fact that *replaced* subtrees are now missing, and that *replacing* subtrees can now be retrieved. As an example, consider the DTD d_4 and the rename update

$$\texttt{replace doc//}\mathit{author}/\mathit{name}\ \texttt{with doc//}\mathit{publisher}/\mathit{name}$$

By applying rule (Replace) we obtain the following.

$$\frac{\Gamma \vdash_{\mathsf{C}_{\mathsf{d}_4}} \texttt{doc//}\mathit{author}/\mathit{name} : (\{\mathsf{bib.book.author.name}\}, \varnothing, \varnothing) \quad \Gamma \vdash_{\mathsf{C}_{\mathsf{d}_4}} \texttt{doc//}\mathit{publisher}/\mathit{name} : (\{\mathsf{bib.book.publisher.name}\}, \varnothing, \varnothing)}{\Gamma \vdash_{\mathsf{C}_{\mathsf{d}_4}} \texttt{replace doc//}\mathit{author}/\mathit{name}\ \texttt{with doc//}\mathit{publisher}/\mathit{name} : \{\ \begin{array}{l}\mathsf{bib.book.author : name},\\ \mathsf{bib.book.author : name.String}\end{array}\ \}}$$

Once again the update chain is obtained by the concatenation of the target and the source expression. Notice that the update chain contains also the expansion of the return chain for the source query. This is done in order to completely trace inserted structures.

Remaining rules for the empty update, concatenation, iteration, let-binding and conditional expressions are formulated in the same spirit as for query chain inference.

In this section we presented rules for update chain inference, showing also that they are built on top of rules for query chain inference. We illustrated the precision of the

chain inference with examples compared with related work. In the next section, we will formally prove that query and update chain inference rules are correct, in the sense that they allow to infer types capturing all data which are accessed and updated by query and update expression.

4.3 Correctness and Precision of Chain Inference

In this section we show the correctness of the chain inference system for queries and updates. We begin by showing correctness of XPath step chain inference, for which completeness also hold. Then, we state the correctness for queries and updates. The proofs of all statements of this section can be found in the appendix 9.

Soundness of axis and node-test chain inference wrt query semantics is stated below.

Lemma 4.3.1 (Soundness of Step Chain Inference)**.** *Let* d *be a DTD,* $t = (\sigma, l_t) \in \mathsf{d}$ *be a valid tree, and* $l \in dom(\sigma)$ *be a location of the tree. If*

$$\sigma, (\mathtt{x} := l) \vDash \mathtt{x/axis} :: \phi \;\Rightarrow\; \sigma, L$$

then for each location $l' \in L$ *we have that*

$$\mathsf{c}^t_{l'} \in \boldsymbol{T}_{\mathsf{C}}(\boldsymbol{A}_{\mathsf{C}}(\mathsf{c}^t_l, \mathtt{axis}), \phi)$$

A *complete* chain inference demands also the set of inferred chains to be minimal, so that each inferred chain is typing a result location belonging to some valid tree instance. The completeness of $\Rightarrow_{\mathsf{d}}$ and $<_{\mathsf{d(a)}}$ entails that step chain inference is minimal.

Lemma 4.3.2 (Completeness of Step Chain Inference)**.** *Let* d *be a DTD and* $\mathsf{c} \in \mathsf{C}_{\mathsf{d}}$ *a chain. If*

$$\mathsf{c}' \in \boldsymbol{T}_{\mathsf{C}_{\mathsf{d}}}(\boldsymbol{A}_{\mathsf{C}_{\mathsf{d}}}(\mathsf{c}, \mathtt{axis}), \phi)$$

then there exists a valid tree $t = (\sigma, l_t) \in \mathsf{d}$ *and a location* $l \in dom(t)$ *such that* $\mathsf{c}^t_l = \mathsf{c}$ *and, assuming that* $\sigma, (\mathtt{x} := l) \vDash \mathtt{x/axis} :: \phi \;\Rightarrow\; \sigma, L$ *we have that* $\mathsf{c}^t_{l'} = \mathsf{c}'$*, for some* $l' \in L$*.*

Showing that chain inference is complete for navigational axes is easy, since chains carry a navigational context that leads inference rules to strictly mimic axis semantics even with backwards and horizontal navigations.

4.3.1 Soundness of Query Chain Inference

Proving soundness of query chain rules consists of proving that, for any schema instance, any node used or built by the query $\mathtt{q}$ is captured (typed) by the chains inferred for $\mathtt{q}$. The proof relies on the notion of projection for XML [MS03, BCCN06].

A tree t' is said to be a *projection* of t, denoted by $t' \preceq t$, if t' is obtained from t by removing some subtrees of t. According to our formalism, we define the projection of a tree t induced by a set of locations $\mathcal{L} \subseteq dom(t)$, provided that $\mathcal{L}$ is non-empty and upward closed with respect to the node parent-child relationship in t. This simply means that if $l \in \mathcal{L}$ and l' is the parent of l in σ then $l' \in \mathcal{L}$.

Given a sequence of locations L, we define the filtering of L wrt a set of node locations $\mathcal{L}$, denoted by $L_{|\mathcal{L}}$, as the subsequence of L that contains only identifiers in $\mathcal{L}$ and that preserves the order of L. The projection of a tree $t = (\sigma, l_t)$ wrt a set of locations $\mathcal{L} \subseteq dom(\sigma)$, is defined as $t_{|\mathcal{L}} = (\sigma_{|\mathcal{L}}, l_t)$, where

$$\sigma_{|\mathcal{L}} \stackrel{def}{=} \{\, l \leftarrow a[L_{|\mathcal{L}}] \mid l \in \mathcal{L},\ (l \leftarrow a[L]) \in \sigma \,\} \cup \{\, l \leftarrow \text{“}\mathtt{txt}\text{”} \mid l \in \mathcal{L},\ (l \leftarrow \text{“}\mathtt{txt}\text{”}) \in \sigma \,\}$$

That is the store obtained by removing the use of all element and text node locations not in $\mathcal{L}$.

Given a query $\mathtt{q}$, we can define a notion of projection wrt $\mathtt{q}$. We say that a store $\sigma_{|\mathcal{L}}$ is a $\mathtt{q}$-projection of σ if, provided that $\sigma, \gamma \vDash \mathtt{q} \Rightarrow \sigma, L$ and $\sigma_{|\mathcal{L}}, \gamma \vDash \mathtt{q} \Rightarrow \sigma_{|\mathcal{L}}, L'$ we can conclude

$$(\sigma, L) \cong (\sigma_{|\mathcal{L}}, L')$$

This definition enforces $\mathcal{L}$ to contain all locations needed in order to evaluate $\mathtt{q}$, without altering the result of $\mathtt{q}$. Given two trees $t = (\sigma, l)$ and $t' = (\sigma', l)$, if σ' is a $\mathtt{q}$-projection of σ then t' is a $\mathtt{q}$-projection of t.

A tree t' is said to be a *minimal* $\mathtt{q}$-projection of t if for all t'' *strict* projections of t' we have that t'' is not a $\mathtt{q}$-projection of t' (or, equivalently, of t). Here strict means that $dom(t'') \subsetneq dom(t')$. Notice that, in general, a projection is not minimal. In the sequel we also show that a minimal projection is not unique, due to the query language considered.

To illustrate the above concepts, consider again the tree t_2 in Figure 4.1 that we report below. A projection of t_2 can be obtained by choosing the set of locations $\mathcal{L} = \{l_{t_2}, l_1, l_2, l_3, l_4\}$, that induces the projection of the book element with its own children nodes. However the sets $\mathcal{L}_1 = \{l_1, l_2\}$ or $\mathcal{L}_2 = \{l_{t_2}, l_2\}$ do not induce projections because they are not closed wrt the node parent-child relation, since $\mathcal{L}_1$ misses l_{t_2} and $\mathcal{L}_2$ misses l_1. Neither $\mathcal{L} = \varnothing$ induces a projection because it contains no location.

Now, let us consider the query $\mathtt{q} = //\mathit{title}$, selecting all titles of book in the store. By evaluating $\mathtt{q}$ over t_2, we get the subtree rooted at l_2, since there is only one book in t_2.

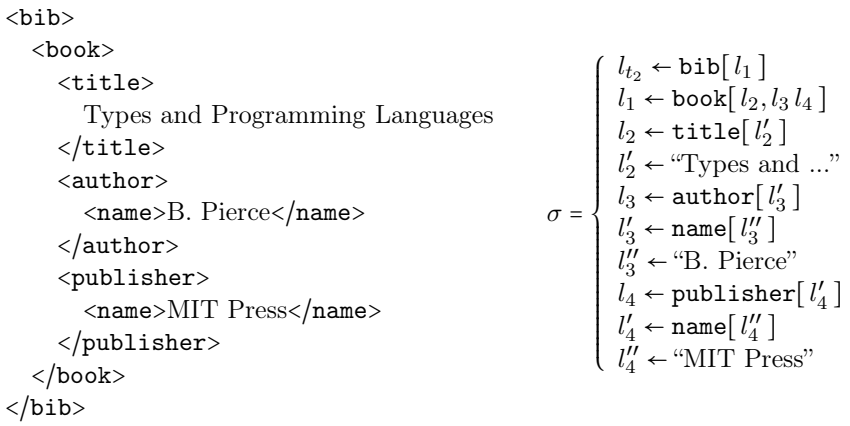


Figure 4.3: Document t_2 and store (σ, l_{t_2})

Also, in order to reach the result, we need all ancestor locations of l_2, namely l_t and l_1. The set of locations $\mathcal{L}_{\mathtt{q}} = \{l_{t_2}, l_1, l_2, l'_2\}$ can be used in order to build a $\mathtt{q}$-projection of t_2. The result of $\mathtt{q}$ over $t_{2|\mathcal{L}_{\mathtt{q}}}$ is equal to $\mathtt{q}$ over t_2. In fact, all other locations in the store are not needed by the query because they concern complementary information such as authors and publishers of a book.

It is worth noticing that in the definition of projection we employed a notion of forest-equivalence $\cong$, that checks only if the two resulting forests are isomorphic, regardless of the node identifiers. Above, we have an example of a stronger equivalence, since not only the structure of the results is the same, but also the node identifiers coincide.

To illustrate minimality of projection, consider again the tree t_2, a dynamic environment $\gamma = \{\, \mathtt{doc} \mapsto l_{t_2} \,\}$ and the query

$$\begin{aligned} \mathtt{q} = \;& \mathtt{for\ x\ in\ doc}//\mathit{book}\ \mathtt{return} \\ & \mathtt{if}\ (\mathtt{x}/\mathit{author}, \mathtt{x}/\mathit{publisher})\ \mathtt{then\ x}/\mathit{title}\ \mathtt{else}\ () \end{aligned}$$

that returns the title of a book either if this has an author or a publisher child node (that is, if the sequence $(\mathtt{x}/\mathit{author}, \mathtt{x}/\mathit{publisher})$ is non-empty). For this expression, we have a number of $\mathtt{q}-$projections, namely

$$\mathcal{L}_1 = \{l_{t_2}, l_1, l_2, l'_2, l_3, l_4\} \quad \mathcal{L}_2 = \{l_{t_2}, l_1, l_2, l'_2, l_3\} \quad \mathcal{L}_2 = \{l_{t_2}, l_1, l_2, l'_2, l_4\}$$

and all backward-closed supersets of $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3$. It could seem surprisingly that $\mathcal{L}_2$ and $\mathcal{L}_3$ are projections, because they both discard a node location that is used for building the sequence used by the conditional test. Anyway this is correct, since we need only one of l_3 and l_4 in order to ensure the conditional expression to be non-empty. In this case we have that $\mathcal{L}_1$ is not minimal simply because $\mathcal{L}_2, \mathcal{L}_3 \subsetneq \mathcal{L}_1$. Differently, $\mathcal{L}_2$ and $\mathcal{L}_3$

are minimal, since by removing any of their locations we may either break backward-closeness or alter the query result (by turning the value of the conditional expressions to empty). This shows that a minimal projection is not unique.

The goal of the chain analysis is to capture all nodes needed by all minimal projections of the query wrt all documents valid wrt a schema. To this end, we define how to connect chains with projections. Given a set of chains τ, the set $\mathcal{L}_\tau^t$ of locations in t typed by chains in τ is defined as follows.

$$\mathcal{L}_\tau^t \stackrel{def}{=} \{\, l \in dom(t) \mid \mathsf{c}_l^t.\mathsf{c} \in \tau \,\}$$

Dynamic and static variable environments have to link data and types in a consistent way, in order to ensure correctness of chain inference. This is stated next.

Definition 4.3.3 (Consistent environments)**.** *Let* d *be a DTD,* $t = (\sigma, l_t) \in \mathsf{d}$ *a valid tree,* γ *a dynamic environment binding variables to locations of* σ *and* Γ *a static environment binding variables to chains of* C_d*. The environments* γ *and* Γ *are said to be consistent if* $l \in \gamma(\mathsf{x})$ *implies* $\mathsf{c}_l^\sigma \in \Gamma(\mathsf{x})$*, for all* $l \in dom(\sigma)$*. We denote this by* $\sigma \vDash_\mathsf{d} \gamma : \Gamma$*.*

The following theorem formally states that chains inferred for a query q cover the structure of data relevant for the query, and newly constructed elements.

Theorem 4.3.4 (Soundness of Query Chain Inference)**.** *Let* d *be a DTD,* $t = (\sigma, l_t) \in \mathsf{d}$ *a valid tree,* q *a query,* γ *a dynamic environment and* Γ *a static environment such that* $\sigma \vDash_\mathsf{d} \gamma : \Gamma$*. Provided* $\sigma, \gamma \vDash \mathsf{q} \Rightarrow \sigma_{new}, L$ *and* $\Gamma \vdash_{\mathsf{C}_\mathsf{d}} \mathsf{q} : (\mathsf{r}, \mathsf{v}, \mathsf{e})$ *the following hold.*

1. *if* t^{min} *is a minimal* q*-projection of* t *then* $t^{min} \preceq t_{|\mathcal{L}_{\mathsf{r}\cup\mathsf{v}}^t} \preceq t$
2. *if* $t^{new} = (\sigma_{new}, l_{new})$ *with* $l_{new} \in L \setminus dom(\sigma)$ *is a fresh subtree then* $dom(t^{new}) = \mathcal{L}_\mathsf{e}^{t^{new}}$

The first item of Theorem 4.3.4 states that chain inference is sound for used and return chains: a projection of any valid input tree made in terms of used and return chains includes every minimal q-projection, hence preserves query semantics (the projection contains all the query needs for its evaluation). The second item is dedicated to element chain inference which is one of the key feature of our query-update analysis as already illustrated. Intuitively, this statement says that if element chains are used to project newly constructed elements (notice that $l' \in L \setminus dom(\sigma)$) no node is pruned out, so element chains cover all possible chains in new elements of the query result. We conclude that our type inference system gives, as a side effect, a type-projector for the query [BCCN06].

4.3.2 Soundness of Update Chain Inference

Proving update chain soundness consists of establishing a link between i) nodes in the stores t and $\mathtt{u}(t)$ that are *involved* in the changes (deletion, insertion, renaming, and replacements) made by $\mathtt{u}$ and ii) nodes in these trees which are typed by the chains statically inferred for $\mathtt{u}$.

We stress that the evaluation of an update $\mathtt{u}$ on a tree $t = (\sigma, l_t) \in \mathtt{d}$ is a two phase process, that first generates the PUL ω together with an extended σ_ω containing the arguments of atomic insert and replace updates, for which the following relations hold.

$$\sigma, \gamma \vDash \mathtt{u} \Rightarrow \sigma_\omega, \omega \qquad\qquad \sigma_\omega \vDash \omega \rightsquigarrow \sigma_\mathtt{u}$$

Recall that σ_ω includes σ, and that $\mathtt{u}(t)$ denotes the tree $(\sigma_\mathtt{u}@l_t, l_t)$.

Definition 4.3.5 (Involved Location)**.** *We say that the update* $\mathtt{u}$ *involves the location* $l \in dom(\sigma_\omega)$ *if one of the following hold.*

- l *is the target location of an elementary delete command in* ω
- l *is the target location of an elementary rename command in* ω
- l *is a location, or the descendant of a location, in the source list* L *of a command* $\mathtt{ins}(L, \_, \_)$ *in* ω
- l *is the target location of an elementary replace command in* ω
- l *is a location, or the descendant of a location, in the source list* L *of a command* $\mathtt{repl}(\_, L)$ *in* ω

The static analysis for update expressions is correct if all involved locations are typed by some update chains. In the following, we present examples of involved locations and how they are typed by the analysis.

The following examples are made over the tree t_2 in Figure 4.1, which is valid against the DTD $\mathtt{d}_4$ in Figure 4.2. As before, we assume that variable `doc` is bound to location l_{t_2} during dynamic evaluation and bound to type `bib` during static analysis.

Consider the following sequence of delete and rename updates

`delete doc`//*title*, `rename doc`//*publisher* `as` “author”

The update produces a pending list composed of two atomic update commands

$$\mathtt{ren}(l_4, \text{“author”}) \ , \ \mathtt{del}(l_2)$$

meaning that the label of l_4 (the publisher node) has to be renamed as “author” and that the location l_2 has to be deleted. In this case both l_2 and l_4 are involved locations, since

they are target locations of elementary update commands. Note first that an involved location may belong to the initial tree t but not to the updated tree $\mathtt{u}(t)$, for instance l_2, and a location may also belong to both trees like for instance l_4.

The static analysis for the delete updates over the DTD d_4 produces the chain c = bib.book : title, that is the chain of the deleted location l_2, in fact $\mathsf{c}_{l_2}^{t_2} = \mathsf{c}$. Concerning the rename expression, two chains are inferred c_1 = bib.book : publisher and c_2 = bib.book : author. These two chains type the renamed node in the original and in the updated tree respectively, since

$$\mathsf{c}_{l_4}^{t_2} = \mathsf{c}_1 \qquad \mathsf{c}_{l_4}^{\mathtt{u}(t_2)} = \mathsf{c}_2$$

Involved locations for insert expressions are illustrated by the following example. Consider the tree t_2 and the update

$$\texttt{insert } \texttt{<}\textit{author}\texttt{/ > into doc/self} :: \textit{bib/book}$$

In this case, the first phase of the evaluation produces a PUL composed of the command $\mathtt{ins}(l', \mathtt{into}, l_1)$ together with the store σ extended with a fresh element node labeled as *author*, which is referenced by a fresh location l'. Then, applying the PUL attaches location l' as children of l_1 (which refers to the book node). The critical location for the PUL is l'. The static analysis for the update over the DTD d_4 returns the chain c = bib.book : author and this is correct since $\mathsf{c}_{l'}^{\mathtt{u}(t_2)} = \mathsf{c}$.

Involved locations for replace expressions are illustrated by the following example. Consider tree t_2 and the update

$$\texttt{replace doc//}\textit{author/name}\ \texttt{with doc/}\textit{publisher/name}$$

here the update first build a fresh tree referenced by l_c, which is a copy of the one rooted at l'_3. Then l_c replaces l'_4 (referencing to the publisher name). This results in the atomic update command

$$\mathtt{repl}(l'_4, l_c)$$

The critical locations are l'_4, l_c and all descendants of l_c. l'_4 is a critical location because it is the target of the replace command. l_c and all of its descendants are critical locations, because l_c is the source locations of the update. We stress on the fact that descendants of l_c are fresh locations as well.

The static analysis wrt the DTD d_4 returns chain c = bib.book.author : name typing the replaced location l'_4. Chain inference outputs also c = bib.book.publisher : name and c = bib.book.publisher : name.String that are the chains for l_c and for the child of l_c.

At this point we can state the correctness of the chain analysis for update expressions. The theorem below states that all locations involved by the update $\mathtt{u}$ are typed by chains inferred from $\mathtt{u}$.

Theorem 4.3.6 (Soundness of Update Chain Inference)**.** *Let* d *be a DTD,* $t = (\sigma, l_t) \in \mathsf{d}$ *a valid tree,* $\mathtt{u}$ *an update,* γ *a dynamic environment and* Γ *a static environment such that* $\sigma \vDash_{\mathsf{d}} \gamma : \Gamma$*. Provided that*

$$\sigma, \gamma \vDash \mathtt{u} \;\Rightarrow\; \sigma_\omega, \omega \qquad \sigma_\omega \vDash \omega \rightsquigarrow \sigma_{\mathtt{u}} \qquad \Gamma \vdash_{\mathsf{C_d}} \mathtt{u} : \mathsf{U}$$

it holds that

- *if* $l \in dom(\sigma)$ *is a location in* t*, and the update* $\mathtt{u}$ *involves* l *then there exists* $\mathsf{c} : \mathsf{c}' \in \mathsf{U}$ *such that* $\mathsf{c}_l^\sigma = \mathsf{c}.\mathsf{c}'$*, where* $\mathsf{c}' \neq \epsilon$*.*
- *if* l *is a location in* $\mathtt{u}(t)$*, i.e.* $l \in dom(\sigma_{\mathtt{u}} @ l_t)$*, and the update* $\mathtt{u}$ *involves* l *then there exists* $\mathsf{c} : \mathsf{c}' \in \mathsf{U}$ *such that* $\mathsf{c}_l^{\sigma_{\mathtt{u}}} = \mathsf{c}.\mathsf{c}'$*, where* $\mathsf{c}' \neq \epsilon$*.*

In the above statement, in case a location l belongs both to t and $\mathtt{u}(t)$, it may be that the chain typing l in t is different from the chain typing l in $\mathtt{u}(t)$ (e.g., due to renaming). Although we made the assumption that the update expression is preserving the schema, it is worth noticing that Theorem 4.3.6 holds also for updates violating schema constraints ($\mathtt{u}(t) \notin \mathsf{d}$), since chains corresponding to deleted or inserted nodes are always traced by the system regardless of correctness wrt the schema.

In this section we showed the correctness of chain inference for query and updates. This will be used in the remaining of the chapter to define a notion of independence, and show its correctness.

4.4 Independence Analysis

In this section we define a notion of query update independence based on chain overlapping. We show that this static notion of independence is correct and implies independence on all schema instances. We show that, for recursive schemas, the notion of independence based on chains does not have a trivial terminating algorithm. To overcome this issue, we propose a finite analysis which is complete wrt the infinite analysis and terminates in all cases.

4.4.1 Infinite Analysis

The notion of query-update independence $\mathtt{q} \perp\!\!\!\perp_{\mathsf{d}} \mathtt{u}$ (Definition 2.5.2) is based on the semantics of $\mathtt{q}$ and $\mathtt{u}$, and involves all possible d instances. The static counterpart of this notion is now proposed and is of course based on query and update chain inference. As chain inference depends on a set C of chains, we first introduce a general static notion of C-independence. Given two sets of chains τ_1 and τ_2, the set of conflicting pairs of chains for τ_1, τ_2 is defined as follows.

$$\mathsf{confl}(\tau_1, \tau_2) \stackrel{def}{=} \{\, (\mathsf{c}_1, \mathsf{c}_2) \mid \mathsf{c}_1 \in \tau_1,\ \mathsf{c}_2 \in \tau_2,\ \mathsf{c}_1 \preceq \mathsf{c}_2 \,\}$$

Definition 4.4.1 (C-independence)**.** *Let* d *be a DTD,* $\mathtt{q}$ *a query and* $\mathtt{u}$ *an update, with at most only one free variable* $\mathtt{x}$*, and* $\Gamma = \{\ \mathtt{x} \mapsto \mathsf{s_d}\ \}$ *a static environment. The query* $\mathtt{q}$ *and the update* $\mathtt{u}$ *are* C*-independent, denoted by* $\mathtt{q} \perp_{\mathsf{C}} \mathtt{u}$*, if provided that* $\Gamma \vdash_{\mathsf{C}} \mathtt{q} : (\mathsf{r}, \mathsf{v}, \mathsf{e})$ *and* $\Gamma \vdash_{\mathsf{C}} \mathtt{u} : \mathsf{U}$ *we have*

$$\mathsf{confl}(\mathsf{r}, \mathsf{U}) = \varnothing \qquad \mathsf{confl}(\mathsf{U}, \mathsf{r}) = \varnothing \qquad \mathsf{confl}(\mathsf{U}, \mathsf{v}) = \varnothing$$

$\mathsf{C_d}$-independence can be decided in a finite amount of time when chain inference produces a finite set of chains i.e. when d has no vertical recursion. The next section provides a technique to capture $\mathsf{C_d}$-independence by means of a finite analysis.

$$\begin{array}{rcl} \mathsf{bib} & \leftarrow & \mathsf{book}* \\ \mathsf{book} & \leftarrow & \mathsf{title}?, \mathsf{author}*, \mathsf{publisher}*, \mathsf{price}? \\ \mathsf{author}, \mathsf{publisher} & \leftarrow & \mathsf{name}* \\ \mathsf{title}, \mathsf{name}, \mathsf{price} & \leftarrow & \mathsf{String}? \end{array}$$

Figure 4.4: DTD d_5

We illustrate the definition of C-independence throughout some examples. We consider DTD d_5 of Figure 4.4. In the queries and updates below, we assume as before that variable `doc` is bound to bib. We check independence of query

$$\begin{array}{l} \mathtt{q} = \ \mathtt{for\ x\ in\ doc}//\mathit{book}\ \mathtt{return} \\ \qquad \mathtt{if\ (x}/\mathit{price}\mathtt{)\ then\ x}/\mathit{author}\ \mathtt{else\ x}/\mathit{publisher} \end{array}$$

against the following updates

$$\begin{array}{ll} \mathtt{u}_1 = & \mathtt{delete\ doc}//\mathit{title} \\ \mathtt{u}_2 = & \mathtt{rename\ doc}//\mathit{publisher}\ \mathtt{as}\ \text{“author”} \\ \mathtt{u}_3 = & \mathtt{insert}\ \text{<}\mathit{price}\text{>}\ 50\ \text{<}\mathit{price}\text{/>}\ \mathtt{into\ doc}//\mathit{book} \\ \mathtt{u}_4 = & \mathtt{replace\ doc}//\mathit{price}/\mathtt{text()\ with}\ \text{“10”} \end{array}$$

The static analysis infers, from the query, $\mathsf{r_q} = \{\ \mathsf{bib.book.author}, \mathsf{bib.book.publisher}\ \}$ and $\mathsf{v_q} = \{\ \mathsf{bib.book}, \mathsf{bib.book.price}\ \}$. The following static independences hold.

$\mathtt{q}$ is $\mathsf{C_{d_5}}$-independent of $\mathtt{u}_1$: the expressions are independent because the update does not delete any node that the query accesses or outputs. The static analysis infers

from the update $\mathsf{U} = \{\ \mathsf{bib.book : title}\ \}$. Indeed, there is no pair of conflicting query and update chains.

query		update
bib.book.price	$\not\geq$	bib.book : title
bib.book.author	$\not\leq \not\geq$	bib.book : title
bib.book.publisher	$\not\leq \not\geq$	bib.book : title

Hence it results that $\mathsf{confl}(\mathsf{r}, \mathsf{U}) = \mathsf{confl}(\mathsf{U}, \mathsf{r}) = \mathsf{confl}(\mathsf{U}, \mathsf{v}) = \varnothing$.

q is not $\mathsf{C}_{\mathsf{d}_5}$-independent of u_2: the expression are conflicting because the update may change the result of the query. The static analysis infers from the update $\mathsf{U} = \{\ \mathsf{bib.book : publisher}, \mathsf{bib.book : author}\ \}$. Indeed, query return-chains conflict with update chains.

query		update
bib.book.publisher	$\preceq$	bib.book : publisher
bib.book.publisher	$\succeq$	bib.book : publisher
bib.book.author	$\preceq$	bib.book : author
bib.book.author	$\succeq$	bib.book : author

Hence it results that $\mathsf{confl}(\mathsf{r}, \mathsf{U}) = \mathsf{confl}(\mathsf{U}, \mathsf{r}) \neq \varnothing$

q is not $\mathsf{C}_{\mathsf{d}_5}$-independent of u_3: the expressions may conflict because the data inserted by the update are accessed by the conditional query. The static analysis infers from the update $\mathsf{U} = \{\ \mathsf{bib.book : price}, \mathsf{bib.book : price.String}\ \}$. Indeed, a used chain conflicts with an update chain.

update		query
bib.book : price.String	$\not\prec$	bib.book.price
bib.book : price	$\preceq$	bib.book.price

Hence it results that $\mathsf{confl}(\mathsf{U}, \mathsf{v}) \neq \varnothing$.

q is $\mathsf{C}_{\mathsf{d}_5}$-independent of u_4: the expressions are independent because the update does not affect the result of the query. In fact, the update impacts the textual node below the price node which is accessed by the query, without changing the boolean value of the conditional expression.

update		query
bib.book.price : String	$\not\prec$	bib.book.price

Hence it results that $\mathsf{confl}(\mathsf{r}, \mathsf{U}) = \mathsf{confl}(\mathsf{U}, \mathsf{r}) = \mathsf{confl}(\mathsf{U}, \mathsf{v}) = \varnothing$.

We conclude this section by formally stating the soundness of the independence analysis based on chains. When C is taken as the set C_d of chains generated for the DTD d, C-independence implies $\perp\!\!\!\perp_\mathsf{d}$ independence, as stated by the following theorem.

Theorem 4.4.2 (Soundness of $\mathsf{C_d}$ independence)**.** *Let* $\mathtt{d}$ *be a DTD,* $\mathtt{q}$ *a query and* $\mathtt{u}$ *an update, with only one free variable. Then,*

$$\mathtt{q} \perp_{\mathsf{C_d}} \mathtt{u} \quad \textit{implies} \quad \mathtt{q} \perp\!\!\!\perp_{\mathtt{d}} \mathtt{u}$$

As already stated, updates are assumed to preserve the schema. The above theorem needs this assumption in order to correctly use *query* chains in the independence analysis. Actually, if deletions violate the schema (a mandatory node is deleted), the notion of static independence $\perp_{\mathsf{C_d}}$ is still sound. The problem comes from insertions creating *new* chains (not belonging to $\mathsf{C_d}$) because they are not considered during chain inference for queries. As a consequence, the analysis made to check $\perp_{\mathsf{C_d}}$ could miss conflicting chains.

In this section we defined a notion of static independence based on chains which is sound. In the next section we will show how independence can be verified when recursive query and schemas may entail the inference of infinite sets of chains.

4.4.2 Finite Analysis

The notion of $\mathsf{C_d}$-independence (Definition 4.4.1) cannot be directly used to define a terminating decision algorithm, because for DTDs with vertical recursion the sets of inferred chains can be infinite. In this section we show how to finitely approximate sets of inferred chains so that $\mathsf{C_d}$-independence can be detected in finite time.

One feature of chains generated by a recursive DTD $\mathtt{d}$ is that some of them contain multiple occurrences of (recursively defined) tags. So one way to characterize a finite set of $\mathtt{d}$-chains is to restrict to chains having at most k occurrences of each tag. Hereafter, these chains are called k-chains, and for any set of chains τ, its subset of k-chains is denoted by τ^k. Thus, $\mathsf{C}^k_{\mathtt{d}}$ denotes the set of k-chains generated by $\mathtt{d}$.

As illustrated next, a *multiplicity value* k can be inferred from the query $\mathtt{q}$ and update $\mathtt{u}$, so that independence according to chains in $\mathsf{C_d}$ is equivalent to independence according to inferred chains in $\mathsf{C}^k_{\mathtt{d}}$. The value k is derived by a two-steps static analysis.

Given an expression $\mathtt{e}$, being either a query $\mathtt{q}$ or an update $\mathtt{u}$, the first step associates a value $k_{\mathtt{e}}$ to $\mathtt{e}$ such that the set of $k_{\mathtt{e}}$-chains inferred for $\mathtt{e}$ is *representative* of all possible inferred chains for $\mathtt{e}$. Intuitively, the representative set of inferred chains for an expression synthesizes all possible inferred chains: any possible inferred chain can be mapped to a chain in the representative set by some folding transformations, according to recursive definitions in the DTD.

The second step infers a value k from the values $k_{\mathtt{q}}$ and $k_{\mathtt{u}}$, such that the search of conflicting chains decisive for statically detecting $\mathtt{q}$-$\mathtt{u}$ independence can be safely done in the finite set of inferred k-chains.

Inferring the values $k_{\mathtt{q}}$ and $k_{\mathtt{u}}$ mainly depends on navigational properties of the XPath expressions occurring in the query and update. Thus, we start the discussion by focusing on XPath expressions `p`, and then consider FLWR expressions.

In the discussion, most of the examples are made on the following recursive DTD d_6. As before, variable `doc` in the expressions is meant to be bound to the root of d_6, that is store.

$$\begin{array}{rcl} \mathsf{store} & \leftarrow & \mathsf{component}* \\ \mathsf{component} & \leftarrow & \mathsf{description}, \mathsf{complist}?, \mathsf{uselist}? \\ \mathsf{uselist} & \leftarrow & \mathsf{component}+ \\ \mathsf{complist} & \leftarrow & \mathsf{component}+ \\ \mathsf{description} & \leftarrow & \mathsf{String} \end{array}$$

The inference of the k value depends on the navigational axes employed in the XPath steps.

Dealing with child, self and parent

The case where paths employ only the child, self or parent axis is a special one. In fact, such paths always generate a finite number of chains, hence a finite number of representative chains, and no folding is needed. What remains to do is to find a k value that fits all representative chains. The particularity of these paths is that they explicitly indicate how many times a single tag has to be matched: a good choice for $k_{\mathtt{p}}$ is the maximal tag frequency in the path `p`.

To illustrate, consider the following downward path `p`

$$\mathtt{doc/self} :: \mathit{store/component/complist/component/description}$$

there are four tags in `p`, all but *component* have frequency 1, while *component* has frequency 2. Therefore, the maximal tag frequency is 2, and indeed 2-chains include the representative chain

$$\mathsf{store}.\overline{\mathsf{component}}.\mathsf{complist}.\overline{\mathsf{component}}.\mathsf{description}$$

which is the only chain inferred for this path (for clarity, we over line the tag with k-frequency).

The same holds for the navigational path with backward axis

$$\mathtt{doc/self} :: \mathit{store/component/complist/component/}\mathtt{parent} :: \mathit{complist}$$

Here, the chains inferred from the path are 2-chains

$$\begin{array}{l} \mathsf{store}.\overline{\mathsf{component}}.\mathsf{complist}.\overline{\mathsf{component}} \\ \mathsf{store}.\overline{\mathsf{component}}.\mathsf{complist}.\overline{\mathsf{component}}.\mathsf{complist} \end{array}$$

Similarly, for the path

$$\texttt{doc/self} :: \mathit{store}/\mathit{component}/\mathit{complist}/*$$

we infer $k_{\mathtt{p}} = 2$, since the wildcard $*$ stands for any label.

It is worth noticing that, for the analysis to be correct, the query/update free variables have to be always bound to the schema root type. If we relax this assumption, the chain analysis may turn out to be unsound for recursive schemas. To see this, consider for instance the query

$$\texttt{doc}/\mathit{component}$$

with doc initially bound to the chain store.component.description. The chain inferred for the query is store.component.description.component, which is a 2-chain. However, since for this expression $k = 1$, the analysis infers the empty set of used, return and element chains for the query, thus deeming it as independent of any update, while it should not.

Dealing with descendant and ancestor

When a path p makes use of the descendant axis, the number of chains inferred for an expression may not be finite, and the length of inferred chains totally unrelated to the length of p. For instance, the path

$$\texttt{doc/descendant} :: \mathit{description}$$

makes the system inferring an infinite number of chains, of unbounded length.

This is what led us to reason in terms of tag frequency rather than path length. To generate a finite set of chains, the value $k_{\mathtt{p}}$ is determined by taking into account the number of occurrences of descendant axes in p.

To illustrate this, we still consider the schema d_6, and observe that the type component is defined in terms of description, uselist and complist, and the two latter are defined in terms of component. These mutual recursive definitions entail that, in a valid document instance, a node with type uselist can be a descendant of a node with type complist, and vice versa, along the same chain of the tree. In addition, a chain connecting uselist and complist nodes always contains an intermediate component label, which also occurs before the first occurrence of a description, uselist, and complist label.

As a consequence, for the following path

$$\mathtt{p} = \texttt{doc/descendant} :: \mathit{complist}/\texttt{descendant} :: \mathit{uselist}/\texttt{descendant} :: \mathit{description}$$

the shortest chain that is inferred is the 3-chain

$$\mathsf{store}.\overline{\mathsf{component}}.\mathsf{complist}.\overline{\mathsf{component}}.\mathsf{uselist}.\overline{\mathsf{component}}.\mathsf{description}$$

Simple tag frequency, like for the previous cases, would lead to $k_{\mathtt{p}} = 1$. This is not satisfactory because no 1-chain is inferred for `p` . To reflect the fact that each recursive axis may permit any tag to repeat once in inferred chains, the correct maximal tag frequency we have to consider for the path `p` is 3; in fact, 3-chains do allow to infer a non-empty set of representative chains - other inferred chains can be obtained by unfolding recursion.

Of course, an XPath expression may combine both recursive and non-recursive axes. In this case, for a path `p` we obtain $k_{\mathtt{p}}$ as the sum of two components computed independently: the maximal tag frequency for non-recursive steps, and the number of recursive steps in `p`. As an example, for

$$\mathtt{p} = \mathtt{doc/descendant} :: \mathit{component}/\mathit{uselist}/\mathit{component}$$

we have $k_{\mathtt{p}} = 2$ since the maximal tag frequency for the path suffix $/\mathit{uselist}/\mathit{component}$ is 1, and there is 1 descendant step $/\mathtt{descendant} :: \mathit{component}$.

Recursive backward axes are handled similarly. Here we have to pay attention to the fact that chains navigated by an `ancestor` step are prefixes of some chains generated by previous steps. Consider

$$\mathtt{p} = \mathtt{doc/descendant} :: \mathit{uselist}/\mathtt{ancestor} :: \mathit{complist}$$

Here $k_{\mathtt{p}}$ has to be such that the *used* chain

$$\mathsf{store}.\overline{\mathsf{component}}.\mathsf{complist}.\overline{\mathsf{component}}.\mathsf{uselist}$$

can be generated. Thus, we enforce `ancestor` steps to increment the tag frequency by 1. This is reminiscent of what we have seen before in the case of `descendant`; the way `p` is processed can be compared to the way would be processed the navigational path

$$\mathtt{doc/descendant} :: \mathit{complist}/\mathtt{descendant} :: \mathit{uselist}$$

and therefore chains containing complist as ancestor of uselist need to be generated for `p`.

Concerning paths `p` employing either `descendant-or-self` or `ancestor-or-self`, $k_{\mathtt{p}}$ is computed as for the self-less axes. The same holds for shortcut expressions like $\mathtt{p} = //\phi$, that stands for $\mathtt{p}' = /\mathtt{descendant\text{-}or\text{-}self} :: */\phi$, when $\phi \neq *$. It follows that $k_{\mathtt{p}} = k_{\mathtt{p}'}$ simply because the sets of chains inferred for the two expressions are exactly the same.

It is worth noticing that by assuming that the root-type of a DTD is non-recursive we can provide a uniform the definition of the k value, and simplify proofs. To illustrate, consider the DTD $\mathtt{d} = \{\ \mathtt{a} \leftarrow \mathtt{a}?\ \}$, the query $\mathtt{q} = \mathtt{x/child} :: a$, and the static environment $\Gamma = \{\ \mathtt{x} \mapsto \mathtt{a}\ \}$. Our system infers from `q` the 2-chain `a.a`. However, since $k_{\mathtt{q}} = \mathcal{F}(\mathtt{a}, \mathtt{x/child} :: \mathtt{a}) = 1$ the chain `a.a` is ruled-out from inference, while it should not. The same happens if $\mathtt{q} = \mathtt{x/descendant} :: \mathtt{a}$. This of course can be captured by setting

$\mathcal{F}(\mathtt{a}, \mathtt{x/axis} :: \phi) = 2$ if i) $\mathtt{x}$ is bound to the singleton root-type, ii) the root-type is recursive, iii) the root-type satisfies ϕ, and of course iv) `axis` is downward. This would make the formal definition for the base case quite inelegant, while assuming a non-recursive root-type we have an uniform formalization of the tag frequency. This discrepancy is also caused by the fact that, as discussed in Chapter 2, we do not have the document-node in our data-model.

Dealing with sibling axes

Sibling axes are managed as child and parent axes. Consider the following recursive DTD d_7

$$\begin{array}{rcl} \mathsf{store} & \leftarrow & \mathsf{component}* \\ \mathsf{component} & \leftarrow & \mathsf{description}, \mathsf{component}? \\ \mathsf{description} & \leftarrow & \mathsf{String} \end{array}$$

and the navigational path

$$\mathtt{doc/descendant} :: \mathit{description}/\mathtt{following\text{-}sibling} :: \mathit{component}$$

For this path, the used 1-chain $\mathsf{store.component.description}$ and the return 2-chain

$$\mathsf{store}.\overline{\mathsf{component}}.\overline{\mathsf{component}}$$

are the needed chains. The presence of the recursive step $/\mathtt{descendant} :: \mathit{description}$ entails $k = 2$.

Dealing with FLWR expressions and Updates

Based on concepts previously illustrated, we provide now formal definitions to deal with the general case of a FLWR expression $\mathtt{e}$.

As seen before, the computation of $k_{\mathtt{e}}$ is decomposed into two tasks. The first one determines via the function $\mathcal{F}(a, \mathtt{e})$ the frequency of each tag $a \in \Sigma$ on the whole expression, in order to derive the maximal frequency. The second task computes via the function $\mathcal{R}(\mathtt{e})$ the maximal number of recursive steps used by a navigational path embodied in the whole expression. The value $k_{\mathtt{e}}$ is the sum of these two values. Formally:

$$k_{\mathtt{e}} \stackrel{def}{=} \max\{\, \mathcal{F}(a, \mathtt{e}) \mid a \in \Sigma \,\} + \mathcal{R}(\mathtt{e})$$

The functions $\mathcal{F}(a, \mathtt{e})$ and $\mathcal{R}(\mathtt{e})$ are defined by structural induction in Table 4.3 and 4.4, respectively.

When $\mathtt{e}$ is a for or a let expression, the value $k_{\mathtt{e}}$ is specified by summing the subexpression values. This is captured by rules ($\mathcal{F}$-For/Let) and ($\mathcal{R}$-For/Let). This

$$\frac{}{\mathcal{F}(a,()) = 0}(\mathcal{F}\text{-Empty}) \qquad \frac{}{\mathcal{F}(a,\text{“txt”}) = 0}(\mathcal{F}\text{-Text}) \qquad \frac{f = \mathcal{F}(a,\mathtt{q})}{\mathcal{F}(a,\texttt{<a>q}) = f + 1}(\mathcal{F}\text{-Element})$$

$$\frac{\texttt{axis is recursive}}{\mathcal{F}(a,\mathtt{axis} :: \phi) = 0}(\mathcal{F}\text{-Step1}) \qquad \frac{\phi \notin \{a,\mathtt{node}()\}}{\mathcal{F}(a,\mathtt{axis} :: \phi) = 0}(\mathcal{F}\text{-Step2}) \qquad \frac{\texttt{axis is not recursive} \quad \phi \in \{a,\mathtt{node}()\}}{\mathcal{F}(a,\mathtt{axis} :: \phi) = 1}(\mathcal{F}\text{-Step3})$$

$$\frac{f_1 = \mathcal{F}(a,\mathtt{q}_1) \quad f_2 = \mathcal{F}(a,\mathtt{q}_2)}{\mathcal{F}(a,\mathtt{q}_1,\mathtt{q}_2) = \max(f_1,f_2)}(\mathcal{F}\text{-Concat}) \qquad \frac{f_0 = \mathcal{F}(a,\mathtt{q}_0) \quad f_1 = \mathcal{F}(a,\mathtt{q}_1) \quad f_2 = \mathcal{F}(a,\mathtt{q}_2)}{\mathcal{F}(a,\texttt{if (}\mathtt{q}_0\texttt{) then }\mathtt{q}_1\texttt{ else }\mathtt{q}_2) = \max(f_0,f_1,f_2)}(\mathcal{F}\text{-If})$$

$$\frac{f_1 = \mathcal{F}(a,\mathtt{q}_1) \quad f_2 = \mathcal{F}(a,\mathtt{q}_2)}{\mathcal{F}(a,\texttt{for x in }\mathtt{q}_1\texttt{ return }\mathtt{q}_2) = f_1 + f_2}(\mathcal{F}\text{-For}) \qquad \frac{f_1 = \mathcal{F}(a,\mathtt{q}_1) \quad f_2 = \mathcal{F}(a,\mathtt{q}_2)}{\mathcal{F}(a,\texttt{let x := }\mathtt{q}_1\texttt{ return }\mathtt{q}_2) = f_1 + f_2}(\mathcal{F}\text{-Let})$$

$$\frac{f = \mathcal{F}(a,\mathtt{q})}{\mathcal{F}(a,\texttt{delete q}) = f}(\mathcal{F}\text{-Delete})$$

$$\frac{f = \mathcal{F}(a,\mathtt{q})}{\mathcal{F}(a,\texttt{rename q as }a) = 1 + f}(\mathcal{F}\text{-Rename1}) \qquad \frac{f = \mathcal{F}(a,\mathtt{q}) \quad a \neq b}{\mathcal{F}(a,\texttt{rename q as }b) = f}(\mathcal{F}\text{-Rename2})$$

$$\frac{f = \mathcal{F}(a,\mathtt{q}) \quad f_0 = \mathcal{F}(a,\mathtt{q}_0)}{\mathcal{F}(a,\texttt{insert q pos }\mathtt{q}_0) = f + f_0}(\mathcal{F}\text{-Insert}) \qquad \frac{f_0 = \mathcal{F}(a,\mathtt{q}_0) \quad f = \mathcal{F}(a,\mathtt{q})}{\mathcal{F}(a,\texttt{replace }\mathtt{q}_0\texttt{ with posq}) = f_0 + f}(\mathcal{F}\text{-Replace})$$

$$\frac{f_1 = \mathcal{F}(a,\mathtt{u}_1) \quad f_2 = \mathcal{F}(a,\mathtt{u}_2)}{\mathcal{F}(a,\mathtt{u}_1,\mathtt{u}_2) = \max(f_1,f_2)}(\mathcal{F}\text{-UConcat}) \qquad \frac{f_0 = \mathcal{F}(a,\mathtt{q}_0) \quad f_1 = \mathcal{F}(a,\mathtt{u}_1) \quad f_2 = \mathcal{F}(a,\mathtt{u}_2)}{\mathcal{F}(a,\texttt{if (}\mathtt{q}_0\texttt{) then }\mathtt{u}_1\texttt{ else }\mathtt{u}_2) = \max(f_0,f_1,f_2)}(\mathcal{F}\text{-UIf})$$

$$\frac{f = \mathcal{F}(a,\mathtt{q}) \quad f' = \mathcal{F}(a,\mathtt{u})}{\mathcal{F}(a,\texttt{for x in q return u}) = f + f'}(\mathcal{F}\text{-UFor}) \qquad \frac{f = \mathcal{F}(a,\mathtt{q}) \quad f' = \mathcal{F}(a,\mathtt{u})}{\mathcal{F}(a,\texttt{let x := q return u}) = f + f'}(\mathcal{F}\text{-ULet})$$

Table 4.3: $\mathcal{F}$ definition.

$$\frac{}{\mathcal{R}(()) = 0}(\mathcal{R}\text{-Empty}) \qquad \frac{}{\mathcal{R}(\text{“}\texttt{txt}\text{”}) = 0}(\mathcal{R}\text{-Text}) \qquad \frac{r = \mathcal{R}(\mathtt{q})}{\mathcal{R}(\texttt{<}a\texttt{>}\mathtt{q}\texttt{</}a\texttt{>}) = r}\ (\mathcal{R}\text{-Element})$$

$$\frac{\texttt{axis is recursive}}{\mathcal{R}(\texttt{axis} :: \phi) = 1}(\mathcal{R}\text{-Step1}) \qquad \frac{\texttt{axis is not recursive}}{\mathcal{R}(\texttt{axis} :: \phi) = 0}\ (\mathcal{R}\text{-Step2})$$

$$\frac{r_1 = \mathcal{R}(\mathtt{q}_1) \quad r_2 = \mathcal{R}(\mathtt{q}_2)}{\mathcal{R}(\mathtt{q}_1, \mathtt{q}_2) = \max(r_1, r_2)}\ (\mathcal{R}\text{-Concat}) \qquad \frac{r_0 = \mathcal{R}(\mathtt{q}_0) \quad r_1 = \mathcal{R}(\mathtt{q}_1) \quad r_2 = \mathcal{R}(\mathtt{q}_2)}{\mathcal{R}(\texttt{if (}\mathtt{q}_0\texttt{) then }\mathtt{q}_1\texttt{ else }\mathtt{q}_2) = \max(r_0, r_1, r_2)}\ (\mathcal{R}\text{-If})$$

$$\frac{r_1 = \mathcal{R}(\mathtt{q}_1) \quad r_2 = \mathcal{R}(\mathtt{q}_2)}{\mathcal{R}(\texttt{for x in }\mathtt{q}_1\texttt{ return }\mathtt{q}_2) = r_1 + r_2}\ (\mathcal{R}\text{-For}) \qquad \frac{r_1 = \mathcal{R}(\mathtt{q}_1) \quad r_2 = \mathcal{R}(\mathtt{q}_2)}{\mathcal{R}(\texttt{let x := }\mathtt{q}_1\texttt{ return }\mathtt{q}_2) = r_1 + r_2}\ (\mathcal{R}\text{-Let})$$

$$\frac{r = \mathcal{R}(\mathtt{q})}{\mathcal{R}(\texttt{delete }\mathtt{q}) = r}(\mathcal{R}\text{-Delete}) \qquad \frac{r = \mathcal{R}(\mathtt{q})}{\mathcal{R}(\texttt{rename }\mathtt{q}\texttt{ as }b) = r}(\mathcal{R}\text{-Rename})$$

$$\frac{r = \mathcal{R}(\mathtt{q}) \quad r_0 = \mathcal{R}(\mathtt{q}_0)}{\mathcal{R}(\texttt{insert }\mathtt{q}\texttt{ pos }\mathtt{q}_0) = r + r_0}\ (\mathcal{R}\text{-Insert}) \qquad \frac{r_0 = \mathcal{R}(\mathtt{q}_0) \quad r = \mathcal{R}(\mathtt{q})}{\mathcal{R}(\texttt{replace }\mathtt{q}_0\texttt{ with posq}) = r_0 + r}\ (\mathcal{R}\text{-Replace})$$

$$\frac{r_1 = \mathcal{R}(\mathtt{u}_1) \quad r_2 = \mathcal{R}(\mathtt{u}_2)}{\mathcal{R}(\mathtt{u}_1, \mathtt{u}_2) = \max(r_1, r_2)}\ (\mathcal{R}\text{-UConcat}) \qquad \frac{r_0 = \mathcal{R}(\mathtt{q}_0) \quad r_1 = \mathcal{R}(\mathtt{u}_1) \quad r_2 = \mathcal{R}(\mathtt{u}_2)}{\mathcal{R}(\texttt{if (}\mathtt{q}_0\texttt{) then }\mathtt{u}_1\texttt{ else }\mathtt{u}_2) = \max(r_0, r_1, r_2)}\ (\mathcal{R}\text{-UIf})$$

$$\frac{r = \mathcal{R}(\mathtt{q}) \quad r' = \mathcal{R}(\mathtt{u})}{\mathcal{R}(\texttt{for x in q return u}) = r + r'}(\mathcal{R}\text{-UFor}) \qquad \frac{r = \mathcal{R}(\mathtt{q}) \quad r' = \mathcal{R}(\mathtt{u})}{\mathcal{R}(\texttt{let x := q return u}) = r + r'}\ (\mathcal{R}\text{-ULet})$$

Table 4.4: $\mathcal{R}$ definition.

is motivated by the fact that, for instance, for-expressions are usually used to encode nested iterations performed by XPath paths, like in the query

$$\texttt{for x in doc/}a\ \texttt{for y in x/}b\ \texttt{return y}$$

This definition leads in some cases to an overestimation of the value $k_{\mathtt{e}}$ that would be actually sufficient for a finite analysis. For instance, for the query

$$\mathtt{q} = \texttt{for x in doc/}a\texttt{/}a\ \texttt{for y in doc/}a\texttt{/}b\ \texttt{return x,y}$$

we have $\mathcal{F}(a, \mathtt{q}) = 3$, while the value 2 would be sufficient. It is worth saying that this overestimation raises the $k_{\mathtt{q}}$ value. More precision can be obtained by tracing variable bindings in the definition of $\mathcal{F}$. The same argument holds for $\mathcal{R}$. However, this would make the formalization cumbersome without being a decisive factor for the analysis. Thus, our choice has been guided by simplicity and conciseness of $\mathcal{F}$ and $\mathcal{R}$ definitions.

The rules for element construction, ($\mathcal{F}$-Element) and ($\mathcal{R}$-Element), deserve some comment. Note that tags of constructed elements are taken into account. Indeed, these elements can be inserted by an update as children of existing elements, thus generating new chains pointing nodes that can be accessed by a query. Consider the recursive schema $\mathtt{d}_7$ and the following update $\mathtt{u}$.

```
for x in doc/self :: store/component return
  insert
    <component>
      <component>
        <description/>
      </component>
    </component>
  into x
```

As already outlined, precision of the independence analysis relies (among other things) on the chains generated for element construction. The rules in Table 4.3 lead to infer $k_{\mathtt{u}} = \mathcal{F}(\mathtt{u}, component) = 3$, and thus the chain

$$\mathsf{store}.\overline{\mathsf{component}} : \overline{\mathsf{component}}.\overline{\mathsf{component}}.\mathsf{description}$$

is inferred for the finite analysis. The inferred value is the same, if $\mathtt{u}$ is a replace update instead of an insert update.

Tag frequency for delete update expression is determined as for queries. For rename expressions tag frequency is determined by observing that for recursive schemas, after renaming, the frequency of the renaming label may increase of 1. To illustrate, consider the DTD $\mathtt{d}_7$ update

$$\mathtt{u} = \texttt{rename doc//}component\ \texttt{as}\ \text{“}component\text{”}$$

In this case, if we settle $\mathcal{F}(\mathtt{u}, component) = 1$ to capture the fact that a new node labeled with *component* appears. Together with $\mathcal{R}(//description) = 1$, this makes $k_{\mathtt{u}} = 2$, and indeed this is the multiplicity value needed to infer the 2-chains

$$\mathsf{store}.\overline{\mathsf{component}} : \mathsf{description} \quad \text{and} \quad \mathsf{store}.\overline{\mathsf{component}} : \overline{\mathsf{component}}$$

Notice that if the frequency of *component* is not kept into account then $k_{\mathtt{u}} = 1$ and chain $\mathsf{store}.\overline{\mathsf{component}} : \overline{\mathsf{component}}$ is wrongly excluded from inference.

Finite independence analysis

We see now how to use the values $k_{\mathtt{q}}$ and $k_{\mathtt{u}}$ in order to determine a k value such that C_{d}-independence can be detected by restricting the analysis to k-chains.

Consider the following query and update pair over the DTD d_6.

$$\mathtt{q} = \mathtt{/descendant} :: complist \qquad \mathtt{u} = \mathtt{delete\ /descendant} :: uselist$$

The two expressions are dependent because *uselist* is an ancestor of *complist* and they are both navigated by a recursive axis. It results that $k_{\mathtt{q}} = 1$ and $k_{\mathtt{u}} = 1$. At this point, we could argue that a sound choice is

$$k = max(k_{\mathtt{q}}, k_{\mathtt{u}})$$

which allows the finite analysis to infer the query chain

$$\mathsf{store.component.complist}$$

and the update chain

$$\mathsf{store.component} : \mathsf{uselist}$$

Unfortunately, these chains do not conflict, and rule out dependence. The problem here comes from the fact that the update may change a descendant of a query returned node, and that $k = max(k_{\mathtt{q}}, k_{\mathtt{u}})$ does not permit to capture this in the finite analysis. It may also occur that a query navigates descendants of nodes inserted/created by the update, and again such a definition rules out independence. To avoid this problem, it is necessary that representative chains that are inferred for the update cover query returned nodes, and vice versa. To this end, while inferring chains for the update $\mathtt{u}$, structural properties of the query $\mathtt{q}$ have also to be taken into account. This is obtained by setting

$$k = k_{\mathtt{q}} + k_{\mathtt{u}}$$

Now with $k = 2$ it is possible to infer 2-chains witnessing the conflict such as

$$\mathsf{store.component.complist} \ \preceq \ \mathsf{store.component.complist.component} : \mathsf{uselist}$$

4.4.3 Soundness of the Finite Chain Analysis

In this section we state one of our main results, soundness of the finite analysis.

Theorem 4.4.3 (Soudness of C_d^k Independence)**.** *Let* d *be a DTD,* $\mathtt{q}$ *a query and* $\mathtt{u}$ *an update, with only one free variable. Let* $k = k_\mathtt{q} + k_\mathtt{u}$ *as defined above, then*

$$\mathtt{q} \perp_{\mathsf{C}_\mathsf{d}^k} \mathtt{u} \quad \textit{implies} \quad \mathtt{q} \perp\!\!\!\perp_\mathsf{d} \mathtt{u}$$

We develop the main steps of the proof of the Theorem. The thesis follows by showing that $\mathtt{q} \perp_{\mathsf{C}_\mathsf{d}} \mathtt{u}$ is equivalent to $\mathtt{q} \perp_{\mathsf{C}_\mathsf{d}^k} \mathtt{u}$. Proving that $\mathtt{q} \perp_{\mathsf{C}_\mathsf{d}} \mathtt{u}$ implies $\mathtt{q} \perp_{\mathsf{C}_\mathsf{d}^k} \mathtt{u}$ is straightforward since $\mathsf{C}_\mathsf{d}^k \subseteq \mathsf{C}_\mathsf{d}$. Also, we reason in terms of *dependence*, rather than independence. We prove that C_d-dependence implies C_d^k-dependence (these notions directly follow from Definition 4.4.1) by showing that from any pair of chains in C_d, witness of dependence, it is possible to identify a pair of k-chains in C_d^k, witness of dependence.

The proof is composed of three steps. First, we show that there exists a folding from query chains to k-query-chains, for any query $\mathtt{q}$ (Lemma 4.4.4). Then, we show that there exists a folding from query chains to k-query-chains also preserving the prefix relation $\preceq$, for any pair of queries $(\mathtt{q}, \mathtt{q}')$ (Lemma 4.4.5). Finally, we show that such a folding exists for chains inferred for any query-update pair $(\mathtt{q}, \mathtt{u})$ (Theorem 4.4.3). Proofs are reported in Chapter 9.

Given a DTD d, we define a folding relation $\hookrightarrow_\mathsf{d} \subseteq \mathsf{C}_\mathsf{d} \times \mathsf{C}_\mathsf{d}$ as

$$\hookrightarrow_\mathsf{d} \stackrel{def}{=} \{ (\mathsf{c}_1, \mathsf{c}_2) \mid \mathsf{c}_1 = \mathsf{c.a.c'a.c''} \ \wedge \ \mathsf{c}_2 = \mathsf{c.a.c''} \}$$

Notice that, above, the symbol a is a recursive type of the schema. We denote by $\hookrightarrow_\mathsf{d}^*$ the reflexive and transitive closure of $\hookrightarrow_\mathsf{d}$.

We state now the correctness of the k-folding for a single query. In the statements below we consider that, if c is a used (respectively return or element) chain, then c' is a used (respectively return or element) chain.

Lemma 4.4.4 (Folding)**.** *Let* d *be a DTD,* $\mathtt{q}$ *a query with only one free variable* $\mathtt{x}$*, and* $\Gamma = \{ \mathtt{x} \mapsto \mathsf{s}_\mathsf{d} \}$ *a static environment. Assume that* $\Gamma \vdash_{\mathsf{C}_\mathsf{d}} \mathtt{q} : (\mathsf{r}, \mathsf{v}, \mathsf{e})$ *and* $\tau = \mathsf{r} \cup \mathsf{v} \cup \mathsf{e}$*. For each chain* $\mathsf{c} \in \tau$ *there exists a chain* $\mathsf{c}' \in \tau$ *such that* $\mathsf{c} \hookrightarrow_\mathsf{d}^* \mathsf{c}'$ *and* c' *is a* $k_\mathtt{q}$*-chain.*

As an example, consider DTD d_6 and the query $\mathtt{q} = \mathtt{//}\textit{component}$. In this case $k_\mathtt{q} = 1$ and we have, for instance, the following 2-chain folding onto a 1-chain.

$$\mathsf{store.component.uselist.component} \quad \hookrightarrow_\mathsf{d} \quad \mathsf{store.component}$$

When $\mathtt{q}$ and $\mathtt{u}$ are C_d-dependent, at least one of $\mathsf{confl}(\mathsf{U},\mathsf{v})$, $\mathsf{confl}(\mathsf{r},\mathsf{U})$, $\mathsf{confl}(\mathsf{U},\mathsf{r})$ is nonempty (see Definition 4.4.1). This implies that there exists a conflicting pair of inferred chains, witness of the C_d-dependence of $\mathtt{q}$ and $\mathtt{u}$. As updates are defined *in terms of* queries, the next lemma which focuses on "conflicting" query chains is needed to conclude the proof of Theorem 4.4.3.

Lemma 4.4.5 (Folding and Conflict Preservation)**.** *Let* d *be a DTD,* $\mathtt{q}_1, \mathtt{q}_2$ *two queries with only one free variable* $\mathtt{x}$*, and* $\Gamma = \{\ \mathtt{x} \mapsto \mathsf{s}_\mathsf{d}\ \}$ *a static environment. Assume that* $\Gamma \vdash_{\mathsf{C}_\mathsf{d}} \mathtt{q}_i : (\mathsf{r}_i, \mathsf{v}_i, \mathsf{e}_i)$ *and* $\tau_i = \mathsf{r}_i \cup \mathsf{v}_i \cup \mathsf{e}_i$*. For each pair of chains* $(\mathsf{c}_1, \mathsf{c}_2) \in \tau_1 \times \tau_2$ *such that* $\mathsf{c}_1 \preceq \mathsf{c}_2$*, there exists* $(\mathsf{c}'_1, \mathsf{c}'_2) \in \tau_1 \times \tau_2$ *such that* $\mathsf{c}'_1 \preceq \mathsf{c}'_2$ *and* $\mathsf{c}_i \hookrightarrow^*_\mathsf{d} \mathsf{c}'_i$ *with* c'_i *a* $(k_{\mathtt{q}_1}{+}k_{\mathtt{q}_2})$*-chain* $(i = 1, 2)$*.*

As an example, consider DTD d_6 and the query $\mathtt{q}$ = *//uselist* and the update $\mathtt{u}$ = `delete` *//complist*. In this case $k = k_\mathtt{q} + k_\mathtt{u} = 2$ and and we have, for instance, the following conflicting chains folding onto k-chains, and preserving the conflict. Below, for the sake of concision, st = store, co = component, ul = uselist, and cl = complist.

$$\begin{array}{ccc} \mathsf{st.co.ul.co.ul.co.ul} & \hookrightarrow^*_\mathsf{d} & \mathsf{st.co.ul} \\ \preceq & & \preceq \\ \mathsf{st.co.ul.co.ul.co.ul.co.cl} & \hookrightarrow^*_\mathsf{d} & \mathsf{st.co.ul.co.cl} \end{array}$$

While, if we have that $k < 2$ then the folding does not preserve the conflicts.

$$\begin{array}{ccc} \mathsf{st.co.ul.co.ul.co.ul} & \hookrightarrow^*_\mathsf{d} & \mathsf{st.co.ul} \\ \preceq & & \not\preceq \\ \mathsf{st.co.ul.co.ul.co.ul.co.cl} & \hookrightarrow^*_\mathsf{d} & \mathsf{st.co.cl} \end{array}$$

Conclusions

In this chapter we presented two of the main contributions of this thesis: a type system for chain inference and a static notion of independence based on chains. We proved also that both the type system and the independence analysis are correct. Moreover, we illustrated how to make the static notion of independence verifiable in a finite time, even in the case where inference entails an infinite number of query and update chain. In the next chapter we will discuss algorithms and data structures for efficiently implementing the chain-based independence.

Chapter 5

Implementing the Chain-based Independence Analysis

In this chapter we present algorithms for efficiently implementing the chain-based independence analysis. Chain inference poses significant challenges from the computational point of view. Because of the expressivity of the schema and of the query/update languages considered, it is easy to have exponential blowups of chains during inference. In light of this, we present a data-structure for storing chains based on DAGs, and algorithms implementing the chain-based independence analysis, that allow to run the independence analysis in polynomial space and time, whilst still retaining an high precision.

5.1 Introduction

Chain inference is a challenging problem not only from a formal point of view, but also from a computational point of view. In fact, the number of distinct chains inferred for a single expression may be exponential in the size of the input DTD. This happens for DTDs that make heavy use of recursive definitions, but also for non-recursive ones, like for DTD d_8, that is defined by the following set of productions.

$$\mathsf{a} \leftarrow \mathsf{b}_1, \mathsf{f}_1 \qquad \ldots \qquad \mathsf{b}_{(i-1)} \leftarrow (\mathsf{b}_i\,,\, \mathsf{f}_i)? \qquad \mathsf{f}_{(i-1)} \leftarrow (\mathsf{b}_i\,,\, \mathsf{f}_i)?$$

with $1 \leq i \leq n$.

The DTD d_8 employes $2n+1$ types, and a simple navigational query like $//b_n$ makes the system inferring $\Omega(2^n)$ chains. Namely, all chains of the form

$$\mathsf{a}\ x_1\ x_2\ \ldots x_{(n-1)}\ \mathsf{b}_n$$

with $x_i \in \{\mathsf{b}_i, \mathsf{f}_i\}$. This happens because types are heavily reused in the schema. If no type is reused, it turns out that, the DTD is non-recursive and the number of chains becomes linear in the size of the schema.

This already demonstrates the expressive power of the constraints that can be defined by a non-recursive DTD. The situation is even exacerbated by the use of recursion, where it is possible to reproduce the previous situation with a schema featuring just three types. Consider the DTd d_9 defined as follows

$$\begin{array}{lcl} \mathsf{a} & \leftarrow & \mathsf{b},\mathsf{f} \\ \mathsf{b} & \leftarrow & (\mathsf{b}\,,\,\mathsf{f})? \\ \mathsf{f} & \leftarrow & (\mathsf{b}\,,\,\mathsf{f})? \end{array}$$

In this case, the chains of length $h+1$ inferred for a navigational step like $//b$, are again all chains of the form $\mathsf{a}\,x_1\,x_2\,\ldots\,x_{(h-1)}\,\mathsf{b}$ with $x_i \in \{\mathsf{b},\mathsf{f}\}$. Again, we have an exponential blowup since the cardinality of the result is in $\Omega(2^h)$. As we show later (see Proposition 5.1.5), when the multiplicity value k is considered instead of the chain length h, the quantity of inferred chains is still exponential.

These examples outline the hardness of chain inference already in terms of cardinality of the results. If chains are not stored efficiently, this may have a direct consequence on space occupancy. This is problematic because if the space complexity for intermediate computations has an exponential upper-bound, then time complexity is at least of the same order, thus compromising tractability of the analysis.

In the remainder of the chapter we will see that the crux in implementing the chain analysis is the choice of the data-structure to use for efficiently storing and retrieving chains. Unsurprisingly, the blowups outlined in the previous example imply that storing chains requires exponential space for instance if we use a trivial data-structure such as lists, or a less trivial one, such as trees. In light of this, we propose a compressed data structure based on DAGs that permits to store chains in polynomial space, paving the way to the tractability of the analysis.

Cardinality of Schema Chains

In this section, we investigate the cardinality of the set of chains induced by a schema, and relative issues concerning their space occupancy. We study both non-recursive and recursive DTDs, and storing chains on lists and trees. Our results show clearly that the static notion of independence we defined in Chapter 4 may require an exponential time to be checked.

Recall that given two functions f and g, we say that f is in $O(g)$ or, equivalently, that g is in $\Omega(f)$ if $f \leq Cg + C'$, for two positive constants C, C' and all values of the variables. We say that f is in $\Theta(g)$ if f is in $O(g)$ and f is in $\Omega(g)$.

The cases when our system produces the larger number of chains are those relative to saturated schema (defined in Chapter 2).

Proposition 5.1.1. *For a saturated non-recursive DTD* d *with* n *types, the cardinality of* $\mathsf{C_d}$ *is* $\Theta(2^n)$.

Proof. Because d is non-recursive and saturated, $\Rightarrow_\mathsf{d}$ is a strict total order. It is total, because all pairs of types in d are adjacent in $\mathcal{G}_\mathsf{d}$. It is strict, because d is non-recursive, and thus $\Rightarrow_\mathsf{d}$ is irreflexive. This implies that any *subset* of types in d together with the order of $\Rightarrow_\mathsf{d}$ corresponds to a chain in $\mathsf{C_d}$. The size of $\mathsf{C_d}$ comes from the size of the powerset of types in d. This is $2^n - 1$, without counting the empty chain. Moreover, the number of *rooted chains* induced by d coincides with the number of subsets of types of d that contain at least the root-type. These are exactly 2^{n-1}. □

We compare the case of using *lists* and *prefix-trees* for storing chains.

List Lists are defined as sequences of objects. Storing chains on lists implies to store each time the full chain. For instance, chains bib.book.author and store.component are represented as [bib; book; author] and [store; component] respectively.

Prefix-Trees Prefix-trees are trees where all chain prefixes are stored only once. Each path in the tree is a chain. This optimizes chain inference wrt using lists. For instance, given two chains bib.book.title and bib.book.author the prefix-tree allow to store only the prefix bib.book on a single edge, even if it is shared by both chains.

Proposition 5.1.2. *The set of* rooted *chains induced by a saturated non-recursive DTD* d *of* $n+1$ *types requires, to be stored on lists,* $\Theta(n2^n)$ *space.*

Proof. The proof follows from some combinatorics. As discussed in the proof of Theorem 5.1.1, d being saturated and non-recursive, the power set of types in d is the equivalent of $\mathsf{C_d}$. Since we count rooted chains, we consider all subsets of types containing the root, and hence we count n types instead of $n+1$. We model space occupancy by making a weighted sum of all subsets of types of size $1, 2, \ldots, n$. The number of sets of types of size i in d can be counted by using the number of combinations of i elements chosen from a set of n elements $\binom{n}{i}$. Then we consider that, by using lists, a subset of size i would need space proportional to $i+1$ (by re-considering also the root). Therefore, list space occupation is

$$\sum_{i=0}^{n}\binom{n}{i}(i+1) \;=\; \sum_{i=0}^{n}\binom{n}{i}i \;+\; \sum_{i=0}^{n}\binom{n}{i} \;=\; n2^{n-1} + 2^n \;=\; (n+2)2^{n-1}$$

and thus we obtain $\Theta(n2^n)$. □

Next statement shows that, even if we use an optimized data-structure such as prefix-trees for storing chains, we still need exponential space for storing all schema chains.

Proposition 5.1.3. *The set of* rooted *chains induced by a saturated non-recursive DTD* d *of* n *types, can be stored on a prefix tree in* $\Theta(2^n)$ *space.*

Proof. The result follows from defining the size of a tree as a recurrence relation. We consider the worst case where there is a maximum type dependency for the non-recursive DTD d. Let us denote by $Size(i)$ the size of the prefix tree containing all rooted chains of a DTD with i types. For simplicity, we define the space occupancy of a node and of an edge as 1. The growth of the size of the tree is depicted in Figure 5.1. In general, for $n > 0$, we have the recurrence relation

$$Size(n) \;=\; 2 \cdot Size(n-1) + 1$$

from which we conclude that the size of the tree is in $\Theta(2^n)$. □

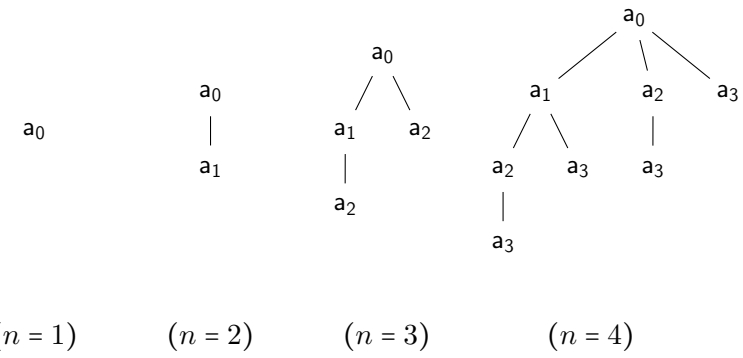


Figure 5.1: Prefix-tree for saturated non-recursive schemas with n types

From the previous results we have that trees are just linearly more succinct than lists for storing chains.

By considering recursive DTDs, the number of chains inferred for an expression augments substantially. We investigate the cardinality of sets of inferred chains for recursive schemas. To this end we adopt multisets, defined as follows.

Definition 5.1.4 (Multiset)**.** *A multiset* (M, μ) *is a set of (distinct) elements* M *together with a function* μ *that assigns to each element in* M *a non-negative integer value of multiplicity.*

For instance, $(\{a, b\}, \{(a, 1), (b, 3)\})$ denotes the multiset in which the element a has one occurrence and b three occurrences. We say that (M', μ') is a sub-multiset of (M, μ) if $M' \subseteq M$ and $\mu'(m) \leq \mu(m)$, for all $m \in M$. For instance, let $M = \{a, b\}$, $\mu = \{(a, 1), (b, 3)\}$ and $\mu' = \{(a, 0), (b, 1)\}$, the multiset (M, μ'), corresponding to the set $\{b\}$, is a sub-multiset of (M, μ). In short, we denote by M_k the multiset (M, μ) where each element of M has multiplicity k, so that $\mu(a) = k$ for all $a \in M$.

Proposition 5.1.5. *A saturated recursive DTD* d *with* $n+1$ *types induces* $\Omega(\frac{(nk)!}{(k!)^n})$ *and* $O(\frac{(nk+1)!}{(k!)^n})$ *rooted* k*-chains.*

Proof. Because d is saturated and recursive, a rooted k-chain is a sequence of types, whose length is upper bounded by $1+nk$. We model this by using multisets. Let M be the set of n mutual recursive types in d, and let M_k be the multiset of M elements with multiplicity k each. A permutation of a multiset M is an ordered arrangement of objects of M. The set of rooted k-chains of d can be seen as the *permutations with repetition* of all sub-multisets of M_k. For instance, let (M,μ') be the multiset with $M=\{m_1,\ldots,m_n\}$ and

$$\mu' = \{\,(m_1, k-1)\,,\ (m_2, k)\,,\ \ldots\,,\ (m_n, k)\,\}$$

We have that (M,μ') is a sub-multiset of M_k. The permutations associated with (M,μ') can be seen as the set of all chains of length nk, where all types repeat exactly k times, but type m_1 repeats exactly $k-1$ times. Of course, if b^1, b^2 are the first and second occurrence of b, respectively, then ab^1b^2 and ab^2b^1 denote the same permutation.

The number of k-chains for d can be computed exactly with the basic combinatorial formula for permutations with repetition, as the sum of all possible configurations of multiplicity values $k_1,\ldots,k_n$ (assigned to type-names $m_1,\ldots,m_n$) that are ranging from 0 to k. This is

$$\sum_{0\le k_1,\ldots,k_n\le k} \frac{(k_1+\cdots+k_n)!}{k_1!\cdot\cdots\cdot k_n!}$$

also counting the singleton root-type.

A lower bound for this quantity can be given by computing just part of the inferred chains, namely the permutations of M_k of maximum length nk (we ignore subsets for the moment). This can be done with the basic formula and results in

$$perm(M_k; nk) = \frac{(nk)!}{k_1!\cdot\cdots\cdot k_n!}$$

with $k_1=\cdots=k_n=k$. This quantity is exactly the number of rooted k-chains of length nk, and thus we have the lower bound $\Omega(\frac{(nk)!}{(k)!^n})$.

An upper bound for the set of k-chains induced by d is obtained in the following way. First, notice that the whole set of rooted k-chains of d is the set of prefixes of the chains of maximal length we computed above. Therefore, consider that each chain of length $1+nk$ has nk prefixes. So as to obtain

$$nk\frac{(nk)!}{(k!)^n}$$

which gives $O(\frac{(nk+1)!}{(k!)^n})$ as upper bound of inferred chains. This upper bound is not tight, because we count twice a prefix shared by two chains, but still helpful in order to understand the result size. It is worth observing that $perm(M_k; nk)$ is strictly smaller than the number of permutations with repetitions of length nk of n types, that is n^{nk}. □

These results show that chain inference is much harder in the presence of recursive schemas. Once again, it is possible to show that by using both lists and prefix-trees for storing chains, the space occupancy is exponential in the size of the DTD and the value k; prefix trees are still polynomially more succinct than lists. Because space complexity has a direct impact on time complexity, this suggests that implementing the chain inference defined in Chapter 4 with one of these two data-structures may compromise scalability, as shown by the following result.

Theorem 5.1.6. *Let* d *be a DTD with* n *types and* e *be a query or update expression of size* m*, exact* k*-chain inference for* e *is in EXPTIME.*

Proof. By Lemma 5.1.5 the maximal number of chains one can infer from DTD d is in $O(\frac{(nk+1)!}{(k!)^n})$. Consider now the case where *each* subexpression of e needs to process all of such chains. All inference rules perform a linear number of operations on a single chain, but rules for insert and replace updates perform a quadratic number of operations. Hence we say that each inference step does at most a polynomial number of operations on the set of input chains. In turn, each operation on a chain can be done in $O(nk)$, that corresponds to the traversal of the whole chain. We conclude that exact chain inference for the whole expression of size m can be done in $O(m \cdot nk \cdot \text{poly}(\frac{(nk+1)!}{(k!)^n}))$. □

This theorem gives an upper bound to the complexity of the chain inference problem and it shows that exact chain inference may have an enormous worst-case cost by using data-structures such as lists or trees. Of course this does not means that for any instance of the problem, chain inference is expensive. However, in practice it turns out that for schemas featuring cliques of mutual recursive types, the k analysis works efficiently only if k is small, and rapidly becomes intractable. From this we can deduce also that, once chains are inferred from queries and updates (in exponential time), checking for a conflict may require an exponential time, because of the number of chains one has to consider.

5.2 Storing Chains on DAGs

To overcome the limits outlined in the previous section, we present now a data-structure for storing chains based on directed acyclic graphs (DAGs) which is exponentially more succinct than trees, and allows to store chains in polynomial space and makes the analysis tractable in practice.

The intuition behind the use of DAGs instead of trees is that rather than storing only once common-prefixes of chains, in many cases we can also store only once common suffixes of chains. This can be done in a particularly efficient way for XML schemas that, being regular tree grammars and generate regular trees-of-chains featuring a number of repeating subtrees.

To illustrate, consider the chains induced by the non-recursive DTD d_8, which is defined by the following set of productions (below, we assume $1 \leq i \leq 3$).

$$\mathsf{a} \leftarrow \mathsf{b}_1, \mathsf{f}_1 \qquad \ldots \qquad \mathsf{b}_{(i-1)} \leftarrow (\mathsf{b}_i\,,\,\mathsf{f}_i)? \qquad \mathsf{f}_{(i-1)} \leftarrow (\mathsf{b}_i\,,\,\mathsf{f}_i)?$$

In Figure 5.2.a we show how the chains of d are stored as a prefix-tree. From now on, we call CTREE a prefix-tree storing chains. The CTREE shows clearly that by unfolding schema chains we are unfolding regular subtrees. We stress this fact by marking repeated subtrees with different colors.

In Figures 5.2.(b-d) we show that repeated subtrees can be recursively compressed until obtaining a DAG that is equivalent to the CTREE. From now on, we call CDAG a DAG storing chains. In fact, when there is a path (equivalently, a chain) in the CTREE, there is a path in the CDAG, and vice-versa.

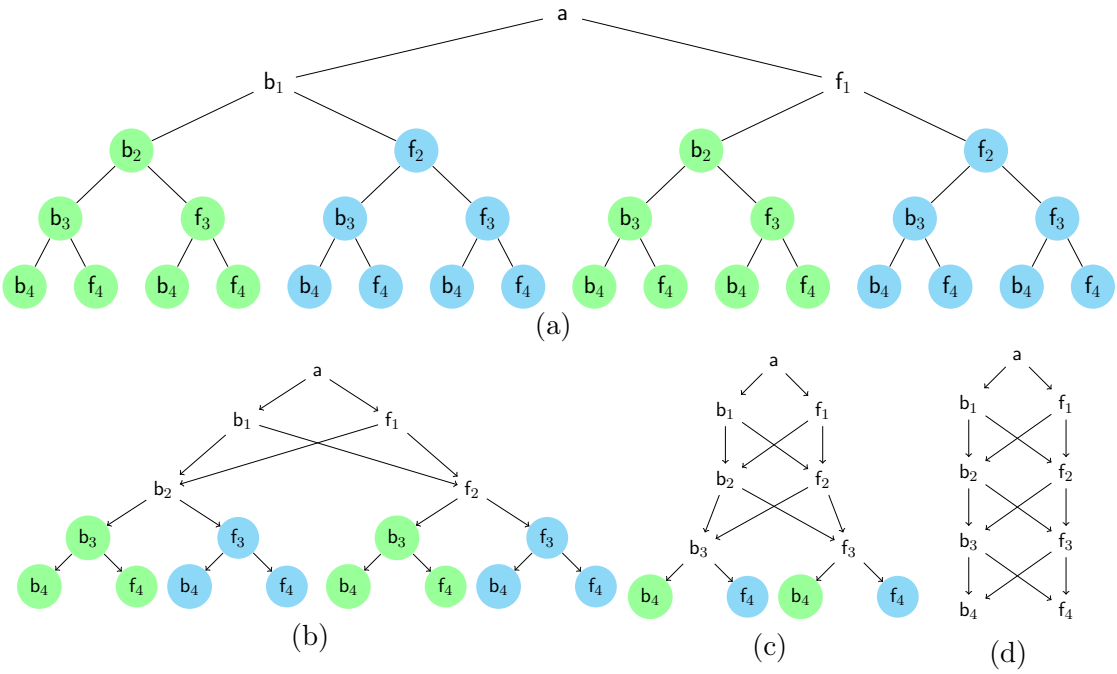


Figure 5.2: Compression of a CTREE to an equivalent CDAG

As suggested by the example, it is more convenient to perform the inference of the CDAG rather than the tree. Of course, we avoid to first infer prefix-trees and then compress them, because this again lead us to infer structures of exponential size. In the following, when speaking about CDAGs and CTREEs, we assume that they are inferred on a given DTD d.

A CDAG G is a directed acyclic graph rooted at the schema root-type, that contains no self-loop and meets the following property.

There is at most one node labeled a *at depth* h *in* G (5.1)

This means that, if two chains happen to have the same type in position h, they will share a common node in the CDAG at depth h. This property is for instance enjoyed by the tree in Figure 5.2.d. A CDAG is thus organized per levels.

Property (5.1) makes the *width* of the CDAG upper-bounded by the number of types in the schema size. Also, the k-analysis makes the *height* of a CDAG linear in the size of the schema. Therefore the number of vertex of a CDAG is always finite, and polynomial in d and q.

Let us illustrate property (5.1) with some examples over recursive and non-recursive schemas.

Consider the saturated non-recursive schema d_2 previously defined.

$$\begin{array}{lcl} \mathsf{a}_1 & \leftarrow & \mathsf{a}_2, \mathsf{a}_3, \mathsf{a}_4, \mathsf{a}_5 \\ \mathsf{a}_2 & \leftarrow & \mathsf{a}_3, \mathsf{a}_4, \mathsf{a}_5 \\ \mathsf{a}_3 & \leftarrow & \mathsf{a}_4, \mathsf{a}_5 \\ \mathsf{a}_4 & \leftarrow & \mathsf{a}_5 \end{array}$$

In Figure 5.3 we compare a CTREE and a CDAG storing all chains of d_2. Once again, we see the unfolding of the regular structure of the schema with the CTREE. By using DAGs instead of trees we can compress common chain-prefixes and store them only once. This is the case, for instance, for the subtrees rooted at a_4 and a_5 at level 2. In this case one could even compress more, by ignoring levels. However, this would substantially complicate not only the formal development, but also the implementation phase, thus resulting error prone for implementors.

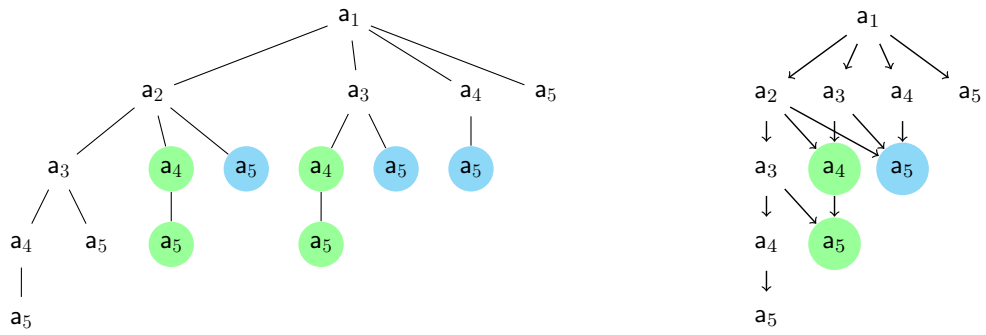
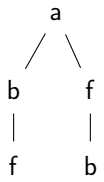


Figure 5.3: CTREE and CDAG storing chains of d_2

Consider now the saturated recursive schema d_9 defined at the beginning of the section.

$$\begin{array}{lcl} \mathsf{a} & \leftarrow & \mathsf{b}, \mathsf{f} \\ \mathsf{b} & \leftarrow & \mathsf{b}*, \mathsf{f}? \\ \mathsf{f} & \leftarrow & \mathsf{b}?, \mathsf{f}* \end{array}$$

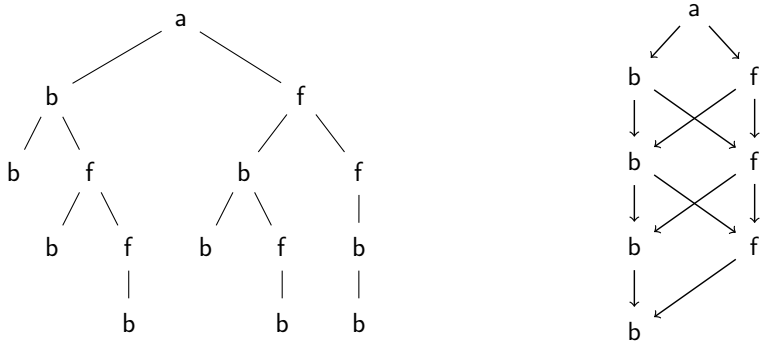
Let us consider all 1-chains induced by d_9. The CTREE and CDAG representing such chains are the same, as depicted in the following. Recall that each rooted path is a chain,



and 1-chains for d_9 are {a, a.b, a.f, a.b.f, a.f.b}.

Now, let us consider all 2-chains for d terminating with type b. This is, for instance, the specification of chains matching a recursive navigation made by the query $//b$. Below we compare a CTREE and a CDAG storing such chains.

A chain in the graph is a rooted path ending with type b. Notice that a chain belongs to the CDAG if it belongs to the CTREE but, in this case, there are also some chains belonging to the CDAG that do not belong to the CTREE. The following chains are obtained as artifacts from the compression.



a.b.b.b.b a.f.f.f.b

While this could suggest that the analysis imprecise (in fact, by inferring more chain than needed we may have false-negatives) it does not. The reason is that, because of the regularity of tree grammars, chains a.b.b.b.b and a.f.f.f.b, are indeed 4-chains matching $//b$ over d_9. Therefore in this case chain inference for $//b$ with CDAGs is both sound and complete, in the sense that it infers all 2-chains matching $//b$ (soundness) and only chains matching $//b$ (completeness).

While at first sight one could argue that, for the purpose of independence detection, a refined notion of completeness (wrt k-chains) may be needed, this is not the case. Indeed a k'-chain matching an expressions, with $k' > k$, cannot generate false negatives.

To conclude the discussion on the size of CDAGs, we prove that they are always polynomial in d.

Lemma 5.2.1. *Let* d *be a non-recursive DTD with* $n+1$ *types. The size of a* C*DAG inferred for all chains of* d *is in* $O(n^3)$.

Proof. Assume that d is a saturated non-recursive schema featuring $n+1$ types. The CDAG storing all d chains is organized in $n+1$ levels where a) the root-level has one node and b) for the remaining n levels, level i has i nodes ($1 \leq i \leq n$). Therefore we conclude that the CDAG has $1 + \frac{n(n+1)}{2}$ nodes. Concerning the number of edges, we notice first, that, at the root level, the root type is connected with all other n types. Then, as we said, level i has i types. Assume that $\mathsf{a}_2, \ldots, \mathsf{a}_i$ is the sequence of types at level i, ordered by the number of outgoing edges (ascending order). Because the schema is saturated, we have that type a_j at level i is connected with $j-1$ types at level $i+1$. We conclude that the size of the edge-set is $n + \sum_{i=1}^{n} \sum_{j=0}^{i-1} j$, which is in $O(n^3)$. □

Lemma 5.2.2. *Let* d *be a recursive DTD with* n *types and* k *a multiplicity value. The size of a* C*DAG inferred for* k*-chains of* d *is in* $O(kn^3)$.

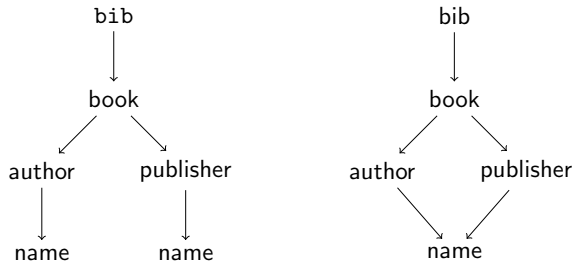
Proof. Assume that d is a saturated recursive schema featuring $n+1$ types. First we notice that the CDAG storing the k-chains of d has at most kn levels. Therefore, because each of the n types can repeat in all kn levels, the CDAG has kn^2 nodes. Because there exists at most n^2 edges between two consecutive graph levels, the CDAG has $O(kn^3)$ edges. We conclude since no edge crosses multiple levels. □

Chain Inference and Edge-labeling

CDAGs do not allow only to reduce space occupancy, but also to reduce the number of operations needed by the inference. To illustrate, consider the query $\mathsf{q} = //book/*/name$ and the DTD d_3 previously defined as

$$\begin{array}{rcl} \mathsf{bib} & \leftarrow & \mathsf{book}* \\ \mathsf{book} & \leftarrow & \mathsf{title}, \mathsf{author}+, \mathsf{publisher}+ \\ \mathsf{author}, \mathsf{publisher} & \leftarrow & \mathsf{name} \\ \mathsf{title}, \mathsf{name} & \leftarrow & \mathsf{String} \end{array}$$

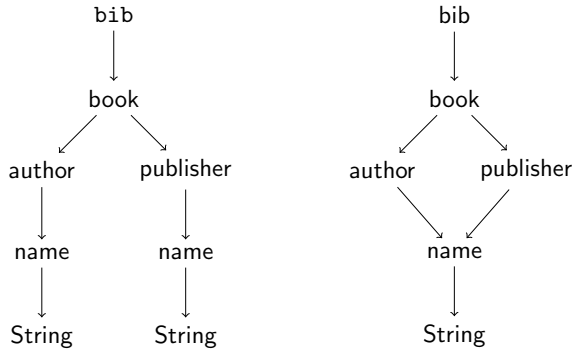
The CTREE and the CDAG storing chains of q over d_3 are reported below.



Now, assume a further inference step has to be done, such as for query

$$\mathsf{q}' = \mathsf{q}/\texttt{text}()$$

Now, in order to extend the CTREE, two operations are needed (one for each node name) while, in order to extend the CDAG, only one operation is needed.



This shows that, with CDAGs, we can also save computations by doing chain inference by groups of chains. The number of groups of chains is polynomial in the size of a schema and of a query (in fact, a group is represented by all chains terminating in one node). Thus this makes polynomial the number of input nodes for an intermediate inference operation.

Another ingredient needed in the CDAG representation is a means to represent chains inferred for unrelated sub-expressions. Intuitively two expressions are related if one draws data from the other (e.g. by means of variable binding). If this does not happen, the expressions are unrelated. To illustrate, consider the bibliographic DTD d_3 and the following sequence

$$\mathsf{q}_1\,,\,\mathsf{q}_2 = \mathit{//book//name/}\texttt{text()}\,,\ \mathit{/bib/book/publisher/name/foo}$$

Here the whole expression is constituted of two unrelated queries q_1 and q_2. It turns out that these may produce unneeded chains a consequence of overlapping.

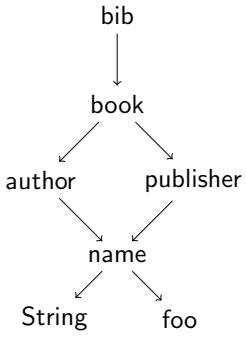
The chains inferred for q_1 over d_3 are

bib.book.author.name.String bib.book.publisher.name.String

The chain inferred for q_2 over d_3 is

bib.book.publisher.name.foo

The naive CDAG representation of the chains for the two subexpressions is

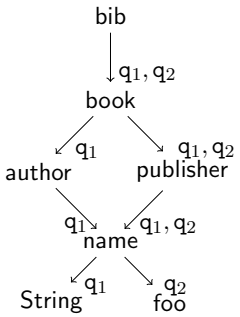


If the chains inferred for the two expressions are stored together, without any distinction, then the CDAG also encodes the chain

bib.book.author.name.foo

which is not a chain that should be inferred neither for q_1 nor q_2.

In order to overcome ambiguity in the CDAG, we introduce some marking for distinguishing among two chains inferred for distinct sub-expressions in the CDAG representation. We do this by putting on any edge connecting two CDAG-nodes a *label* identifying the query/update expressions that produced the chain during inference. These identifiers are necessary to precisely perform chain inference, and independence checking as well. The CDAG representation of chains for $\mathsf{q}_1, \mathsf{q}_2$ is the following.



We see that the labelled CDAG allows for a more precise analysis, in this case for instance the chain bib.book.author.name.foo cannot be traced anymore. (through q_1 or q_2 identifiers).

Labels help also in dealing with backward navigations. We can observe that if in the above query $\mathsf{q}_1, \mathsf{q}_2$ we replace q_2 by

$$\mathsf{q}_2 = /bib/book/publisher/foo/\texttt{ancestor::}*$$

when inferring chains for the last step of q_2, thanks to edge-labeling, avoids to navigate upward parts of the CDAG that have not been generated by q_2 (i.e., a author node), thus preserving the precision of the analysis in most of the cases.

Notice that backtracking on unvisited nodes does not affect the correctness of the analysis, but only its precision. The same observation holds for tracing unvisited chains. Removing labels, is a possibility if one wants to further lower the complexity of the analysis, that would result in dealing with a dimension less.

5.3 Chain Inference with CDAGs

In this section we present algorithms for implementing the chain inference with CDAGs. We first give some auxiliary definitions, and then we present chain inference. We begin with operators for XPath single steps, then treat navigational queries, element construction and, finally, we consider updates.

Preliminary definitions

Definition 5.3.1 (CDAGs)**.** *A CDAG is a directed acyclic rooted graph* $G = (V, E, \mathsf{s})$*, where* $V \subseteq \Sigma_{\mathsf{S}} \times \mathbb{N}$ *is a set of nodes that store a type in* Σ_{S} *and an integer value denoting the distance of the node from the root in* G*, where node* $(\mathsf{s}, 0) \in V$ *denotes the root type, and* $E \subseteq V^2 \times \mathbb{N}$ *is a set of edges among nodes recording the code of the query/update subexpression that produced that edge.*

In order to implement our analysis we need some auxiliary tools for managing chains inferred from queries and updates.

We first need an *encoding* of query and update expressions that is used for separating the chains inferred from unrelated subexpressions. The concept of subexpression is standard: $\mathtt{e}'$ is a subexpression of $\mathtt{e}$, denoted by $\mathtt{e}' \in \mathtt{e}$, if the parse tree of $\mathtt{e}'$ is a subtree of the parse tree of $\mathtt{e}$.

Definition 5.3.2 (Query encoding)**.** *The encoding of a query or update expression* $\mathtt{e}$ *is an injective function* $id()$ *mapping each subexpression of* $\mathtt{e}$ *to an integer in* $\mathbb{N}$*. The encoding of a set of expressions* $\{\mathtt{e}_1, \ldots, \mathtt{e}_n\}$ *is a function obtained by the union of functions encoding each* $\mathtt{e}_i$*, with* $1 \leq i \leq n$*, provided that this union is still injective.*

The definition says that basically any coding would work for the inference task, provided that we are able to uniquely identify each subexpression. This is why the function is required to be injective. As an example, consider the sequence

$$\mathtt{q}_0 = (\mathtt{q}_1, \mathtt{q}_2)$$

for which we can have the following encoding.

$$id(\mathtt{e}) = \begin{cases} 1 & if \quad \mathtt{e} = \mathtt{q}_1 \\ 2 & if \quad \mathtt{e} = \mathtt{q}_2 \\ 12 & if \quad \mathtt{e} = \mathtt{q}_0 \end{cases}$$

provided that there are no other subexpressions in $\mathtt{q}_1, \mathtt{q}_2$.

The second tool we need is a relation for capturing related expressions. As already said, two expressions are related if one draws data from the other. As an example, for the query

$$\texttt{for x in } //book \texttt{ return x}/author$$

expressions $//book$ and $\mathtt{x}/author$ are related because the first provides nodes that are bound to $\mathtt{x}$, and the second accesses the content of $\mathtt{x}$. As the example suggests, in the language we considered this relationship is established by the use of variables, and allows to reconstruct all navigational-paths embodied by a query or an update. We assume wlog that queries are written by using distinct variable-names for distinct variable-definitions.

To formalize this, we need a way to retrieve, given a variable $\mathtt{x}$, the set of navigational steps that provide data to $\mathtt{x}$. Assume that

$$\mathtt{e} = \texttt{for/let x in/} := \ \mathtt{e}_1 \ \texttt{return } \mathtt{e}_2$$

then the set of navigational expressions providing data to $\mathtt{x}$ is defined as $ret(\mathtt{e}_1)$, where

$$ret(\mathtt{e}) \stackrel{def}{=} \begin{cases} \{\mathtt{e}\} & \text{if } \ \mathtt{e} = \texttt{y/step} \\ ret(\mathtt{e}_2) & \text{if } \ \mathtt{e} = \texttt{for/let y in/} := \ \mathtt{e}_1 \ \texttt{return } \mathtt{e}_2 \\ ret(\mathtt{e}_1) \cup ret(\mathtt{e}_2) & \text{if } \ \mathtt{e} = \texttt{if } (\mathtt{e}_0) \texttt{ then } \mathtt{e}_1 \texttt{ else } \mathtt{e}_2 \text{ or } \mathtt{e} = \mathtt{e}_1, \mathtt{e}_2 \\ \varnothing & \text{otherwise} \end{cases}$$

To illustrate function $ret()$, consider the following query

$$\mathtt{q} = \mathtt{for\ x\ in\ q_1\ return\ q_2}$$

with $\mathtt{q_1} = (//title, //author)$ and $\mathtt{q_2} = \mathtt{x}//book$. In this case we have that

$$ret(\mathtt{q}) = \{\ \mathtt{x}//book\ \} = ret(\mathtt{q_2}) \qquad ret(\mathtt{q_1}) = \{\ //title, //author\ \}$$

We now define expressions related by the use of variables.

Definition 5.3.3 (Variable-Reference)**.** *Let* e *be a query or update expression, and* x *a variable defined in* e*. The set of navigational steps referenced by* x*, denoted by* $vr(\mathtt{x})$*, is defined as* $vr(\mathtt{x}) = ret(\mathtt{e_x})$*, provided that* $\mathtt{e_x}$ *is a for/let expression introducing* x*.*

To illustrate this definition, consider the query

```
q =  for x in
      (for y in //a,//d return y//b)
     return
      let z :=
       if (//b) then x else x/parent::*
       return x
```

The variable-reference relation, and its transitive closure, are graphed below.

$$\mathtt{y} \xrightarrow{vr(\mathtt{y})} //a, //d \qquad\qquad \mathtt{x} \xrightarrow{vr(\mathtt{x})} \mathtt{y}//b \xrightarrow{vr(\mathtt{y})} //d$$
$$\mathtt{x} \xrightarrow{vr(\mathtt{x})} \mathtt{y}//b \xrightarrow{vr(\mathtt{y})} //a \qquad\qquad \mathtt{z} \xrightarrow{vr(\mathtt{z})} \mathtt{x}, \mathtt{x/parent::*}$$

$$\mathtt{z} \xrightarrow{vr(\mathtt{z})} \mathtt{x} \xrightarrow{vr(\mathtt{x})} \mathtt{y}//b \xrightarrow{vr(\mathtt{y})} //a \qquad\qquad \mathtt{z} \xrightarrow{vr(\mathtt{z})} \mathtt{x/parent::*} \xrightarrow{vr(\mathtt{x})} \mathtt{y}//b \xrightarrow{vr(\mathtt{y})} //a$$
$$\mathtt{z} \xrightarrow{vr(\mathtt{z})} \mathtt{x} \xrightarrow{vr(\mathtt{x})} \mathtt{y}//b \xrightarrow{vr(\mathtt{y})} //d \qquad\qquad \mathtt{z} \xrightarrow{vr(\mathtt{z})} \mathtt{x/parent::*} \xrightarrow{vr(\mathtt{x})} \mathtt{y}//b \xrightarrow{vr(\mathtt{y})} //d$$

With a little abuse of notation, we denote by $vr(\mathtt{e})$, $vr(i)$ and $vr(\mathtt{x})$ the same set, provided that $\mathtt{e} = \mathtt{x/step}$ and $id(\mathtt{e}) = i$. The transitive closure of $vr(\mathtt{x}), vr(\mathtt{e})$ and $vr(i)$ are denoted by $vr^+(\mathtt{x}), vr^+(\mathtt{e})$ and $vr^+(i)$, respectively.

The following property shows that an exponential number of sequences induced by the reference variable relation may arise.

Proposition 5.3.4. *Let* e *be a query or update expression and* $\mathsf{e}_1, \ldots, \mathsf{e}_n$ *be a sequence of subexpressions of* e *such that* $\mathsf{e}_{(i+1)} \in vr(\mathsf{e}_i)$*. The following holds.*

1. *the sequence size is bounded by the query size,* $n \sim O(|\mathsf{e}|)$
2. *the number of such sequences is in the worst case exponential* $O(2^n)$
3. *if no concatenation or conditional expression is nested in the left-branch of a for or let expression, then the number of such sequences is linear* $O(n)$*.*

The first item states that the *length* of a navigational path embodied in a query is upper bounded by the query size. The second item shows the succinctness of the query and update languages. It states that a query of size n can embody 2^n navigational paths. The third item states that this succinctness is given by two operators: query concatenation and conditional expressions.

Proposition 5.3.4 suggests also that chain inference may lead to perform an exponential number of operations if the whole relation $vr^+()$ has to be visited. For instance, this may happen in the presence of backward navigations. Nevertheless, differently from trees and lists, CDAGs allow to design algorithms that are sound, that run in polynomial time and that are still precise, as we illustrate next.

Auxiliary Data-Structures The CDAG inference algorithms make use of some global data structures that we define below. The role of such structures will be clarified during the presentation of inference algorithms.

- RET() is an *inverted index* mapping an expression id i to a set of nodes M in the CDAG. The set of nodes N corresponds to terminal of *return* chains inferred for e.
- UPD() is an *inverted index* mapping an expression e to a set of nodes M in the CDAG. The set of nodes N corresponds to nodes in the suffix of an update chain inferred for e.
- CLEAN() is an *inverted index* mapping a step e to a set nodes representing dangling chains for which inference for e failed.

The input of the chain inference algorithm is a query or update expression e, a schema d and a multiplicity value k. The system outputs a CDAG G_{rv} representing used and return chains, a CDAG G_{e} representing element chains and a G_{U} representing update chains inferred form e, respectively. Note that used and returned chains are put together in the same CDAG.

Given a CDAG G, and node v belonging to G is a pair $v = (\mathsf{a}, h)$, where a is the type of the node and h is the level of the node in the G. The type of a node v is denoted by $v.type$ while its height is denoted by $v.level$. In the examples, a node is also denoted by $v_{[\mathsf{type},height]}$.

Chain Inference Algorithms

Algorithm 1: Procedure CHILD

Input : Node-set $M \subseteq V_{\mathsf{rv}}$, ϕ, id i
Output: Infer chains for $\texttt{child} :: \phi$, side-effect on RET(), CLEAN(), V_{rv} and E_{rv}

```
1  foreach v ∈ M do
2      new_edge ← false
3      if v.level + 1 does not exceeds the chain space then
4          foreach a child-type of v.type do
5              if a satisfies φ then
6                  if (a, v.level + 1) ∉ V_rv then
7                      V_rv ← V_rv ∪ {(a, v.level + 1)}
8                  v' ← (a, v.level + 1)
9                  E_rv ← E_rv ∪ (v, v', i)
10                 new_edge ← true
11                 RET(i) ← RET(i) ∪ {v'}
12     if !new_edge then
13         CLEAN(i) ← CLEAN(i) ∪ {v}
```

The first procedure we present is the one to infer chains for a navigational expression of the form $\texttt{x/child} :: \phi$. Procedure CHILD is the most important procedure of the system, since it is the one responsible of extending the CDAGs. Other operators presented later on are defined in terms of the procedure CHILD.

The procedure takes as input a set of nodes $M \subseteq V_{\mathsf{rv}}$, and tries to connect each $v \in M$ with a child node v', provided that v' satisfies the filtering condition ϕ. If such v' is not a node of the CDAG, then v' is built and added to the vertex-set of the CDAG (line 7). Then v' becomes a terminal node of the expression invoking the `child` axis. The step expression $\texttt{x/child} :: \phi$, for which the procedure infers chains, is referred by identifier i, which is also part of the input.

In line 3, we have a condition that ensures the termination of the inference. It says that a child navigation could be performed, provided that chains do not exceed a certain length. The maximal length of a chain we infer is nk, provided that that n is the number of types in d, and k is the multiplicity value in input. This is a sufficient condition to ensure that all k chains are correctly inferred. Note also that this avoids to keep an hash table recording the frequency of each tag while doing the chain inference, thus ensuring better performances.

Finally, in the cases that the child extension for the input node v produces no new edges, then v is added to the list of possibly dangling nodes to backtrack (Lines 12-13).

This happens, for instance, if v exceeds the chain space, or none of the children on v satisfy ϕ. A dangling node is later removed, provided that itself or one of its descendants are terminals of used or return chains in the CDAG. This essentially implements the chain filtering defined in chain inference rules of Chapter 4. For the sake of conciseness, we do not model this operation here.

Let us illustrate the workflow of the procedure with an example. Consider chain inference for the query $\mathtt{x}/book$ over DTD d_3, with $\mathtt{x} \mapsto \mathsf{bib}$. Inference for the step takes in input the following CDAG

$$G_{\mathsf{rv}} = (\{\ v_{[\mathsf{bib},0]}\ \}, \varnothing)$$

and a set $M = \{\ v_{[\mathsf{bib},0]}\ \}$. The procedure first checks that the chain space would not be exceeded by attaching a child to $v_{[\mathsf{bib},0]}$, which is trivially true since

$$v_{[\mathsf{bib},0]}.level \ < \ nk$$

where we recall that n is the number of types in d_3 (in this case $n = 8$) and k is the multiplicity value (in this case $k = 1$).

At this point, because book is the only child-type of bib, and book satisfies ϕ=*book*, the node $v_{[\mathsf{book},1]}$ is added to V and connected to $v_{[\mathsf{bib},0]}$. Below, we assume $id(\mathtt{x/child::}\phi) = i$. Node $v_{[\mathsf{book},1]}$ is now the terminal of chains inferred for the query. The resulting G_{rv} and RET() are the following.

bib
$\downarrow\ i$
book

$$G_{\mathsf{rv}} = (V_{\mathsf{rv}}, E_{\mathsf{rv}})$$
$$V_{\mathsf{rv}} = \{\ v_{[\mathsf{bib},0]}\ ,\ v_{[\mathsf{book},1]}\ \}$$
$$E_{\mathsf{rv}} = \{\ (v_{[\mathsf{bib},0]}\ ,\ v_{[\mathsf{book},1]}\ , i)\ \}$$

$$\textsc{Ret}(i) = \{\ v_{[\mathsf{book},1]}\ \}$$

The second procedure defined is the one implementing the `self` axis. Procedure SELF receives as input a set of nodes $M \subseteq V_{\mathsf{rv}}$ and only checks if the type-name of each input node satisfies condition ϕ. If this is true, the input node becomes a terminal for the expression. This is captured by line 3.

Algorithm 2: Procedure SELF

Input : Node-set $M \subseteq V_{\mathsf{rv}}$, ϕ, id i
Output: Infer chains for `self` axis, side-effect on RET()

```
1 foreach v ∈ M do
2     if v.type satisfies φ then
3         RET(i) ← RET(i) ∪ {v}
```

Notice that, if a node does not satisfy ϕ, then no operation is performed here. This is made on purpose, nodes that do not satisfy the axis test should be dealt with by the external procedures calling SELF.

Consider chain inference for query $\mathtt{y/self} :: *$ over DTD $\mathtt{d}_3$, provided that $\mathtt{y}$ is bound to the result of $\mathtt{x}/book$ and $\mathtt{x} \mapsto \mathsf{bib}$, computed as in the previous example. Chain inference for $\mathtt{y/self} :: *$ results in the following CDAG. As before, we assume that $id(\mathtt{x}/book) = i$ and $id(\mathtt{y/self} :: *) = i'$.

$$G_{\mathsf{rv}} = (V, E) \qquad \text{RET}(i) = \{ v_{[\mathsf{book},1]} \}$$
$$V = \{ v_{[\mathsf{bib},0]} , v_{[\mathsf{book},1]} \} \qquad \text{RET}(i') = \{ v_{[\mathsf{book},1]} \}$$
$$E = \{ (v_{[\mathsf{bib},0]} , v_{[\mathsf{book},1]} , i) \}$$

Note that procedure SELF adds no edge in the CDAG, but it just affects RET(). In this case because node $v_{[\mathsf{book},1]}$ trivially satisfies $\phi = *$ we have $\text{RET}(i') = \{ v_{[\mathsf{book},1]} \}$.

The algorithm implementing the descendant axis is described in the following. We just describe the algorithm implementing axis `descendant` since `descendant-or-self` is similar.

DESCENDANT-REC is a recursive procedure that computes chains for navigational steps of the form $\mathtt{x/descendant} :: \phi$. The descendant axis is computed by invoking the CHILD procedure. Because of this, in the pseudo-code, we use a fresh query-id i_v for labeling edges obtained from the iteration of the procedure, that started from node v. This is simply needed to correctly perform set operations and clean the graph form artificial edges.

At each iteration step, the procedure tries to extend of one level each input node v, with all of its children-types (line 5), by calling procedure CHILD. Before calling procedure CHILD, the abstract predicate satisfies uses schema information to check if some descendant-type of $v.type$ do satisfy the filter ϕ (line 4). When this is not true, any operation can be skipped and node v can be added to the backtrack list for the expression.

If some descendant-type of $v.type$ do satisfy the filter ϕ but the result of the child-navigation with $*$-filter is empty, it simply means that the node is at the frontier of the chain space. In this case the node is added to the list of possibly dangling nodes (line 14). If the result of the child navigation is non-empty (line 6), then we record that some new labeled-edges has been created (line 7). All nodes in $\text{RET}(i_v)$ are saved in the set M' (line 8), and the descendant procedure is iterated on M' (line 15). Also, all nodes in $\text{RET}(i_v)$ that satisfy ϕ, are set as terminal nodes for the expression, by means of the

Algorithm 3: Procedure DESCENDANT-REC

Input : Node-set $M \subseteq V_{\mathsf{rv}}$, ϕ, id i
Output: Infer chains for `descendant` axis, side-effect on RET(), CLEAN(), V_{rv} and E_{rv}

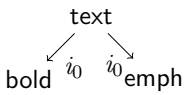
```
1  M' ← ∅
2  foreach v ∈ M do
3      new_edge ← false
4      if some descendant-type of v.type satisfies φ then
5          CHILD({v}, *, i_v)
6          if RET(i_v) ≠ ∅ then
7              new_edge ← true
8              M' ← M' ∪ RET(i_v)
9              SELF(RET(i_v), φ, i)
10             foreach (v, v', i_v) ∈ E do
11                 E ← E ∖ (v, v', i_v)
12                 E ← E ∪ (v, v', i)
13     if !new_edge then
14         CLEAN(i) ← CLEAN(i) ∪ {v}
15 DESCENDANT-REC(M', φ, i)
```

SELF procedure (line 9). At this point, all edges produced by the CHILD procedure with the input code i_v are re-labeled as i edges. Notice that i_v is needed to perform this operation.

We illustrate chain inference for step `x/descendant` :: *bold* with `x` bound to text over the recursive DTD d_{12} defined as

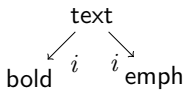
$$\begin{array}{rcl} \mathsf{text} & \leftarrow & (\,\mathsf{bold} \mid \mathsf{emph}\,)* \\ \mathsf{bold} & \leftarrow & \mathsf{emph}?, \mathsf{String} \\ \mathsf{emph} & \leftarrow & \mathsf{bold}?, \mathsf{String} \end{array}$$

In this case $k_{\mathsf{q}} = 1$ and thus we want to infer 1-chains for the query, namely text.bold and text.emph.bold. To this aim, procedure DESCENDANT-REC is iterated three times. In the first iteration, the procedure takes as input the triplet $(\{v_{[\mathsf{text},0]}\}, \phi, i)$. A child navigation from the root is then performed. This gives the following intermediate result. Below, i_0 is used to denote $i_{v_{[\mathsf{text},0]}}$.



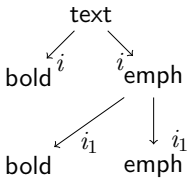
$$\text{RET}(i_0) \;=\; \{\, v_{[\mathsf{bold},1]}, v_{[\mathsf{emph},1]} \,\}$$

Now, because node $v_{[\mathsf{bold},1]}$ satisfies ϕ, it set as terminal node for the query, and we have $\textsc{Ret}(i) = \{\, v_{[\mathsf{bold},1]} \,\}$. Then edges labeled as i_0 are relabeled with i. We assume also that the terminal-index $\textsc{Ret}(i_0)$ is set to empty. However, since this is not mandatory for the correctness of the analysis we do not add it in the pseudocode. The result of the first iteration is the following.



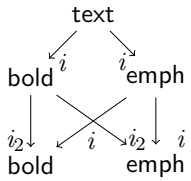
$$\textsc{Ret}(i) \;=\; \{\, v_{[\mathsf{bold},1]} \,\}$$

In the second iteration, the procedure takes as input the triplet $(\{v_{[\mathsf{bold},1]}, v_{[\mathsf{emph},1]}\}, \phi, i)$. The procedure iterates over the two input nodes. Let us consider first node $v_{[\mathsf{emph},1]}$. Below, i_1 is used as a shorthand for $i_{v_{[\mathsf{emph},1]}}$. Then, node $v_{[\mathsf{emph},1]}$ is extended with its children types, in the following way.



$$\begin{aligned}\textsc{Ret}(i) &= \{\, v_{[\mathsf{bold},1]} \,\}\\ \textsc{Ret}(i_1) &= \{\, v_{[\mathsf{bold},2]}, v_{[\mathsf{emph},2]} \,\}\end{aligned}$$

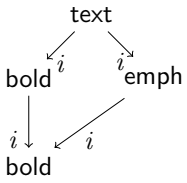
By following line 9, node $v_{[\mathsf{bold},2]}$ is set as terminal node since it satisfies test ϕ. Let us consider now node $v_{[\mathsf{bold},1]}$. Below we assume $i_2 = i_{v_{[\mathsf{bold},1]}}$. The child navigation creates an edge between $v_{[\mathsf{bold},1]}$ and $v_{[\mathsf{bold},2]}$. As we already discussed this is not problematic because the chain text.bold.bold indeed matches q. Notice also that there is a group of chains ending at $v_{[\mathsf{emph},2]}$ which does not match the query. They will be removed in the next iteration.



$$\begin{aligned}\textsc{Ret}(i) &= \{\, v_{[\mathsf{bold},1]}, v_{[\mathsf{bold},2]} \,\}\\ \textsc{Ret}(i_2) &= \{\, v_{[\mathsf{bold},2]}, v_{[\mathsf{emph},2]} \,\}\end{aligned}$$

The third recursive call of the procedure is the one that terminates the inference. In this iteration step, the input of the procedure is the triplet $(\{v_{[\mathsf{bold},2]}, v_{[\mathsf{emph},2]}\}, \phi, i)$. It turns out that both input nodes have some descendant types that satisfy ϕ, but they cannot be further extended, without exceeding the chain space. The maximal length of chains to consider is $1 + nk = 3$, because the number of types in the schema is 2 (plus the root) and $k = 1$.

Both nodes $v_{[\mathsf{bold},2]}$ and $v_{[\mathsf{emph},2]}$ are thus added to the backtracking list. Because node $v_{[\mathsf{bold},2]}$ is a terminal node for a query return chain it will not be removed. Because



$\textsc{Ret}(i) \;=\; \{\, v_{[\mathsf{bold},1]}, v_{[\mathsf{bold},2]} \,\}$

node $v_{[\mathsf{emph},2]}$ is not a terminal node, and it is not shared by any other expression, it is removed, and also all of its incoming edges. The resulting CDAG is the following.

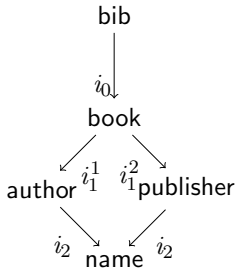
Procedure PARENT computes chains for backward steps of the form $\mathtt{x/parent} :: \phi$. In analogy with the child and descendant axes, this procedure is also used to compute the ancestor axis. The algorithm backtracks on each parent v' of an input node v, when the two nodes are connected by an edge i' belonging to the set I of ids that can be backtracked (which is part of the input). If the procedure is called to infer chains for a step $\mathtt{e} = \mathtt{x/parent} :: \phi$ then I is $I = vr^+(\mathtt{e})$. If the procedure is iterated to infer chains for an ancestor then I is defined wrt the ancestor step, as discussed next.

To illustrate, consider chain inference over DTD $\mathtt{d}_3$ and the following query

$$
\begin{array}{l}
\mathtt{for}\ \mathtt{x}_0\ \mathtt{in}\ \mathtt{x}//\mathit{book} \\
\mathtt{for}\ \mathtt{x}_1\ \mathtt{in}\ \mathtt{x}_0/\mathit{author},\ \mathtt{x}_0/\mathit{publisher} \\
\mathtt{for}\ \mathtt{x}_2\ \mathtt{in}\ \mathtt{x}_1/\mathit{name} \\
\mathtt{for}\ \mathtt{x}_3\ \mathtt{in}\ \mathtt{x}_2/\mathit{parent} :: * \\
\mathtt{for}\ \mathtt{x}_4\ \mathtt{in}\ \mathtt{x}_3/\mathit{parent} :: \mathit{book} \\
\mathtt{return}\ \mathtt{x}_4
\end{array}
$$

where $\mathtt{x}$ is bound to bib.

Below we assume the following coding $id(//\mathit{book}) = i_0$ and $id(\mathtt{x}_0/\mathit{author}) = i_1^1$ and $id(\mathtt{x}_0/\mathit{publisher}) = i_1^2$ and for the remaining steps $id(\mathtt{x}_i/\mathtt{step}_i) = i_{(i+1)}$. Chain inference for the first three lines of the query gives the following CDAG.



$$
\begin{aligned}
\textsc{Ret}(i_0) &= \{\, v_{[\mathsf{book},1]} \,\} \\
\textsc{Ret}(i_1^1) &= \{\, v_{[\mathsf{author},2]} \,\} \\
\textsc{Ret}(i_1^2) &= \{\, v_{[\mathsf{publisher},2]} \,\} \\
\textsc{Ret}(i_2) &= \{\, v_{[\mathsf{name},3]} \,\}
\end{aligned}
$$

Processing of step $\mathtt{x}_2/\mathtt{parent} :: *$ is now illustrated. Recall that $id(\mathtt{x}_2/\mathtt{parent} :: *) = i_3$.

Algorithm 4: Procedure PARENT

Input : Node-set $M \subseteq V_{\mathsf{rv}}$, ϕ, id i, id set I
Output: Infer chains for `parent` axis, side-effect on RET(), CLEAN()

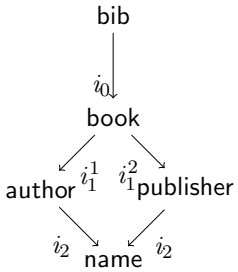
```
1 foreach v ∈ M do
2     some_parent ← false
3     foreach (v', v, i') ∈ E do
4         if v'.type satisfies φ then
5             if i' ∈ I then
6                 RET(i) ← RET(i) ∪ {v'}
7                 some_parent ← true
8     if !some_parent then
9         CLEAN(i) ← CLEAN(i) ∪ {v}
```

The procedure iterates over chains bound to $\mathtt{x}_2$, thus over node $v_{[\mathsf{name},3]}$. Because $i_2 \in vr^+(i_3)$, the procedure backtracks to the edges labeled as i_2, namely $(v_{[\mathsf{author},2]}, v_{[\mathsf{name},3]}, i_2)$ and $(v_{[\mathsf{publisher},2]}, v_{[\mathsf{name},3]}, i_2)$.

Let us consider first the edge $(v_{[\mathsf{author},2]}, v_{[\mathsf{name},3]}, i_2)$. Because $v_{[\mathsf{author},2]}$ satisfies ϕ, it is set as terminal node for the parent navigation, so $\text{RET}(i_3) = \{\ v_{[\mathsf{author},2]}\ \}$. The second iteration step on edge $(v_{[\mathsf{publisher},2]}, v_{[\mathsf{name},3]}, i_2)$ is similar.

Processing of step $\mathtt{x}_3$/`parent`::*book* is now illustrated. Recall that $id(\mathtt{x}_3/\mathtt{parent}::book)=i_4$.

The procedures iterates over chains bound to $\mathtt{x}_2$, thus $v_{[\mathsf{author},2]}$ and $v_{[\mathsf{publisher},2]}$. It backtracks on both edges labeled as i_1^1 and i_1^2, because $i_1^1, i_1^2 \in vr^+(i_3)$. Let us consider first edge $(v_{[\mathsf{book},1]}, v_{[\mathsf{author},2]}, i_1^1)$. Because $v_{[\mathsf{book},1]}$ satisfies $\phi = book$, it is set as terminal node for the step, so $\text{RET}(i_4) = \{\ v_{[\mathsf{book},1]}\ \}$. The second iteration step on edge $(v_{[\mathsf{book},1]}, v_{[\mathsf{publisher},2]}, i_1^2)$ is similar, and the resulting CDAG is the following.



$$
\begin{aligned}
\text{RET}(i_0) &= \{\ v_{[\mathsf{book},1]}\ \} \\
\text{RET}(i_1^1) &= \{\ v_{[\mathsf{author},2]}\ \} \\
\text{RET}(i_1^2) &= \{\ v_{[\mathsf{publisher},2]}\ \} \\
\text{RET}(i_2) &= \{\ v_{[\mathsf{name},3]}\ \} \\
\text{RET}(i_3) &= \{\ v_{[\mathsf{author},2]}, v_{[\mathsf{publisher},2]}\ \} \\
\text{RET}(i_4) &= \{\ v_{[\mathsf{book},1]}\ \}
\end{aligned}
$$

Note that the procedure does not add new edges to the CDAG but it just affects RET().

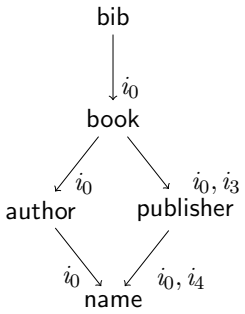
Algorithm 4 is simple, sound and ensure a quite precise analysis at once. It is worth outlining that, in some rare cases, the algorithms can infer unneeded chains, that is a

chain that would not be inferred by the type system presented in Chapter 4. This may happen for expressions that navigate twice the same part of the schema, as illustrated by the following example over the DTD d_3.

$$\begin{array}{l}\texttt{let } \texttt{x}_0 \texttt{ := x//}\mathit{name}\\ \texttt{let } \texttt{x}_1 \texttt{ := x}_0\texttt{/parent::*}\\ \texttt{let } \texttt{x}_2 \texttt{ := x}_1\texttt{/parent::*}\\ \texttt{let } \texttt{x}_3 \texttt{ := x}_2\texttt{/}\mathit{publisher}\\ \texttt{let } \texttt{x}_4 \texttt{ := x}_3\texttt{/}\mathit{name}\\ \texttt{return } \texttt{x}_4\texttt{/parent::*}\end{array}$$

where we assume x is bound to bib.

Below we assume $id(\texttt{x//}\mathit{name}) = i_0$ and $id(\texttt{x}_i\texttt{/step}) = i_{i+1}$. In this case, the last backward step of the query $\texttt{x}_4\texttt{/parent::*}$ produces an unneeded chain. To see this, consider the CDAG obtained by inference over all precedent steps.



$$\begin{array}{lcl}\textsc{Ret}(i_0) & = & \{\, v_{[\mathsf{name},3]} \,\}\\ \textsc{Ret}(i_1) & = & \{\, v_{[\mathsf{author},2]}, v_{[\mathsf{publisher},2]} \,\}\\ \textsc{Ret}(i_2) & = & \{\, v_{[\mathsf{book},1]} \,\}\\ \textsc{Ret}(i_3) & = & \{\, v_{[\mathsf{publisher},2]} \,\}\\ \textsc{Ret}(i_4) & = & \{\, v_{[\mathsf{name},3]} \,\}\end{array}$$

When processing step $\texttt{x}_4\texttt{/parent::*}$, the procedures iterates over chains bound to $\texttt{x}_4$, thus $v_{[\mathsf{name},3]}$. It backtracks on both edges labeled as i_0 and i_4, because $i_0, i_4 \in vr^+(i_5)$ and i_5 is the id of $\texttt{x}_4\texttt{/parent::*}$. At this point the procedures set as terminal nodes both $v_{[\mathsf{author},2]}$ and $v_{[\mathsf{publisher},2]}$, while it should backtrack toly on $v_{[\mathsf{publisher},2]}$.

In general, for a pair of codes i_1, i_2 such that $i_1 \in vr^+(i_2)$, by running Procedure 4 it is possible to visit edges labeled as i_1 before those labeled as i_2, and this can lead to infer chains as false negatives. However, it is indeed by avoiding to follow the transitive closure of relation $vr()$ in the proper order, that we can compute backward navigations in polynomial time.

The way backward navigations are used in the above example resembles that of a conditional expression. They allow to verify that node elements labeled with *book* do have descendants labeled with *name* and, if this is the case, descendant nodes of *book* ones labeled with *publisher* are further navigated. Notice that this is not the usual way of expressing such a property in an XQuery program. The above expression could be for instance reformulated by using a conditional expression such as

$$\texttt{if (x//}\mathit{name}\texttt{) then x/}\mathit{publisher}\texttt{/}\mathit{name}\texttt{/}\mathit{parent}\texttt{::* else ()}$$

Algorithm 5: Procedure ANCESTOR-REC

Input : Node-set $M \subseteq V_{\mathsf{rv}}$, node-set $S \subseteq V_{\mathsf{rv}}$, ϕ, id i
Output: Infer chains for `ancestor` axis, side-effect on RET(), CLEAN()

```
1  M' ← ∅
2  foreach v ∈ M do
3      if some ancestor-type of v.type satisfies φ then
4          PARENT({ v }, ⋆, i_v, vr⁺(i))
5          S ← S ∪ RET(i_v)
6          SELF(RET(i_v), φ, i)
7          M' ← M' ∪ RET(i_v)
8      else
9          CLEAN(i) ← CLEAN(i) ∪ {v}
10 ANCESTOR-REC((M' ∖ S), S, φ, i)
```

For this expression, chain inference over the DTD $\mathtt{d}_3$ is sound and complete wrt the inference system presented in Chapter 4, because edges inferred for the conditional query `x`//*name* would not be mixed with edges inferred for the rest of the expression.

As illustrated, cases of incompleteness exist but are quite rare in practice and may have an equivalent formulation for which the analysis is complete. The precision of our algorithm have been tested over the wide XMark/XPathMark benchmarks as shown in Chapter 6, without incurring in a situation analogous to the one presented here, and by showing high improvements wrt the state-of-the-art method.

Algorithms implementing ancestor axes are presented in the following. We just describe the algorithm implementing axis `ancestor` since the case for `ancestor-or-self` is similar. Procedure ANCESTOR-REC computes chains for steps of the form `x/ancestor` :: ϕ.

The procedure performs a bottom-up visit of the CDAG, by simply iterating the procedure for the parent axis (Line 4). The set of codes that can be visited by procedure PARENT is set as $vr^+(i)$, as discussed before. The parents of a node $v \in M$, that are recorded in RET(i_v), are then filtered by means of procedure SELF to check if the satisfy the ancestor navigation.

Furthermore, the procedure records a set of visited nodes S, so as to perform only once the parent navigation to each node of the G. This optimization makes the procedure running in linear time in the size of the CDAG. A node that has no parent satisfying ϕ represent a dangling chain that may be removed.

We illustrate the procedure with an example. Consider the DTD d_{11} defined as follows

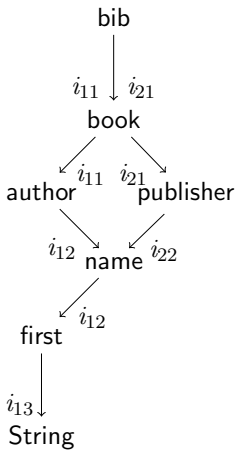
$$\begin{array}{rcl} \mathsf{bib} & \leftarrow & \mathsf{book}* \\ \mathsf{book} & \leftarrow & \mathsf{title}, \mathsf{author}, \mathsf{publisher} \\ \mathsf{author}, \mathsf{publisher} & \leftarrow & \mathsf{name} \\ \mathsf{name} & \leftarrow & \mathsf{first}, \mathsf{last}, \mathsf{String} \\ \mathsf{title}, \mathsf{first}, \mathsf{last} & \leftarrow & \mathsf{String} \end{array}$$

and the query sequence

$$\begin{array}{rll} \mathtt{q} = & \mathtt{for\ x_1\ in\ x}//\mathit{author} & \mathtt{for\ y_1\ in\ y}//\mathit{author} \\ & \mathtt{for\ x_2\ in\ x_1}//\mathit{first} \quad , & \mathtt{for\ y_2\ in\ y_1}/\mathit{name} \\ & \mathtt{return\ x_2/text()} & \mathtt{return\ y_2/ancestor::*} \end{array}$$

with both $\mathtt{x}$ and $\mathtt{y}$ bound to bib.

Below, we assume that $id(\mathtt{x}//\mathit{author}) = i_{11}$, $id(\mathtt{x_1}//\mathit{first}) = i_{12}$, $id(\mathtt{x_2/text()}) = i_{13}$, $id(\mathtt{y}//\mathit{author}) = i_{21}$, $id(\mathtt{y_1}/\mathit{name}) = i_{22}$ and $id(\mathtt{y_2/ancestor::}\mathit{book}) = i_{23}$. Provided that both $\mathtt{x}$ and $\mathtt{y}$ are bound to bib, chain inference for $\mathtt{q}$, yields the following CDAG.



$$\begin{array}{rcl} \textsc{Ret}(i_{11}) & = & \{\ v_{[\mathsf{author},2]}\ \} \\ \textsc{Ret}(i_{12}) & = & \{\ v_{[\mathsf{first},4]}\ \} \\ \textsc{Ret}(i_{13}) & = & \{\ v_{[\mathsf{String},4]}\ \} \\ \textsc{Ret}(i_{21}) & = & \{\ v_{[\mathsf{publisher},2]}\ \} \\ \textsc{Ret}(i_{22}) & = & \{\ v_{[\mathsf{name},3]}\ \} \end{array}$$

Now, inferring chains for $\mathtt{y_2/ancestor::*}$ consist of visiting all ancestors of nodes of $v_{[\mathsf{name},3]}$ on G_{rv} with a label in $vr^+(i_{22})$, that means i_{21} and i_{22}. The set of terminal nodes for $\mathtt{y_2/ancestor::*}$ is thus $\textsc{Ret}(i_{22}) = \{\ v_{[\mathsf{publisher},2]}, v_{[\mathsf{book},1]}, v_{[\mathsf{bib},0]}\ \}$. Notice that thanks to the edge labeling we avoid to backtrack also to node $v_{[\mathsf{author},2]}$.

Procedure Following-Sibling computes chains for the preceding-sibling navigational axis, by means of the Parent and Child procedures. Notice that the other procedure for sibling axis Preceding-Sibling just differ from this one by the test made in Line 5.

Algorithm 6: Procedure FOLLOWING-SIBLING

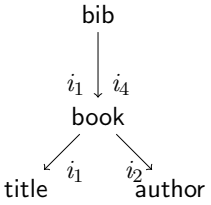
Input : Node set $M \subseteq V_{\mathsf{rv}}$, ϕ, id i
Output: Infer chains for `preceding-sibling`, `following-sibling` axes, side-effect on RET(), CLEAN(), V and E

```
1 foreach v ∈ M do
2     new_sibling ← false
3     PARENT({v}, *, i_v, vr⁺(i))
4     foreach v′ ∈ RET(i_v) do
5         foreach a following-sibling type of v.type that satisfies φ do
6             CHILD({v}, a, i)
7             new_sibling ← true
8     if !new_sibling then
9         CLEAN(i) ← CLEAN(i) ∪ {v}
```

To illustrate the procedure, consider chain inference for the following query q over DTD d_3 with $\mathsf{x} \mapsto \mathsf{bib}$.

$$\begin{aligned}\mathsf{q} = \ & \texttt{for } \mathsf{x}_1 \texttt{ in } \mathsf{x}_0//\mathit{title} \\ & \texttt{for } \mathsf{x}_2 \texttt{ in } \mathsf{x}_1/\texttt{following-sibling} :: \mathit{author} \\ & \texttt{for } \mathsf{x}_3 \texttt{ in } \mathsf{x}_2/\texttt{parent} :: * \\ & \texttt{return } \mathsf{x}_3/\texttt{following-sibling} :: \mathit{book}\end{aligned}$$

Below, we assume that $id(\mathsf{x}_i/\texttt{step})=i_{i+1}$. Chain inference for q gives the following CDAG.



$$\begin{aligned}\text{RET}(i_1) &= \{\, v_{[\mathsf{title},2]} \,\} \\ \text{RET}(i_2) &= \{\, v_{[\mathsf{author},2]} \,\} \\ \text{RET}(i_3) &= \{\, v_{[\mathsf{book},1]} \,\} \\ \text{RET}(i_4) &= \{\, v_{[\mathsf{book},1]} \,\}\end{aligned}$$

Chain inference for step $\mathsf{x}_2/\texttt{following-sibling} :: \mathit{author}$ is performed by backtracking on edges labeled as $i_1 \in vr^+(i_3)$ created by the previous descendant navigation, and then by adding node $v_{[\mathsf{author},2]}$ in the CDAG. After this, the parent navigation further backtracks to node $v_{[\mathsf{book},1]}$. Chain inference for step $\mathsf{x}_3/\texttt{following-sibling} :: \mathit{book}$ is performed again by backtracking on edges labeled with $i_1 \in vr^+(i_4)$.

Notice that the backward navigation in Line 3 makes this procedure inherently sound but not complete. Also, in the case where the schema enforces that a type a is both a following-sibling and a preceding-sibling of a type b (e.g., $\mathsf{r} \leftarrow \mathsf{a}, \mathsf{b}+, \mathsf{a}?$) we have that the result of chain inference for both the sibling axis is the same, if we start from some CDAG node labeled with b. Again this approximation is sound, but it may be not complete, because expressions like $\texttt{doc//b/following-sibling} :: \mathsf{a}$ and

Algorithm 7: Procedure INFER-QUERY

Input : Query q, G_{rv} and G_{e}
Output: Infer chains for q, side effect on G_{rv}, G_{e} and RET()

```
1  i ← id(q)
2  switch (q) do
3      case q = x/axis :: φ
4          call procedure computing x/axis :: φ
5      case q = q1, q2 or q = if (q0) then q1 else q2
6          foreach j ≤ 2 do
7              INFER-QUERY(qj)
8              ij ← id(qj)
9          RET(i) ← RET(i1) ∪ RET(i2)
10     case q = for/let x in/ := q1 return q2
11         INFER-QUERY(q1)
12         INFER-QUERY(q2)
13         RET(i) ← RET(i2)
14     case q = "txt"
15         BUILD(String, _, i)
16     case q = <a>q'</a>
17         INFER-QUERY(q')
18         i' ← id(q')
19         BUILD(a, RET(i'), i)
20     CLEAN-GRAPH(i)
```

`delete doc//a/preceding-sibling :: b` are deemed dependent, while they are not. However, this does not depend on how chain inference is implemented, but on the fact that chains abstract away from structural constraints concerning the horizontal order of data.

Procedure INFER-QUERY infers chains for whole query expressions. The procedure does essentially two things. It recursively infer chains for each subexpression of the query, and it computes used, return, and element chains for the whole query.

The base case of procedure INFER-QUERY consist of calling the proper axis navigation for a step of the form $\mathtt{x/axis} :: \phi$. Recall that for parent axis, the procedure also needs to pass $vr^+(\mathtt{x})$ as input. Return chains for q are set as return chains for the proper subexpression of q. Note that we do not need any particular extra structure for used chains of q, since they result to be marked as return chains of each q′ subexpression of q.

We assume that in the first call of the procedure all CDAGs G_{rv} and G_{e} are initialized, and the root node is added in G_{rv}. Finally, after chain inference for each subexpression

Algorithm 8: Procedure BUILD

Input : Type $\mathsf{a} \in \Sigma_{\mathsf{S}}$, node-set $M \subseteq V_{\mathsf{e}}$, expression id i
Output: Infer element chains for q, side effect on V_{e} and E_{e}

```
1 v_a ← (a, 0)
2 V_e ← V_e ∪ { v_[a,0] }
3 foreach v ∈ M do
4 |   v' ← COPY(v, 1, i)
5 |   E_e ← E_e ∪ (v_a, v', i)
6 |   if v ∈ V_rv then
7 |   |   ELTRET(i) ← ELTRET(i) ∪ { v' }
8 RET(i) ← RET(i) ∪ { v_a }
```

the CDAGs is done, dangling chains are removed by procedure CLEAN-GRAPH. This last operation correspond to the chain filtering made by the rule for iteration presented in the type system of Chapter 4.

Element chains are inferred by procedure BUILD. This procedure extends the CDAG of element chains G_{e}. The procedure is called either for string query "`txt`", or for an expression of the form <*a*>q</*a*>. Because we assume that constructed elements are never navigated (see Chapter 2), element chains in the G_{e} are simply stored as trees. This comports no relevant overhead.

The procedure first creates a fresh node representing the root of the constructed structure. Then, all nodes in M are copied (Line 4) and connected with v_{a} (Line 5).

Function COPY(v), despite not being explicitly modeled here, does three things. First, it distinguishes if the input node belongs to G_{rv} or to G_{e}. In the first case, it simply copies the node. In the second case, it copies the node and all the structure below. Second, it sets as 1 the height of the copied nodes, and accordingly of all copied descendants. Third, it marks edges of the copied structure with i.

Finally, according to the rule for element construction presented in Chapter 4, to record the fact that that a node v' copy of v should be meant as a whole subtree, we record v' in ELTRET(i). This has to be done only if the original node v belongs to the CDAG G_{rv}.

Algorithm 9: Procedure INFER-UPDATE

Input : Update u, CDAG G_{U}
Output: Infer chains for u, side effect on G_{U}

```
1  i ← id(u)
2  switch (u) do
3      case u = delete q0
4          INFER-QUERY(q0)
5          i0 ← id(q0)
6          G_U ← G_rv
7          UPD(i) ← UPD(i0)
8          DESCUPD(i) ← UPD(i0)
9      case u = rename q0 as a
10         INFER-QUERY(q0)
11         i0 ← id(q0)
12         M' ← RET(i0)
13         SIBLING(RET(i0), a, i0)
14         RET(i0) ← RET(i0) ∪ M'
15         G_U ← G_rv
16         UPD(i) ← UPD(i0)
17         DESCUPD(i) ← UPD(i0)
18     case u = insert q pos q0 or u = replace q0 with q
19         INFER-QUERY(q)
20         INFER-QUERY(q0)
21         i0 ← id(q0)
22         i1 ← id(q)
23         G_U ← G_rv
24         if pos ∈ into (as first|as last)? then
25             BUILD-UPDATE(RET(i1), RET(i0), i0, G_e)
26         else
27             M' ← RET(i0)
28             RET(i0) ← ∅
29             I ← vr+(ret(q0))
30             PARENT(M', *, i0, I)
31             BUILD-UPDATE(RET(i0), RET(e1), i0, G_e)
32         UPD(i) ← UPD(i0)
33         if u is a replace then
34             DESCUPD(i) ← UPD(i0)
```

Procedure INFER-UPDATE infers chains from an update expression $\mathtt{u}$. As updates are defined in terms of queries, this mainly consists of calling procedure INFER-QUERY for the update subexpressions.

For delete expressions, update chain inference is the same as query chain inference for the target expression $\mathtt{q}_0$. The update CDAG G_{U} is set as the query CDAG G_{rv}. Here, CDAG nodes in the prefix $\mathtt{c}'$ of an update chain of the form $\mathtt{c} : \mathtt{c}'$ are recorded by the inverted index UPD(). Furthermore it makes use of another index DESCUPD() thats records the fact that the whole subtrees rooted at RET(i_0) has to be considered as impacted by the update. This permits to avoid to infer all chains corresponding to descendants of nodes in RET(i_0).

Chain inference is similar for rename expressions, with the difference that chains capturing renamed node are inferred by means of a further navigation on the siblings of the return nodes for $\mathtt{q}_0$. Procedure SIBLING performs a following-sibling and a preceding-sibling navigation on the terminal nodes of $\mathtt{q}_0$.

Chain inference for insert and replace expressions are similar. They both combine chains inferred fro the source and for the target expression. The case where an insert update uses either position `after` or `before` is the same as for the replace command. In both cases it is necessary to perform a parent step, on the result locations of $\mathtt{q}_0$. In order to allow a backward navigation, the set of edges that can be visited has to be computed. This is done by first retrieving return subexpressions of $\mathtt{q}_0$ by means of function $ret()$ and $vr^+()$, as already outlined.

Chains recording the structure of inserted subtrees are inferred by procedure BUILD-UPDATE. The procedure puts an edge between terminal nodes of the target and the source query expressions. Nodes belonging to the source query are copied, setting also the level and the label of edges.

Algorithm 10: Procedure BUILD-UPDATE

Input : Target node-set $T \subseteq V_{\mathsf{U}}$, source node-set $S \subseteq V_{\mathsf{e}}$, expression id i, CDAG G_{e}

Output: Infer element chains for $\mathtt{q}$, side effect on V_{U} and E_{U}

```
1 foreach v ∈ T do
2     foreach v' ∈ S do
3         v'' ← COPY(v', v.level + 1, i)
4         V_U ← V_U ∪ { v'' }
5         E_U ← E_U ∪ (v', v'', i)
```

Algorithm 11: Function CHECKINDEPENDENCE

Input : CDAG for query G_{rv} and CDAG for update G_{U}, node sets $M_{\mathsf{r}}, M_{\mathsf{v}} \subseteq V_{\mathsf{rv}}$ and $M_{\mathsf{U}}, M'_{\mathsf{U}} \subseteq V_{\mathsf{U}}$

Output: Check for a common chain between two graphs

```
1  foreach h = 0..min(h_{G_rv}, h_{G_U}) do
2      foreach (v, v', i) ∈ E_rv and (v, v', j) ∈ E_U do
3          if ((i', j') ∈ I^{(v'',v)} and i' ∈ vr*(i) and j' ∈ vr*(j)) or h = 0 then
4              I^{(v,v')} ← I^{(v,v')} ∪ { (i, j) }
5              if v ∈ M_r ∪ M_v and v ∈ M_U then
6                  return false
7              if v ∈ M_r then
8                  if a descendant of v ∈ G_U also belongs to M_U then
9                      return false
10             if v ∈ M_U then
11                 if a descendant of v ∈ G_rv also belongs to M_r ∪ M_v (return-element)
                   then
12                     return false
13 return true
```

The last algorithm we present, is the one checking if query and update CDAGs have a conflicting chain. The procedure takes as input the set of terminal nodes denoting chains for used and return query chains, as well as for update nodes and for update and all descendant nodes. In particular, $M_{\mathsf{r}} \subseteq V_{\mathsf{rv}}$ is set of nodes that are terminal of query return chains, $M_{\mathsf{v}} \subseteq V_{\mathsf{rv}}$ is set of nodes that are terminal of query used chains, $M_{\mathsf{U}} \subseteq V_{\mathsf{U}}$ is set of nodes that are in the suffix of an update chain and $M'_{\mathsf{U}} \subseteq V_{\mathsf{U}}$ is set of nodes that are in the suffix of an update chain and whose whole subtree is impacted by the update.

The procedure iterates over all the levels of the CDAGs and inductively computes a product graph, where edges between nodes are sets of pairs of ids of query and update expressions that simulate the same chain.

At level h, a pair of codes (i, j) is added on the edge connecting a node v with a node v' (Line 4) only if it is the continuation of some chains computed at level $h-1$ (Line 3). Formally, this happens when for some node v'' parent of v there is a pair of codes (i', j') such that $i' \in vr^*(i)$ and $j' \in vr^*(j)$. Here $vr^*(i)$ denotes the reflexive and transitive closure of $vr(i)$. This test is trivial for $h = 0$ since the root has no incoming edge. The set $I^{(v,v')}$ denotes all pair of edges between v and v' in the product graph.

In order to detect conflicts between query and update chains, and hence to implement Definition 4.4.1, we perform the following three tests. If we reached a node v such that

v is both the terminal node of a used or return chain in G_{rv}, and an update node of G_{U}, then we raise a dependence. Otherwise, if v is only the terminal of a return chain, we visit G_{U} to search for a descendant of v that also belongs to M_{U}. If this node exists, we raise a dependence. Otherwise, if v is either an update node of M_{U} or an update node with all descendants of M'_{U} then we visit G_{rv} and we search for a descendant of v that also belongs to $M_{\mathsf{r}} \cup M_{\mathsf{v}}$. Of course, this is done by properly following edges corresponding to related subexpressions.

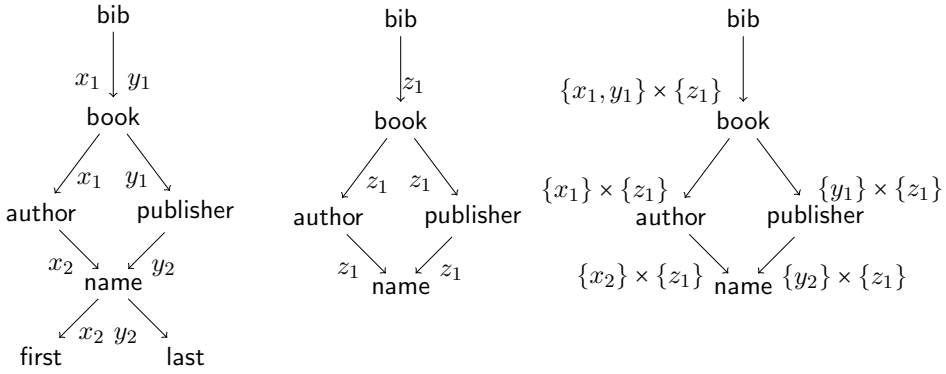
To illustrate, consider DTD $\mathtt{d}_{11}$ and the following query and updates.

$$\mathtt{q} = \texttt{for x}' \texttt{ in x}//\mathit{author} \texttt{ return x}'//\mathit{first} \quad , \quad \texttt{for y}' \texttt{ in y}//\mathit{publisher} \texttt{ return y}'//\mathit{last}$$

$$\mathtt{u}_1 = \texttt{delete z}//\mathit{name} \qquad \mathtt{u}_2 = \texttt{for z}' \texttt{ in z}//\mathit{publisher} \texttt{ return delete z}'//\mathit{first}$$

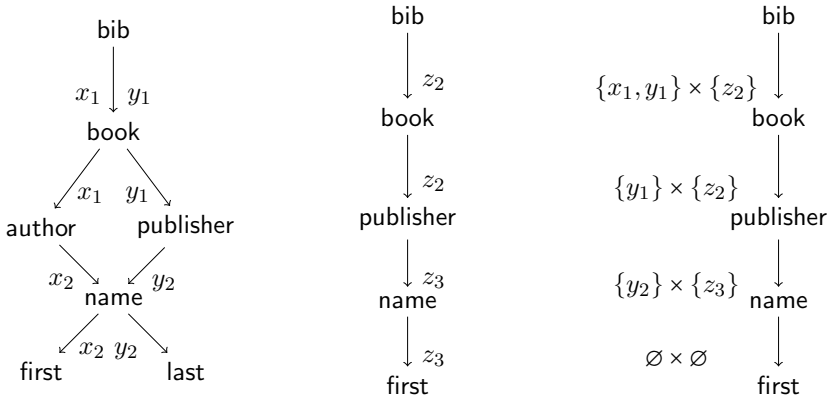
We run Algorithm 11 and show the result of the product graph for the pairs $\mathtt{q}$-$\mathtt{u}_1$ and $\mathtt{q}$-$\mathtt{u}_2$, respectively. Below, we assume that $\mathtt{x}, \mathtt{y}$ and $\mathtt{z}$ are all bound to bib. Furthermore, $id(\mathtt{x}//\mathit{author}) = x_1$, $id(\mathtt{x}'//\mathit{first}) = x_2$, $id(\mathtt{y}//\mathit{publisher}) = y_1$, $id(\mathtt{y}'//\mathit{last}) = y_2$, $id(\mathtt{z}//\mathit{name}) = z_1$, $id(\mathtt{z}//\mathit{publisher}) = z_2$ and $id(\mathtt{z}'//\mathit{first}) = z_3$.

The product graph for $\mathtt{q}$-$\mathtt{u}_1$ is the following.



In this case there are two conflicting chains because the query and the update access and impacts nodes labeled with *name*. After node $v_{[\mathsf{name},3]} \in G_{\mathsf{U}}$ has been recognized as a node in the prefix of an update chain, the procedure start searching for a descendant of node $v_{[\mathsf{name},3]} \in G_{\mathsf{rv}}$ that is also the terminal of a used or return chain, by following ids x_2, y_2. The procedure then reaches both $v_{[\mathsf{first},4]}$ and $v_{[\mathsf{last},4]}$ and hence a conflict is correctly raised by the procedure.

The product graph for q-u_2 is the following.



In this case there is no conflicting chain between the two expressions. Notice first that chains for the left subexpression of sequence q does not simulate any chain inferred for u. This is reflected by the fact that in the product graph there is no edge with id x_1 after node $v_{[\mathsf{book},1]}$. Similarly, also the right subexpression of sequence q does not simulated after node $v_{[\mathsf{name},3]}$ in the CDAG, because in the product graph there is no edge connecting $v_{[\mathsf{name},3]}$ with node $v_{[\mathsf{first},4]}$.

5.4 Complexity of Chain Inference with CDAGs

In this section we present some results on the complexity of algorithms defined in the previous section. We mainly distinguish between recursive and non-recursive schemas, and outline how complexity can be lowered for some interesting classes of queries and updates.

Theorem 5.4.1. *Let* d *be a non recursive DTD with n type definitions, and* e *a query or update expression of size m. By using Algorithms 1-10 of Section 5.3 chain inference can be done in $O(mn^3)$ space and $O(mn^5)$ time.*

Proof. When d is not recursive, the k value stop being determinant for the analysis since no label repeats twice in any schema chain. We first consider the case where e is not an insert or replace update. By Lemma 5.2.1, we know that the worst-case size of the CDAG inferred for e is $O(n^3)$. This, together with the fact that each edge can be labeled with a code, and that the number of codes is upper bounded by the size of the query entails that the CDAG needs at most $O(mn^3)$ space. Moreover, by using Algorithms 1-11 of Section 5.3 Chain inference for any navigational step, can be computed in the worst case with a traversal of the CDAG, hence in $O(n^3)$, for each input nodes. The number of input nodes is upper bounded by the number of nodes in the CDAG. By Lemma 5.2.1, there are at most $O(n^2)$ nodes in the CDAG. For the whole expression it follows that inference time is in $O(mn^5)$. If e is an insert or replace update, then by

making a cartesian product of chains for the target and source update expressions with Procedure BUILD UPDATE, we have an overhead which is at most quadratic in the number or terminal nodes of each expression. However, since we assumed that updates preserve the schema, this cannot result in more than $O(n^2)$ pairs of CDAG nodes corresponding to types satisfying the insert or replace expression. This overhead does not affect the asymptotical complexity. □

Corollary 5.4.2. *Let* d *be a non recursive DTD with n type definitions, and* e *a query or update expression of size m. By using Algorithms 1-10 of Section 5.3, the followings holds.*

1. *If* e *does not use step filter* node() *then chain inference can be done in* $O(mn^4)$ *time;*
2. *If* e *does not use recursive downward an backward axis then chain inference can be done in* $O(mn^2)$ *time;*
3. *If* e *does not use both step filter* node() *and recursive axis then chain inference can be done in* $O(mn)$*;*
4. *If each navigational step of* e *returns a fixed number of chains, then chain inference can be done in* $O(mn^3)$*.*

Proof. Analogous to the proof of Theorem 5.4.1. □

The above restriction are often met in practice, and in particular by expressions used in our testbed (when the recursive component of the XMark schema is not visited at all by the expression).

Theorem 5.4.3. *Let* d *be a recursive DTD with* $n + 1$ *type definitions, and* e *a query or update expression of size m. By using Algorithms 1-10 of Section 5.3 chain inference can be done in in* $O(mkn^3)$ *space and* $O(mk^2n^5)$ *time.*

Proof. We first consider the case where e is not an insert or replace update. By Lemma 5.2.2 we know that the worst-case size of the CDAG inferred for k-chains of e is $O(kn^3)$. Because each edge can be labeled with any code, the worst-case space occupancy for the CDAG is in $O(mkn^3)$. By using Algorithms 1-10 of Section 5.3, chain inference for any navigational step can be computed in the worst case with a traversal of the CDAG, hence in $O(kn^3)$, for each input nodes. The number of input nodes is upper bounded by the number of nodes in the CDAG. By Lemma 5.2.1, there are at most $O(kn^2)$ nodes in the CDAG. For the whole expression then inference time is in $O(mk^2n^5)$. If e is an insert or replace update, Procedure BUILD UPDATE introduce an overhead which is quadratic in the the size of the number or terminal nodes of each expression. At most $O(kn^2)$ nodes are inferred for each expression. However, since we assumed that updates preserve the schema, this cannot result in more than $O(mkn^3)$ pairs of CDAG nodes corresponding to types satisfying the insert or replace expression. Hence asymptotic complexity is not affected by this operation. □

Further restrictions on the the expression would lower again upper bounds.

Corollary 5.4.4. *Let* d *be a recursive DTD with* $n+1$ *type definitions, and* e *a query or update expression of size* m*. By using Algorithms 1-10 of Section 5.3, the followings holds.*

1. *If* e *does not use step filter* node() *then chain inference can be done in* $O(mk^2n^4)$ *time;*
2. *If* e *does not use recursive downward an backward axis then chain inference can be done in* $O(mkn^3)$ *time;*
3. *If* e *does not use both step filter* node() *and recursive axis then chain inference can be done in* $O(mkn^2)$ *time;*
4. *If each navigational step of* e *returns a fixed number of chains, then chain inference can be done in* $O(mkn^3)$ *time.*

Proof. Analogous to the proof of Theorem 5.4.1. □

As suggested by the statement complexity of chain inference can be lowered in many cases of practical relevance. In particular, if we assume that during chain inference each XPath step can have a fixed number of CDAG nodes as input, time complexity goes down to $O(mkn^3)$. The size of the input is likely to be close to 1 for most XPath steps used in practice. This holds in particular for XMark and XPathMark expressions. Another fact observable from such expressions is that they employ a small number of recursive navigations, thus further lowering the complexity.

Theorem 5.4.5. *Checking independence between two CDAGs inferred for a query* q *and an update* u *over a DTD* d*, can be done by Function 11 in* $O(km^2n^3)$ *time, where* $k = k_{\mathtt{q}} + k_{\mathtt{u}}$ *and* $m = \min\{\ |\mathtt{q}|, |\mathtt{u}|\ \}$.

Proof. Function 11 iterates on the levels of the CDAGs. Each CDAG has at most nk levels, and there are at most n^2 edges between the two levels. Moreover, each edge is labeled with at most m codes. Each code for an edge in the query CDAG may be paired with each code for the same edge in the update CDAG, and this can be done in $O(m^2)$ per edge. Putting everything together we have $O(km^2n^3)$. □

Conclusions

In this section we provided algorithms for efficiently implementing our independence analysis based on schema chains. We showed that chain inference is challenging from the computational point of view, because schema constraints can lead to infer an exponential number of chains. To overcome this limitation, we proposed a conservative implementation of the chain analysis that runs in polynomial space and time. In the next chapter experiments will show that the proposed CDAG algorithm is highly precise.

Chapter 6

Experiments

In this chapter we report on experiments made for validating the efficiency of the chain-based independence analysis. We describe the benchmarks we used as well as results obtained witnessing precision and efficiency of our technique.

6.1 Benchmarks and Experimental Settings

Our technique has been fully implemented in Java 6. We performed extensive experiments by using our Java implementation, in order to measure *i*) efficiency, *ii*) precision and *iii*) scalability of our static analysis. We used two different benchmarks: a first one based on XMark /XPathMark, and a second one, dubbed R-benchmark, we specifically designed to measure scalability, that are now described.

Independence benchmark We used a superset of the independence benchmark adopted by Benedikt and Cheney in [BC09a]. Our benchmark is composed of a set of 36 queries $\mathtt{v}_i$ and a set of 31 updates $\mathtt{u}_i$. A query belongs to either the XMark query set $\mathtt{q}_1$–$\mathtt{q}_{20}$ [SWK+02], or to the XPathMark query set $A1$–$A8/B1$–$B8$ [Fra05]. Queries in the group Ai of XPathMark only use downward axes, whereas Bi queries use upward and horizontal axes as well. Concerning updates, a first set corresponds to those used in [BC09a]. These are derived from the XPathMark query set $A1$–$A8/B1$–$B8$ and are of the form UAi=`delete` Ai or UBi=`delete` Bi.

It is worth observing that not all of the delete-updates preserve the schema, but as already seen (Section 4) this is safe because if a chain does not belong to the schema then it is not inferred for these updates. Instead, remaining updates are defined so as to preserve the schema. We added a set of 15 updates formed by insert expressions $UI1$–$UI5$, rename expressions $UN1$–$UN5$, and replace expressions $UP1$–$UP5$. These updates have been defined so as to cover all different types of nodes in XMark documents, and in particular those parts defined by mutually recursive types. It is worth remarking that, even if not all of the delete-updates of the testbed preserve the schema (see UA4, UA5, UA6, UA7, UA8, UB1, UB5, UB6, UB7, UB8), the correctness of our technique is still

ensured, since no new chain is created by these expressions. As outlined before, our technique is just unaware of new chains built by breaking schema constraints. In light of this, insert, rename and replace update expressions have been chosen in order to preserve document validity.

Before performing the tests, XMark and XPathMark expressions have been opportunely rewritten into expressions belonging to the XQuery fragment we consider (Section 3), as done in [BC10]. The rewriting essentially consists of: putting predicate conditions in disjunctive form, removing attribute use, and extracting paths from functions calls and arithmetic expressions. Clearly, the rewriting is such that a query and an update are independent if the rewritten query and update are. Updates extending XMark/XPathMark benchmark are reported in Table 6.2.

This benchmark has been used in order to measure precision and efficiency of our technique.

R-benchmark This benchmark is formed by schemas and expressions with a massive use of recursion. The benchmark is constituted of five saturated-recursive schemas with $1, 3, 5, 10$ and 20 recursive types, denoted by $\mathsf{d1}, \mathsf{d3}, \mathsf{d5}, \mathsf{d10}$, and $\mathsf{d20}$ respectively. In general, schema $\mathsf{d}n$ is defined as

$$\begin{array}{l} \mathsf{a}_0 \leftarrow (\mathsf{a}_1 | \ldots | \mathsf{a}_n)* \\ \mathsf{a}_1 \leftarrow (\mathsf{a}_1 | \ldots | \mathsf{a}_n)* \\ \quad \vdots \\ \mathsf{a}_n \leftarrow (\mathsf{a}_1 | \ldots | \mathsf{a}_n)* \end{array}$$

The benchmark also features a series of XPath expressions consisting on $1, 5$ and 10 consecutive $\texttt{descendant-or-self} :: \texttt{node}()$ steps, denoted by $\mathsf{e1}, \mathsf{e5}$ and $\mathsf{e10}$ respectively. In general, expression $\mathsf{e}n$ is defined as

$$\texttt{descendant-or-self} :: \texttt{node}()_1 / \ldots / \texttt{descendant-or-self} :: \texttt{node}()_n$$

This benchmark is designed for understanding the impact of recursion in the performances of our analysis, and draw conclusions on its scalability.

Configuration

We ran all tests on a desktop 4-core Intel Xeon 2.13 GHz machine with 8 GB RAM (the JVM was given 2 GB) running Linux. To avoid perturbations coming from system activity, we ran each experiment ten times, discarded the best and the worst performance, and computed the average of the remaining times.

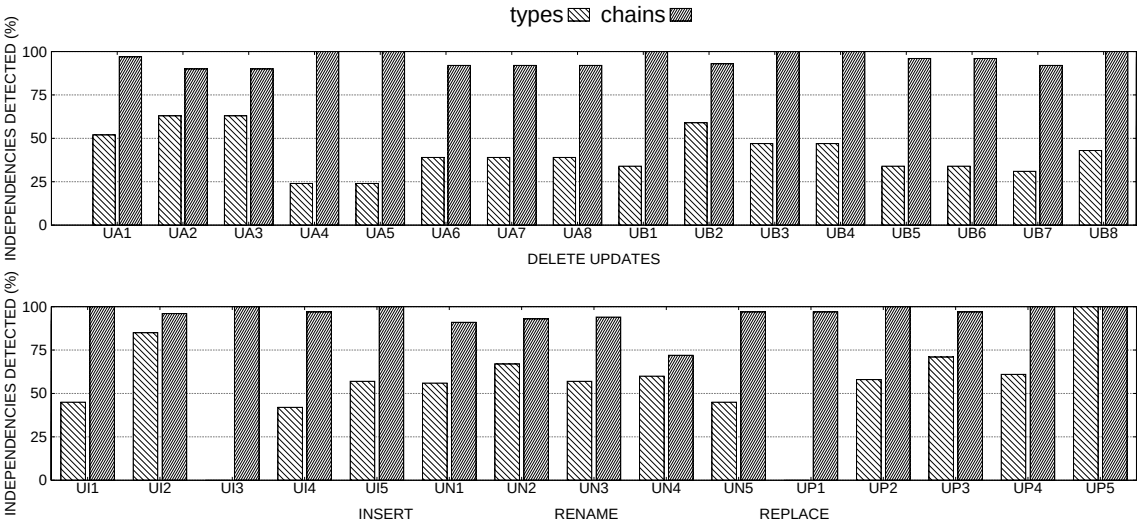


Figure 6.1: Static Analysis Precision: Types vs Chains

6.2 Experimental Results

Precision on XMark

Independence (Definition 2.5.2) is undecidable in general [BC09a], so for the purpose of measuring precision, for each update $\mathtt{u}_i$ we manually determined independent pairs $(\mathtt{u}_i, \mathtt{q}_j)$, and reported in Table 6.2 of the appendix. (note that for most pairs in the considered testbed independence is evident, so this process is much less time consuming than one may guess). We then express precision as the percentage of independent pairs that are deemed independent by our static analysis too. To estimate improvements wrt the alternative schema-based technique [BC09a] we computed the same percentages for that technique by using the public tool [Che09].

Results are reported in Figure 6.2.b. Our chain-based analysis turned out to be precise. Percentages goes from 72% to 100%, while the average precision is 96%. Also, Figure 6.2.b shows that the analysis proposed in [BC09a] (that has an average detection of 49%) is always outperformed in terms of precision by our static analysis, and in some cases improvements are huge. This happens in particular for updates UB1, UB5, UB6, UB8 (employing backward and horizontal axes). For these updates, the over-approximation made by type rules in [BC09a] entails a high number of false negatives. Our chain based inference instead is so precise to avoid most of these false-negatives. In general, improvements in terms of precision go from 8% (UN4) to 96% (UP1), and the average gain is 46%. In particular, precision of our analysis remains high in the presence of views using upward and horizontal axes (XPathMark queries in the group B). These queries are likely to be among the most expensive ones to re-evaluate after document updating.

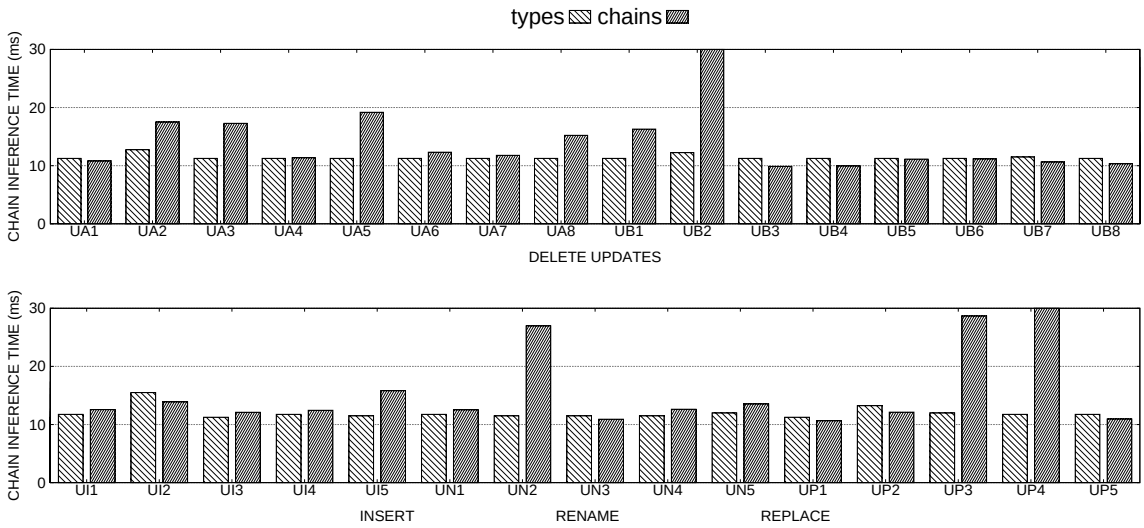


Figure 6.2: Static Analysis Time: Types vs Chains

As expected, we found that in some cases independence is not captured by our static analysis. This happens for pairs that make use of particular node position, identity or value conditions whose semantics is not precisely captured by our static analysis. Also, our system is not able to understand that the renaming performed by UN4 does not change any document tag, thus excluding independence in many cases (the same holds for [BC09a]).

Inference time on XMark

We measured the time needed by the static analysis to detect independence of each update wrt the whole set of XMark queries. The XMark schema is particularly suitable for testing the performances of our technique since the type dependency graph of this schema contains 5 mutually recursive types that form two cliques of size 2 and 3 respectively. We recall that the execution cost depends on the three parameters $|\mathsf{d}|$, $|\mathsf{e}|$ and k. In this testbed we have $|\mathsf{d}| = 76$, and $|\mathsf{e}| \leq 20$, while multiplicity values k range from 2 to 6. As observed in Section 5.4, in many cases chain inference can be substantially lowered.

Time values include the time for CDAGs inference and comparison, for each pair of expressions. Results are collected in Figure 6.2. It shows that the analysis is quite fast: in the worst case the analysis is performed in less than 40 ms for the whole set of queries, while the average cost is around 15 ms. According to complexity results of Section 5.4, inference time is influenced by i) the k values needed by a query-update pair and ii) the number of recursive types of the schema effectively unfolded. We see small changes in inference time values according to the k value (e.g., the pair UB1-UB2). Yet, two expressions having the same k value may have different time costs for chain inference,

depending on the effective number of recursive types unfolded by the analysis (e.g., the pair UI3-UP3).

Running times obtained from the available OCaml implementation [Che09] of the analysis presented in [BC09a] are rather close to ours: the average time for analyzing an update vs all of the queries is around 10 ms. It is worth observing, that inference time for [BC09a] has no sensible oscillations, while in our case inference time depends on k, hence on the query and update expressions. The analysis presented in [BC09a] has worst case time complexity $O((|\mathsf{d}|^2+|\mathsf{q}|)^2+|\mathsf{u}|)$, and thus is expected to be faster than our analysis in the presence of recursive schemas. Nevertheless, as shown shortly, our running times remain low enough to ensure high time savings in queries maintenance, even when queries are defined on relatively small documents.

Benefits of the Independence Analysis

Besides experimenting on our prototype, we verified that the precision and the efficiency of our method directly translate in time-savings when our static analysis is used to optimize view maintenance. Along the line of [BC09a], we simulated a view-maintenance scenario, where views are expressed as query, and on updates views are simply refreshed. We measured time savings obtained by avoiding the refreshing of views (queries) which our analysis deem as independent of an update.

We used three XQuery engines: Saxon 9.2EE, BaseX 7.0.1 and QizX 4.4 [1]. This, in order to make a faithful comparison of the three systems. We considered a 1MB XMark document and we scaled to 10MB and 100MB, in order to measure time savings in larger XML corpora. Our test results only take into account query answering time, dried of import time and indexing time. This, in order to make a faithful comparison of the three systems. Saxon is a main-memory engine, that uses data summaries for rapidly access data. Differently, BaseX and QizX are persistent system that exploit indexing. We settled the common *full-text* indexes for the latter engines. Document were updated off-line, in order to ensure query answering independent from update executions. For this experiment only, the JVM was given 4GB of RAM, in order to minimize memory swapping.

Results are reported in Figure 6.2.c. Refreshing time has been measured without import and indexing time (a full-text index has been settled for both BaseX and QizX), that we assume asymptotically constants since done only once, and updating and index-maintenance times, that is orthogonal to our goals. As in [BC09a], for each update u_i we measured the time r_i needed for refreshing *all* the 36 queries after the update, and the time r_i^{type} and r_i^{chain} needed to refresh only queries that are not deemed as independent by the static analysis of [BC09a] and by ours, respectively. In Figure 6.2.c, for each of the three used engines we report the averages of all refreshing times r_i, r_i^{type}

[1] www.saxproject.org, www.basex.org, www.xmlmind.com

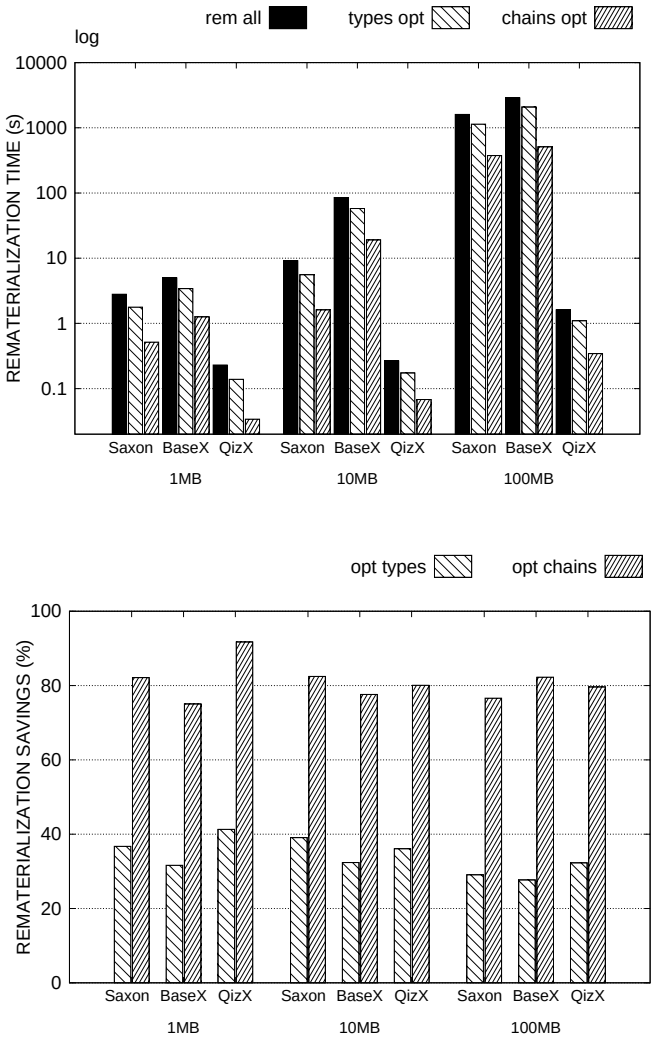


Figure 6.3: View Refreshing: Types vs Chains

r_i^{chain}. As a consequence of time efficiency and precision of our static analysis, even for a relatively small document of 1MB, our independence analysis ensures high time savings for all engines: 82% for Saxon, 75% for BaseX and 85% for QizX. While type based analysis [BC09a] ensures much lower time savings: 36% for Saxon, 31% for BaseX and 37% for QizX. These percentages are essentially the same as those obtained for 10MB and 100MB documents, both for our technique and for that of [BC09a]. This is because in the considered benchmark, queries that are not statically deemed as independent of an update, and hence refreshed, are the most expensive ones to refresh.

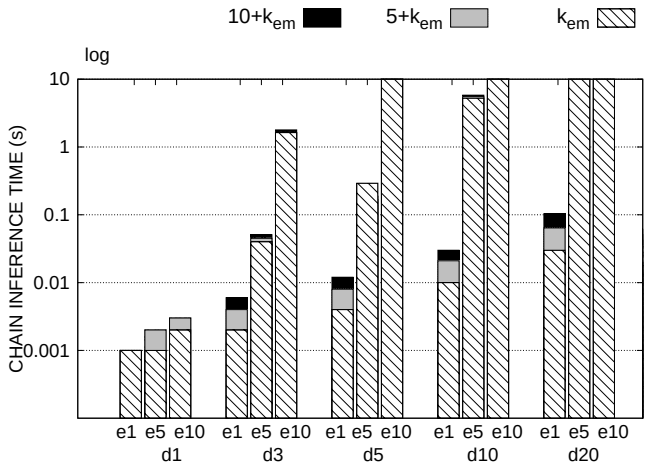


Figure 6.4: Chain Inference Time on R-benchmark

Scalability on R-benchmark

We ran experiments on the R-benchmark by ranging over each saturated recursive schema $\mathtt{d}n$ (n ranging over $\{1, 3, 5, 10, 20\}$), and each expression $\mathtt{e}m$ (m ranging over $\{1, 5, 10\}$). Also, for each run we varied the multiplicity values, by considering $k_{\mathtt{e}m}$, $k_{\mathtt{e}m} + 5$ and $k_{\mathtt{e}m} + 10$.

Results are now described. The schema $\mathtt{d5}$ is quite complex, it contains 5 mutually recursive types. We can see from Figure 6.2 that even with such complex form of recursion, for $\mathtt{e5}$, and for each $k \in \{5, 10, 15\}$, chain inference is still fast (inference time is around a decimal of a second). For schema $\mathtt{d10}$, featuring an extremely complex form of recursion, inference time is around five seconds for $\mathtt{e5}$, while for $\mathtt{e10}$ the time exceeds ten seconds. The same happens for more complex cases. These test results show that even for forms of recursions that are unlikely to occur in practice (like the $\mathtt{d5}$-$\mathtt{e5}$ case), chain inference is still fast, while it takes more than one second for extremely complex cases.

We see that augmenting k does not substantially raise the inference time for an expression. This explained by the fact that a larger k value just enlarges the space of returned chains, without resulting in more context switches (where chains for step i are passed as input of chains for step i+1) that dominate the cost of inference. We see that for $\mathtt{d10}$ and $\mathtt{d20}$, computing chains for $\mathtt{e5}$ is expensive, while for smaller queries the analysis can be afforded on such schemas. In such a case, even if unlikely to occur in practice, our analysis can still ensure substantial time savings in view maintenance and thus it may be worth running the analysis.

In this chapter we presented a set of experiments validating the precision and the efficiency of our independence analysis. On the XMark benchmark, our method resulted to have double precision as the related approach based on types and comparable inference time, thus validating the interest of using our analysis based on chains. Despite the fact

that our method runs in polynomial space and time, the non-linear complexity required us to perform a worst-case analysis, so as to outline when a massive use of recursion may slow down inference time. This confirmed the analysis is fast for a large class of schemas we find in practice.

A1	A2	A3	A4	A5	A6	A7	A8	B1	B2	B3	B4	B5	B6	B7	B8				
1	2	2	1	2	2	1	1	2	3	2	2	3	3	2	3				
Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20
1	1	2	1	2	3	3	1	2	1	1	1	1	2	2	2	1	1	2	1

Table 6.1: k values for XMark/XPathMark query and update expressions

UI1 $k = 3$	`for x in` *//person/people*[`not`(*homepage*)] `return` `insert` <*homepage*>{"www.myhome.com"}</*homepage*> `into x`	Insert a default node for unfilled homepages.

UI2
$k = 3$
	`let i` :=*//item*[1] `return`
	`for x in` *//regions* `return`
	`if`(`empty`(`x`/*item*)) `then insert i into x else` ()
		Insert the first document item into empty regions.

UI3 $k = 2$	`insert` *site/closed_ auctions/closed_ auction*[1] `into` */site/closed_ auctions*	Duplicate the first closed auction.
UI4 $k = 3$	`for x in` *//watches//watch* `return` `insert` <*watch open_ auction*={"open_auction5"}/> `after x`	Insert a node for each watch element.
UI5 k=1	`insert` "first_text_element" `as first into` *//text*[1]	Mark the first text element with a statement.
UN1 $k = 2$	`for x in` *//item/description/text/keyword* `return` `rename x with` *bold*	Rename some keywords elements as bold.
UN2 $k = 2$	`for x in` *//bold* `return` `rename x with` *emph*	Rename all bold elements as emph.
UN3 $k = 2$	`for x in` *//profile* `return` `rename x`/*education* `with` *interest*	Rename all education elements as interest.
UN4 $k = 2$	`for x in` *//item*[*payment*] `return` `rename x with` *item*	Rename some item elements as item.
UN5 $k = 3$	`for x in` *//person*[not(*address*) and *phone*] `return` `rename x`/*phone* `with` *address*	Rename some phone element as address.
UP1 $k = 2$	`replace` *//open_ auctions/open_ auction*[@*id*="open_auction0"] `with` ()	Replace an open auction with empty.
UP2 $k = 3$	`for x in` *//open_ auction*[@*id*="open_auction0"] `return` `replace x`/*bidder/personref*[@*id*="person1"] `with` <*personref id*={"person89"}/>	Replace a bidder in an open auction.

UP3
$k = 2$
	`for x in` *//emph* `return`
	`let y`:= `x/text() return`
	`replace x with` <*bold*>{`y`}</*bold*>
		Replace all emph elements with bold ones,
		yet preserving content.

UP4 $k = 3$	`for x in` *//text*[`count`(*//keyword*="fornitures")=0] `return` `replace x with` <*text*>{"element_removed"}</*text*>	Replace some text elements removing their content.
UP5 $k = 2$	`replace` () `with` *//keyword//bold*	Replace nothing.

Table 6.2: insert, rename and replace update expressions extending benchmark of [BC09a]

	A1	A2	A3	A4	A5	A6	A7	A8	B1	B2	B3	B4	B5	B6	B7	B8	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Types[BC09a]	Chains	TOT	Types[BC09a] (%)	Chains (%)
UA1	D	D	D	D	D				I	I			I	I	I						I	I	I	I	I					I	I	I			I		17	30	31	55	97
UA2	D	D	D	D	D					D			I	I	I						I	I	I							I	N	N			I		20	27	30	67	90
UA3	D	D	D	D	D					D			I	I	I						I	I	I							I	N	N			I		20	27	30	67	90
UA4	I	I	I	D	D					I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I		I	I	I		I	I		I	I		8	34	34	24	100
UA5	I	I	I	D	D					I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I		I	I	I		I	I		I	I		8	34	34	24	100
UA6		I	I		I	D	D	D	I	I			I	I	N		D				I	I	I	D	D	D	D	I	I	I	I		D		I	N	11	25	27	41	93
UA7		I	I		I	D	D	D	I	I			I	I	N		D				I	I	I	D	D	D	D	I	I	I	I		D		I	N	11	25	27	41	93
UA8		I	I		I	D	D	D	I	I			I	I	N		D				I	I	I	D	D	D	D	I	I	I	I		D		I	N	11	25	27	41	93
UB1		I	I		I	I	I	I	D	I			D	D	I						I	I	I	I	I	I	I	I	I	D	I		I		D	I	11	31	31	35	100
UB2	I	D	D	I	N					N			I	I	I						I	I	I						D	I	I	I			I		20	31	33	61	94
UB3		I	I		I					I	D	D	I		I	D	I	D	D	D	I	I	I				I	I	I		I			I		I	15	30	30	50	100
UB4		I	I		I					I	D	D	I		I	D	I	D	D	D	I	I	I				I	I	I		I			I		I	15	30	30	50	100
UB5		I	I		I	I	I	I	D	I			D	D	I						I	I	I	I	N	I	I	I	I	D	I		I		D	I	11	30	31	35	97
UB6		I	I		I	I	I	I	D	I			D	D	I						I	I	I	I	N	I	I	I	I	D	I		I		D	I	11	30	31	35	97
UB7		I	I		I	D	D	D	I	I			I	I	N		D				I	I	I	D	D	D	D	I	I	I	I		D		I	N	11	25	27	41	93
UB8		I	I		I					I	I	I	I		I	D	I	I	I		I	I	I				I	I	I		I			I		I	15	35	35	43	100
UI1		I	I		I	I	D	D		I			I		I						I	I	I	I	I	D	I	I	I		I		D			D	15	31	31	48	100
UI2		I							D	D			D	D	I						I	D	D	I	N				D	D	I				D		23	26	27	85	96
UI3	D	D	D	D	D	I	I	I	I	D	I	I	I	I	I	I	I	I	I	I	D	I	D	D	D	I	I	I	I	I	D	I	I	I	I	I	0	24	24	0	100
UI4		I	I		I	I	I	I		I			I		I		I					I	I	I	I	I	I	I	I		I		I			N	15	35	36	42	97
UI5	I	I	I	I	I					I			I	I	I						I	I	I						D	I	I	I			I		20	35	35	57	100
UN1	N	N	N	I	I					I			I	I	I						I	I	I						D	I	I	I			I		20	32	35	57	91
UN2	D	D	D	I	D					N			I	I	I						I	I	I							I	D	D			I		20	28	30	67	93
UN3		I	I		I	I				I			I		I						I	I	I			D	I	N	I		I					N	20	33	35	57	94
UN4		I							N	N			N	N	I						I	N		I	N					N	I				N		23	26	36	64	72
UN5		I	I		I	I	D	D		I			I		I						I	I	I	I	I	D	I	I	I		I			I		N	15	32	33	45	97
UP1	I	I	I	I	I	I	I	I	I	N	D	D	I	I	I	D	I	D	D	D	I	I	D	I	I	I	D	D	I	I	I	I	I	D	I	I	0	25	26	0	96
UP2		I								I	D	D	I		I	I	I	I	I	D	I	I					I	I	I		I			I			19	33	33	58	100
UP3	D	D	D	I	D					N			I	I	I						I	I	I						D	I	D	D			I		20	28	29	69	97
UP4	D	D	D	D	D				I	D			I	I	I						I	I	I	I	I				D	I	D	D			I		17	27	27	63	100
UP5																																					36	36	36	100	100
																																								49%	**96%**

Table 6.3: Query-Update independent pairs detected. D=dynamic dependence; I=independence detected by chains but not by types; N=independence undetected by both approaches; White space = independence detected by both approaches

Chapter 7

Extensions

In this chapter we illustrate how our chain-based type system can be extended in order to deal with features not considered in the previous formal development. We will first focus on additional features concerning queries and updates, and then on those concerning schema mechanisms.

7.1 Queries and Updates

The XQuery fragment we have considered, is the same as the one considered in the related approaches [BC09a, BC10], and leaves out several query mechanisms. These can be handled by extending our framework by either by means of query rewriting or by defining new inference rules.

7.1.1 Query rewriting

Query rewriting can be adopted to rewrite a query-update pair q-u into a pair $\mathtt{q}'$-$\mathtt{u}'$ belonging to the language we presented in Section 2.3, and such that static independence holds for q-u if it holds for $\mathtt{q}'$-$\mathtt{u}'$. Thus, here query rewriting is used wrt independence, not wrt to equivalence of XQuery expressions.

As already mentioned, rewriting has already been used in related approaches [BC09a, BCCN06] as well as in our tests for dealing with XMark queries in performed tests. For instance, XMark query $\mathtt{q}_{14}$

```
let $auction := doc("auction.xml") return
for $i in $auction/site//item
where
contains(string(exactly-one($i/description)),"gold")
return $i/name/text()
```

is rewritten into

```
let $auction := doc("auction.xml") return
for $i in $auction/site//item
if ($i/description/descendant-or-self::node())
 then $i/name/text()
 else ()
```

The rewriting replaces the `where` clause with an `if – then – else` clause; function calls are discarded and only the used paths are retained; note the adding of the step `descendant-or-self::node()` in order to capture the fact that discarded function calls needs descendants of nodes selected by the path $\$i/description$. Recall that symbol $ is used to denote a variable in XQuery.

Query rewriting is an obvious solution to deal with *conditional* queries of the form

$$\texttt{if (q) then } q_1 \texttt{ else } q_2$$

If we do not consider negation, the condition `q` is rewritten into a query $\texttt{q}'$, by replacing in `q` each operator $\theta \in \{\texttt{and}, \texttt{or}, =, <=\}$ with the query concatenation operator ",". For instance, let `x` and `y` be two variables, predicate

$$\texttt{q} = (\texttt{x}/a \texttt{ and } \texttt{x}/b) \texttt{ or } \texttt{x}/c/\texttt{text()}=\texttt{y}/f/\texttt{text()}$$

is rewritten into $\texttt{q}' = (\texttt{x}/a,\ \texttt{x}/b,\ \texttt{x}/c/\texttt{text()},\ \texttt{y}/f/\texttt{text()})$. It is easy to see that if an update does not impact $\texttt{q}'$ then it does not impact `q`.

Negation requires to take into account that an update modifying a node which is neither used nor returned by `q` can impact `not(q)`. For instance, assume variable `x` bound to a node having a and b children only, then an update deleting all a children impacts $\texttt{not}(\texttt{x}/b)$.

Negation is taken into account by rewriting `not(q)` into $\texttt{q}'$ by simply replacing each node-test in `q` with the node-test `node()`. This rewriting ensures a sound analysis with a loss of precision in some cases. For the previous example, $\texttt{not}(\texttt{x}/b)$ is rewritten into `x/node()`, enabling the chain analysis to exclude independence with the aforementioned deletion.

The above described rewriting is essentially the same used in [BCCN06] in the context of type-based projection for query optimization. According to the above illustrated rewriting, a projection for $\texttt{q}'$ is indeed a sound projection for `q` as well.

7.1.2 New Inference Rules

Query mechanisms left out can be handled by introducing new rules for inferring chains and by determining the k multiplicity value. For instance we can add new rules to handle

`where` and `order – by` clauses and builtin functions. These new rules are quite similar to the existing ones, and presented in Table 7.1 whereas rules for inferring the multiplicity k value are presented in Table 7.2.

For the `where` clause the extension consists of adding two new rules for `for` and `let` expressions containing a `where` clause. For the `order – by` clause a similar reasoning applies. For conciseness, we restrict to descending ordering (the parameter does not affect the analysis), and assume `where` always co-occurs with `order – by`.

Concerning *types switch* expressions like

```
typeswitch(q)
 case T1 return q1
          ⋮
 case Tn−1 return qn−1
default return qn
```

a sound and precise analysis can be ensured without involving types $\mathtt{T_i}$ in the analysis. The chain inference for these expressions is similar to the chain inference for `if` expressions.

During chain inference for each query $\mathtt{q}_i$, the type $\mathtt{T_i}$ could be taken into account to improve precision, by filtering out those chains inferred from `q` that can not be generated by any of the $\mathtt{T_i}$'s in the spirit of the precise type analysis provided in [BCF03] for pattern-matching in the ℂDuce language.

For *functions call* $\mathtt{f}(\mathtt{q}_1, \ldots, \mathtt{q}_n)$ new rules needs to take into account the nature of the function itself, and more specifically the way input parameters are used. Table 7.1 collects rules for several XQuery built-in functions used in the XMark query set that show how to deal with possible different cases.

For instance, the rule for `count(q)` is simple: it states that *used* chains of `count(q)` are those of `q` plus the return chains of `q`, and also that `count(q)` has no return chains. The return-to-used conversion is illustrated by the following example. Consider the query `count(x//`*employee*`)`, for the argument `x//`*employee* we infer return chains pointing to *employee* elements selected by `x//`*employee*.

Since counting these nodes does not depend on their descendants (e.g., *employee* has *salary* element as child), for a precise-analysis the return-to-used conversion is needed. Concerning element chains eventually inferred from the input `q`, these do not have to be considered as chains for `count(q)`. Recall that element chains are inferred for `return` clauses of queries inside updates, so that precise chain inference for insert/replace updates can be done.

For the function `string` (that converts a value or a node to a string) the rule is simpler: used and return chains of `string(q)` are those of `q`. In this case the conversion is

not needed due to the semantics of the `string` function: the descendants of return nodes of $\mathtt{q}$ are needed to compute the function output. Aggregation functions `sum`, `min`, `max` and `avg`, as well as conversion functions such as `data` and `number` are treated in the same way as `string`. Finally, `contains` function is defined similarly too, by collecting used and return chains of the queries passed as parameters.

For *user-defined functions calls* the rewriting method can be used as follows. Assuming that the definition of the function `f` is *non recursive* and given by

$$(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{q}$$

the function call $\mathtt{f}(\mathtt{q}_1, \ldots, \mathtt{q}_n)$ is rewritten into

$$\begin{array}{l} \mathtt{let}\ \mathtt{x}_1 := \mathtt{q}_1 \\ \qquad \vdots \\ \mathtt{let}\ \mathtt{x}_n := \mathtt{q}_n \\ \mathtt{return}\ \mathtt{q} \end{array}$$

for chain inference and independence analysis (of course the rewriting is recursively done inside `q` eventually). Dealing with recursive functions raises problems wrt to determining the multiplicity value k, and is out of the scope of this work.

Concerning *transform* expressions of the form

$$\begin{array}{l} \mathtt{copy}\ \mathtt{x} := \mathtt{q}_1 \\ \mathtt{modify}\ \mathtt{u}_1 \\ \mathtt{return}\ \mathtt{q}_2 \end{array}$$

we have to notice that they do not change the input document, but rather a copy of some of its fragments.

In order for a transform query to be independent of an update `u`, we need that $\mathtt{q}_1$, $\mathtt{q}_2$ and the 'query component' of $\mathtt{u}_1$ (the part of $\mathtt{u}_1$ that queries the schema instance) be independent of `u`. So chain inference for transform queries can be specified by relying on the existing chain inference for $\mathtt{q}_1$ and $\mathtt{q}_2$, while for inferring chains from $\mathtt{u}_1$ some modifications on update chain inference need to be done so that chains coming from the query component of `u` be taken into account. We leave this extension as future work. Note that the same kind of extension is needed for the related approach [BC09a].

Concerning the inference of the k multiplicity value for added constructs this is done along the lines of definitions for the core considered in Section 2.3. The extended definition is reported in Table 7.2.

$$\frac{\begin{array}{c}\Gamma \vdash_C q_1 : (r_1, v_1, e_1) \\ \Gamma[x \mapsto c] \vdash_C q^w : (r_c^w, v_c^w, e_c^w) \\ \Gamma[x \mapsto c] \vdash_C q^o : (r_c^o, v_c^o, e_c^o) \\ \Gamma[x \mapsto c] \vdash_C q_2 : (r_c, v_c, e_c) \quad \text{for any } c \in r_1\end{array}}{\Gamma \vdash_C \begin{array}{l}\texttt{for } x \texttt{ in } q_1 \\ \quad \texttt{where } q^w \\ \quad \texttt{order-by } q^o \\ \texttt{return } q_2\end{array} : (\bigcup_{c \in r_1} r_c,\ v_1 \cup \bigcup_{\substack{c \in r_1 \\ \lambda \in \{w,o\}}} (r_c^\lambda \cup v_c^\lambda) \cup \bigcup_{\substack{c \in r_1 \\ r_c \cup e_c \neq \emptyset}} (v_c \cup \{c\}),\ \bigcup_{c \in r_1} e_c)} \text{ (ForWhereOb)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_C q_1 : (r_1, v_1, e_1) \\ \Gamma[x \mapsto r_1] \vdash_C q^w : (r^w, v^w, e^w) \\ \Gamma[x \mapsto r_1] \vdash_C q^o : (r^o, v^o, e^o) \\ \Gamma[x \mapsto r_1] \vdash_C q_2 : (r, v, e)\end{array}}{\Gamma \vdash_C \begin{array}{l}\texttt{let } x := q_1 \\ \quad \texttt{where } q^w \\ \quad \texttt{order-by } q^o \\ \texttt{return } q_2\end{array} : (r,\ r_1 \cup \bigcup_{\lambda \in \{w,o\}} (r^\lambda \cup v^\lambda) \cup v_2,\ e_2)} \text{ (LetWhereOb)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_C q : (r_0, v_0, e_0) \\ \Gamma[x \mapsto r] \vdash_C q_i : (r_i, v_i, e_i) \quad \text{for any } i \in \{1, \ldots, n\}\end{array}}{\Gamma \vdash_C \begin{array}{l}\texttt{typeswitch } q \\ \quad \texttt{case } T_1 \quad \texttt{return } q_1 \\ \quad\quad : \\ \quad \texttt{case } T_{n-1} \texttt{ return } q_{n-1} \\ \quad \texttt{default} \quad \texttt{return } q_n\end{array} : (\bigcup_{i \in \{0 \ldots n\}} r_i,\ \bigcup_{i \in \{0 \ldots n\}} v_i,\ \bigcup_{i \in \{1 \ldots n\}} e_i)} \text{ (TypeSwitch)}$$

$$\frac{\Gamma \vdash_C q : (r, v, e)}{\Gamma \vdash_C \texttt{count}(q) : (\emptyset, r \cup v, \emptyset)} \text{ (Count)} \qquad \frac{\Gamma \vdash_C q : (r, v, e)}{\Gamma \vdash_C \texttt{string}(q) : (r, v, \emptyset)} \text{ (String)}$$

$$\frac{\Gamma \vdash_C q_i : (r_i, v_i, e_i) \quad \text{for } i = 1, 2}{\Gamma \vdash_C \texttt{contains}(q_1,\ q_2) : (r_1 \cup r_2, v_1 \cup v_2, \emptyset)} \text{ (Contains)}$$

Table 7.1: Chain Inference Rules for Extensions

$$\frac{f = \mathcal{F}(a, \mathtt{q}_1) + \max\{\,\mathcal{F}(a, \mathtt{q}_2), \mathcal{F}(a, \mathtt{q}^w), \mathcal{F}(a, \mathtt{q}^o)\,\}}{\mathcal{F}(a, \mathtt{for/let\ x\ q}_1\ \mathtt{where\ q}^w\ \ \mathtt{order{-}by\ q}^o\ \mathtt{return\ q}_2) = f}(\mathcal{F}\text{-ORDER-BY})$$

$$\frac{f = \mathcal{F}(a, \mathtt{q}) + \max\{\mathcal{F}(a, \mathtt{q}_i)\}}{\mathcal{F}(a, \mathtt{typeswitch\ q\ case\ T}_i\ \mathtt{return\ q}_i\ \ \mathtt{default\ return\ q}_n) = f}(\mathcal{F}\text{-TYPESWITCH})$$

$$\frac{f = \max\{\,\mathcal{F}(a, \mathtt{q}_1), \mathcal{F}(a, \mathtt{q}_2)\,\}}{\mathcal{F}(a, \mathtt{contains(q}_1\mathtt{,q}_2\mathtt{)}) = f}(\mathcal{F}\text{-CONTAINS})$$

$$\frac{f = \mathcal{F}(a, \mathtt{q})}{\mathcal{F}(a, \mathtt{string(q)}) = f}(\mathcal{F}\text{-STRING}) \qquad \frac{f = \mathcal{F}(a, \mathtt{q})}{\mathcal{F}(a, \mathtt{count(q)}) = f}(\mathcal{F}\text{-COUNT})$$

$$\frac{r = \mathcal{R}(\mathtt{q}_1) + \max\{\,\mathcal{R}(\mathtt{q}_2), \mathcal{R}(\mathtt{q}^w), \mathcal{R}(\mathtt{q}^o)\,\}}{\mathcal{R}(\mathtt{for/let\ x\ q}_1\ \mathtt{where\ q}^w\ \ \mathtt{order{-}by\ q}^o\ \mathtt{return\ q}_2) = r}(\mathcal{R}\text{-ORDER-BY})$$

$$\frac{r = \mathcal{R}(\mathtt{q}) + \max\{\,\mathcal{R}(\mathtt{q}_i)\,\}}{\mathcal{R}(\mathtt{typeswitch\ q\ case\ T}_i\ \mathtt{return\ q}_i\ \ \mathtt{default\ return\ q}_n) = r}(\mathcal{R}\text{-TYESWITCH})$$

$$\frac{r = \max\{\,\mathcal{R}(\mathtt{q}_1), \mathcal{R}(\mathtt{q}_2)\,\}}{\mathcal{R}(\mathtt{contains(q}_1\mathtt{,q}_2\mathtt{)}) = r}(\mathcal{R}\text{-CONTAINS})$$

$$\frac{r = \mathcal{R}(\mathtt{q})}{\mathcal{R}(\mathtt{string(q)}) = r}(\mathcal{R}\text{-STRING}) \qquad \frac{r = \mathcal{R}(\mathtt{q})}{\mathcal{R}(\mathtt{counts(q)}) = r}(\mathcal{R}\text{-COUNT})$$

Table 7.2: $\mathcal{F}(,)$ and $\mathcal{R}()$ definition for Extensions

7.2 Schemas

7.2.1 Attributes

As most of the studies on XQuery type systems [BCCN06, BC09a], we have excluded attribute types in our presentation, mainly because these does not introduce any particular issue. In any case, our implementation and our tests take attribute types into account.

To handle attribute types, the implementation considers chains possibly having a final label of the form $@\alpha$. A chain $\mathsf{c}.@\alpha$ is generated by a schema if an element pointed by c in a schema instance may have an attribute labeled as α (w.l.o.g, we assume the symbol @ not used in element tags defined by the schema). To infer such chains, the following new rule handling the `attribute` axis has been implemented:

$$\mathsf{A}_{\mathsf{C}}(\mathsf{c}, \texttt{attribute}) \stackrel{def}{=} \{\, \mathsf{c}.@\alpha \mid \mathsf{c}.@\alpha \in \mathsf{C} \,\}$$

Chain based independence is straightforwardly extended to handle the attribute extension. Furthermore, the inference of the k multiplicity value is not impacted by attribute types, as they cannot occur more than once in a chain.

7.2.2 Keys and foreign-keys integrity constraints

Concerning ID/IDREF constraints in DTDs, and key/keyref constraints in XSDs (studied in [AFL02]), we assume they are preserved by updates, as we assume that validity is preserved (Section 2.4). So, in order to ensure precise and sound independence analysis, chain inference does not need to consider these constraints. Our notion of static independence only concerns the type component of the schema, while these constraints pose restrictions on the values of attributes and elements in a document, and do not impact its structure.

To better illustrate, consider the following XMark $\mathtt{q}_8$ [SWK+02] query

```
let $auction := doc("auction.xml")
for $p in $auction/site/people/person
let $a :=
  for $t in $auction/site/closed_auctions/closed_auction
  where $t/buyer/@person = $p/@id
  return $t
return <item person = "$p/name/text()" > count($a) < /item >
```

Here ID/IDREF attributes are used, and selected by the following path expressions.

$$\mathtt{p}_1 = \mathtt{doc}(\text{"}auction.xml\text{"})/site/people/person/@id$$
$$\mathtt{p}_2 = \mathtt{doc}(\text{"}auction.xml\text{"})/site/site/closed\_auctions/closed\_auction/buyer/@person$$

Hence our chain inference system generates chains

site.people.person.@id

site.site.closed_auctions.closed_auction.buyer.@person

If an update u changes either an *id* or *person* attributes, then are generated update chains like

site.people.person : @id

site.site.closed_auctions.closed_auction.buyer : @person

These chains conflict with query chains, hence static independence is excluded.

Keyref constraints in XML Schema XML Schema provides powerful mechanisms to define keys and references formed by multiple attribute/element nodes. To this end, a fragment of XPath including downward axes and no predicates is used to specify the position and structure of keys and references. The example below is borrowed from the W3C specification [TBMM04].

```
< xs : key name = "regKey" >
<! − −vehicles are keyed by a pair of state and plate − − >
  < xs : selector xpath = ".//vehicle"/ >
  < xs : field xpath = "@state"/ >
  < xs : field xpath = "@plateNumber"/ >
< /xs : key >

< xs : keyref name = "carRef" refer = "regKey" >
  <! − − people"s cars are a reference − − >
  < xs : selector xpath = ".//car"/ >
  < xs : field xpath = "@regState"/ >
  < xs : field xpath = "@regPlate"/ >
< /xs : keyref >
```

Both declarations are made inside the definition of a `root` element (whose definition can be found in [TBMM04]). So the two selectors .//*vehicle* and .//*car* navigates starting from a *root* element, and respectively select the parents of the key $regKey$ and foreign key $carRef$. Both key and foreign keys consist of a pair of attributes selected by relative path expressions in the `xs:field` elements. Field paths navigate starting from nodes selected by respective selectors. An important distinction wrt DTDs is that a field path may also select subelements of elements selected by the selector.

It is worth observing that the specification of key and keyref constraints and the specification of the type of the documents are different in nature. The purpose of key and keyref is to constraint the values of the elements of documents valid wrt a schema and have no impact on their structure. These constraints are defined using paths navigating valid documents. As a consequence, as for DTDs, key and keyref constraints can be totally

ignored by the static analysis in order to ensure a sound and precise static detection of independence.

One of the nice features of our static analysis is that it can be directly extended so as to capture Extended DTDs, and thus it is suitable also for XML Schema and a core of RelaxNG type languages.

7.2.3 Extended DTDs

As outlined in Chapter 1, EDTDs differ from DTDs mainly because they allow to write several type definitions for the same element. In order to perform our chain analysis, we have to reflect this aspect while doing chain inference.

Recall that in an EDTD [PV00] we have a "type alphabet" $\Sigma' = \{\mathsf{a}_i | \mathsf{a} \in \Sigma\}$ and a mapping to a "tag alphabet" μ such that $\mu(\mathsf{a}_i) = \mathsf{a}$ for all $\mathsf{a}_i \in \Sigma'$. This captures the fact that two types differently indexed produce the same label but possibly different content models.

That said, it is sufficient to change the definition of reachability Definition 4.1.2, and of some step chain inference rules. Both changes are straightforward. Reachability should be redefined so as chains contain indexed symbols a_i instead of a, referring to different definitions of the same element tags.

Then, chain inference rule for step filtering has to be modified in order to deal with indexed symbols, as follows.

$$\mathsf{T}_\mathsf{C}(\,\mathsf{c.a}_i, \mathsf{b}\,) = \{\,\mathsf{c.a}_i \mid \mu(\mathsf{a}_i) = \mathsf{b}\}$$

Note that inference rules for axis chain inference remains unchanged.

Once chains have been inferred for a query and an update, to check query-update independence we use a slight variation of Definition 4.4.1, where we look for overlapping chains up to indexes i in chains symbols a_i's.

Notice that precision of the inference as well as complexity results remain unchanged for the EDTD case, since this can be simply seen as a schema with more definitions. Also $<_\mathsf{r}$ definition can be easily adapter for dealing with interleaving operator of Relax NG.

For EDTDs, the CDAG compression needs to be a bit more careful. In Figure 7.2.3 we write an EDTD d_{10}, featuring two definitions of element name, that are used in different contexts. Consider CDAG storing chains of d_{10} made by considering only the tag-name of types, depicted in (a). This way of representing chains would include some artifacts (chains that do not belong to the schema), such as the following ones.

book.author.name.String book.publisher.name.first.String book.publisher.name.last.String

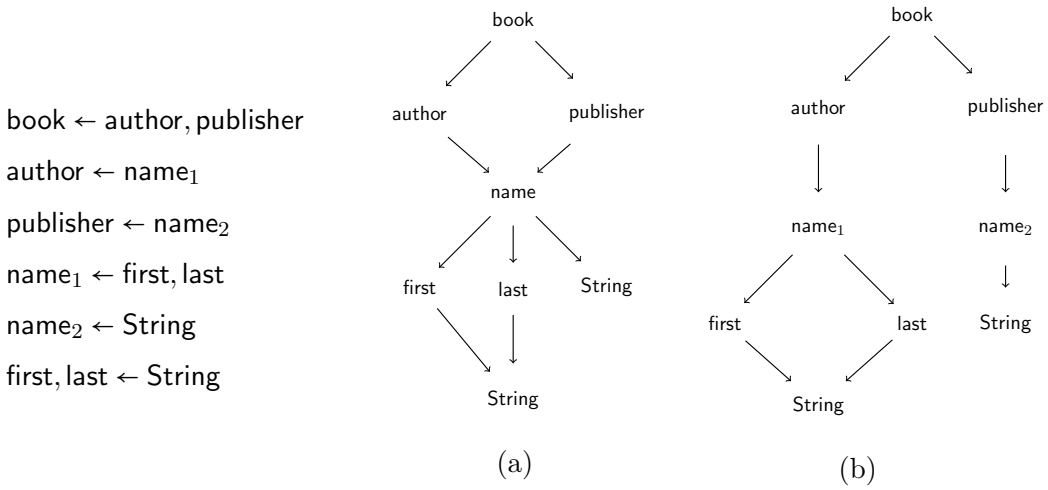


Figure 7.1: Example of CDAGs for EDTD

However, this situation can be simply avoided by considering that type name_1 and name_2 come from different definitions. The result is depicted in (b). We stress that types name_1 and name_2 are considered as different when we store chains, but as the same type name when we do independence detection. Notice that the CDAG inferred from an EDTD is again polynomial in the size of the EDTD.

Conclusions

In this chapter we discussed some extension of our static analysis based on schema chains. We would like to stress that dealing with extensions is easy because our chain inference and independence analysis are based on the notions of used, return and element nodes (Chapter 4). These are *universal* and *essential* notions, in the sense that each processed node of whatever kind of query one wants to add to the framework falls in one of these three categories. Thus, generalizing our framework for a new query construct mainly consists of identifying how used, return and element nodes are determined. This simply requires the understanding of the semantics of query constructs. In the next chapter we draw the conclusions of our work and we outline possible future perspectives.

Chapter 8

Conclusions and Future Perspectives

In this thesis we have tackled the problem of statically detecting query-update independence in the context of typed XML databases. After having outlined the important issues raised by this problem in the context of XML data management, we provided an exhaustive overview of related existing techniques, by including both typed an untyped approaches. As discussed, existing techniques still suffer of several limitations.

To overcome these limitations, in this thesis we made the following contributions:

- We presented a type system able to statically detect XML query-update independence. One of the main features of the type system is the chain inference component, allowing to infer information at the basis of an highly precise analysis.
- We proposed a method to restrict the analysis to a finite set of chains in the presence of recursive schemas.
- We proposed techniques for a sound and efficient implementation of the above mentioned finite analysis.
- We conducted extensive experiments in order to validate precision and efficiency of our static analysis. As shown by test results, our technique succeeded from both aspects.

We also illustrated how the static analysis can be extended in order to deal with several features not considered in the core query and update languages, in order to show that the analysis is amenable to be adopted in most of the XQuery query and update languages, by preserving precision and efficiency.

8.1 Future Perspectives

We discuss here some future directions that we believe are worth being investigated.

α-types

One of the main goals of this thesis is to show that schema information can be an extremely powerful tool for optimizing XML queries and updates. We presented a type system for inferring static projections of queries and updates (Theorems 4.3.4 and 4.3.6), used for the purpose of checking independence, that takes deeply into account informations concerning the hierarchical structure of data. We believe that a schema-based analysis, that is precise, cannot abstract away from such structural constraints. At the same time, when designing a method, there is a quest for tractability and efficiency of static analysis. Our method can run in polynomial time by introducing some approximations and has been proved fast by experiments. However particular applications may be interested in further lowering the computational time needed by the analysis. As shown in Chapters 3 and 6, one way to do this is to simplify the system and infer simple types [BC09a] rather than chains of types. To mitigate the lack of precision that this approach comports, it would be interesting to study inference for a *parametric* notion of type, called α-type. An α-type is a chain (not necessarily rooted) whose length is given by α (a positive natural value) and whose first $\alpha-1$ type labels determine the *context* of the type indicated by the α^{th} symbol, typing a set of nodes accessed by the query or update. The parameter α determines the amount of contextual information used to disambiguate types during the inference, and it can be opportunely tuned so as to vary the precision of the independence analysis wrt the computational resources that are available, because also the cost of the inference would result to be mainly determined by α. Note that with such an approach we would have that $\alpha = 1$ would enable to capture the type system in [BC09a], while unbounded α would enable to capture the infinite analysis here proposed. One of the main interesting aspects of this research consists of finding methods for computing optimal α values, by taking into account queries, updates, schemas, and required constraints about execution time.

Probabilistic Independence Analysis

The decision procedure that we developed to check static independence gives a binary answer yes/no. However, because our method performs a conservative analysis, a query and an update may still be independent over a schema instance, even if this has been excluded by the static analysis. This may happen for instance, because a conflict is detected on an optional element tag that occurs quite rarely in practice. This said, it seems interesting to refine our method so as to take into account schemas with probabilities. Probabilistic schemas are useful in many contexts. As discussed in [AAD+12], they can be used for instance to describe the distribution of the data, telling how many element nodes of a certain type are found in the database. By assigning to each chain an uncertainty value one could also estimate the probability that a conflict happened. This for

instance can be used to design more sophisticated view-maintenance algorithms that do maintenance only when the estimated probability of conflict is higher than a treshold.

Schema-less Analysis

Designed for the Web, XML documents do not always come with a schema. Besides relying on schema inference techniques such as the one proposed by Bex et al. [BNSV10], this case can be handled by performing chain inference starting from a data summary (e.g., DataGuide) in place of a schema. A DataGuide essentially is a superset of the chains belonging to the data, and can be efficiently inferred [GW97]. Our system rules can be easily adapted because of the chain based nature of a DataGuide.

Independence Analysis for the Semantic Web

Query-update independence for Semantic Web databases is an open problem. Semantic web databases are graphs written in RDF. Despite designed for tree-shaped data, we believe that our method can be applied also for RDF graphs by making an analysis of accessed classes of data, provided that an ontology or a data summary is available. Also, our method could be applied for n-SPARQL, the navigational language for RDF [PAG10].

Chapter 9

Proofs

In this chapter we prove the statements concerning the correctness of our chain analysis. The whole proof is organized in two main parts. In the first part we prove that the chain-based independence analysis is sound, in the case where a possibly infinite set of chains is inferred for an expression over a schema. In the second part we prove that the finite restriction to the subset of k-chains yields an analysis that is equivalent to the infinite one.

Proving that the possibly infinite chain analysis is sound mainly consists of showing that the sets of return, used, element and update chains inferred for an expression correctly abstract the data that is accessed/updated during the evaluation of the expression, over a valid schema instance. As a corollary of this we conclude the correctness of our notion of chain-based independence.

Proving that the finite restriction to the subset of k-chains yields an analysis that is equivalent to the infinite one. Consists of two main tasks. We first show that all chains can be reduced to k-chains. Then we show that any pair of conflicting chains can be reduced to a pair of conflicting k-chains. These proofs are developed by first introducing *linear-path queries* (see Section 9.2).

9.1 Soundness of Chain Inference in the Possibly Infinite Case

In this section we prove that our analysis makes a sound abstraction of accessed/updated data in the possibly infinite case. Because updates are defined in terms of queries, and queries in turn are defined by using navigational capabilities, the proof is structured as follows. We first prove correctness for XPath navigational steps. We then prove correctness for XQuery expressions. We finally prove correctness for XQuery Update expressions. Furthermore, we show that for XPath navigational steps chain inference is also complete.

9.1.1 Soundness and Completeness of Schema Type Relations

Correctness (and also completeness) of chain inference for XPath navigational steps relies on the correctness of the auxiliary relations defined to capture with types the parent-child and sibling relations among documents nodes, namely $\Rightarrow_{\mathsf{d}}$ and $<_{\mathsf{r}}$.

We show that $\Rightarrow_{\mathsf{d}}$ and $<_{\mathsf{r}}$ are both sound and complete. Soundness means that for any pair of parent-child (respectively sibling) locations belonging to a valid tree, there exists a correspondent pair of types in $\Rightarrow_{\mathsf{d}}$ (respectively in $<_{\mathsf{r}}$). Completeness means that these relations are also minimal, thus removing one pair from $\Rightarrow_{\mathsf{d}}$ or $<_{\mathsf{r}}$ we beak soundness.

Lemma 4.1.3 (Soundness of $\Rightarrow_{\mathsf{d}}$) *Let* d *be a DTD and* $t = (\sigma, l_t) \in \mathsf{d}$ *a valid tree. If* $(l, l') \in \mathit{Child}_t$ *then* $\mathsf{type}(l) \Rightarrow_{\mathsf{d}} \mathsf{type}(l')$.

Proof. Since $t \in \mathsf{d}$, by definition of validity (Section 2.2) it follows that $\mathsf{type}(l_2) \in \mathsf{d}(\mathsf{type}(l_1))$ and thus $\mathsf{type}(l_1) \Rightarrow_{\mathsf{d}} \mathsf{type}(l_2)$. □

Lemma 4.2.2 (Soundness of $<_{\mathsf{r}}$) *Let* d *be a DTD and* $t = (\sigma, l_t) \in \mathsf{d}$ *a valid tree. If* $(l, l') \in \mathit{FollowingSibling}_t$ *and* $\mathsf{c}_l^t = \mathsf{c.a}$ *then* $\mathsf{type}(l) <_{\mathsf{d(c)}} \mathsf{type}(l')$.

Proof. If $(l_1, l_2) \in \mathit{FollowingSibling}_t$ and $(l, l_1) \in \mathit{Child}_t$ and $\mathsf{r} = \mathsf{d}(\mathsf{c}_l^t)$ then, by definition of validity, there exists a word $w \in \mathsf{Lang}(\mathsf{r})$ where $\mathsf{type}(l_1)$ occurs before $\mathsf{type}(l_2)$. We show that this in turn implies that $\mathsf{type}(l_1) <_{\mathsf{r}} \mathsf{type}(l_2)$. By cases on r.

Base.

(Cases $\mathsf{r} = \epsilon$ and $\mathsf{r} = \mathsf{a}$) Trivial since such w cannot exist.

Induction.

(Case $\mathsf{r} = \mathsf{r}_1,\mathsf{r}_2$) Because $\mathsf{Lang}(\mathsf{r}_1, \mathsf{r}_2) = \mathsf{Lang}(\mathsf{r}_1) \cup \mathsf{Lang}(\mathsf{r}_2) \cup \mathsf{Lang}(\mathsf{r}_1) \times \mathsf{Lang}(\mathsf{r}_2)$ then either $w \in \mathsf{Lang}(\mathsf{r}_1) \cup \mathsf{Lang}(\mathsf{r}_2)$ or $w \in \mathsf{Lang}(\mathsf{r}_1) \times \mathsf{Lang}(\mathsf{r}_2)$. In the first case, we conclude by induction on $\mathsf{r}_1, \mathsf{r}_2$. In the second case, we conclude since $(\mathsf{type}(l_1), \mathsf{type}(l_2)) \in \mathsf{Sym}(\mathsf{r}_1) \times \mathsf{Sym}(\mathsf{r}_2) \subseteq <_{\mathsf{r}}$.

(Case $\mathsf{r} = \mathsf{r}_1|\mathsf{r}_2$) In this case we have that $\mathsf{Lang}(\mathsf{r}) = \mathsf{Lang}(\mathsf{r}_1) \cup \mathsf{Lang}(\mathsf{r}_2)$. Hence at least one of $\mathsf{type}(l_1) <_{\mathsf{r}_1} \mathsf{type}(l_2)$ and $\mathsf{type}(l_1) <_{\mathsf{r}_2} \mathsf{type}(l_2)$ hold, and we conclude by induction.

(Cases $\mathsf{r} = \mathsf{r}^+$ and $\mathsf{r} = \mathsf{r}^*$ and $\mathsf{r} = \mathsf{r}^?$) Immediate since $<_{\mathsf{r}_1^+} = <_{\mathsf{r}_1,\mathsf{r}_1}$ and $\mathsf{r}^* = \mathsf{r}^+|\epsilon$ and $\mathsf{r}^? = \mathsf{r}|\epsilon$.

□

Let $L = (l_1, \ldots, l_n)$ be a sequence of location, in the following, with a little abuse of notation, we write $\mathsf{type}(L)$ for denoting the word $\mathsf{type}(l_1), \ldots, \mathsf{type}(l_n)$.

Lemma 4.1.4 (Completeness of $\Rightarrow_{\mathsf{d}}$) *For all DTD* d*, if* $\mathsf{a} \Rightarrow_{\mathsf{d}} \mathsf{b}$ *then there exists a valid tree* $t = (\sigma, l_t) \in \mathsf{d}$*, and a pair of locations* $(l, l') \in \mathit{Child}_t$*, such that* $\mathsf{type}(l) = \mathsf{a}$ *and* $\mathsf{type}(l') = \mathsf{b}$.

Proof. *i*) Recall that the DTD d does not contain any type definition which is non-terminating or unreachable from the root. If $\mathsf{a} \Rightarrow_{\mathsf{d}} \mathsf{b}$ then there exists a chain $\mathsf{c} \in \mathsf{C}_{\mathsf{d}}$ of the form

$$\mathsf{a}_1 \ldots \mathsf{a}_{(n-1)}.\mathsf{a}_n$$

where $\mathsf{a}_1 = \mathsf{s}_{\mathsf{d}}$, $\mathsf{a}_{n-1} = \mathsf{a}$, and $\mathsf{a}_n = \mathsf{b}$. In order to conclude the thesis, we want to show that there exists a valid tree $t \in \mathsf{d}$ and a pair of locations $(l_1, l_2) \in \mathit{Child}_t$ such that

$$\mathsf{c}^t_{l_2} = \mathsf{a}_1 \ldots \mathsf{a}_n$$

Such t can be built with the following recursive procedure.

Start with $i = 1$.

1. (Generate an element of the desired chain)

   Let l be a location such that $\mathsf{type}(l) = \mathsf{a}_i$. Attach to l a finite sequence of locations L such that the following hold.

   - $\mathsf{type}(L) \in \mathsf{Lang}(\mathsf{d}(\mathsf{a}_i))$;
   - $\mathsf{type}(l') = \mathsf{a}_{(i+1)}$ for some $l' \in L$;

2. (Recursively generate the desired chain)

   For all $l' \in L$ such that $\mathsf{type}(l') = \mathsf{a}_{(i+1)}$ do the following. If $i < n$ then increment i by 1 and apply recursively step 1 on l'. Else, the desired chain of nodes is obtained. Go to the next step.

3. (Complete the tree so as to make it valid)

   For all $l' \in \mathit{dom}(t)$, such that $\mathit{Child}_t(l') = \varnothing$ and that $\epsilon \notin \mathsf{Lang}(\mathsf{type}(l'))$ do the following. Attach to l' a finite and valid sequence of locations L' such that let $\mathsf{r}' = \mathsf{d}(\mathsf{type}(l'))$

   - r'' is the regular expression obtained by replacing each $*$-guarded or optional subterm of r' with ϵ (e.g. $\mathsf{a}?|\mathsf{b}+, \mathsf{a}*$ becomes $\epsilon|\mathsf{b}+$);
   - $\mathsf{type}(L') \in \mathsf{Lang}(\mathsf{r}'')$

   Repeat this step until obtaining a valid tree.

This procedure always terminates producing a finite tree since the all definitions in d terminates. Furthermore, this tree contains a pair of locations $(l_1, l_2) \in \mathit{Child}_t$ such that $\mathsf{c}^t_{l_2} = \mathsf{a}_1 \ldots \mathsf{a}_n$, where of course $\mathsf{type}(l_1) = \mathsf{a}$ and $\mathsf{type}(l_2) = \mathsf{b}$. □

Lemma 4.2.3 (Completeness of $<_{\mathsf{r}}$) *For all DTD* d*, If* $\mathsf{a} <_{\mathsf{d(c)}} \mathsf{b}$ *then there exists a valid tree* $t = (\sigma, l_t) \in \mathsf{d}$*, and a pair of locations* $(l, l') \in \mathit{FollowingSibling}_t$*, such that* $\mathsf{type}(l) = \mathsf{a}$ *and* $\mathsf{type}(l') = \mathsf{b}$*, with* $\mathsf{c}_l^t = \mathsf{c.a}$ *and* $\mathsf{c}_{l'}^t = \mathsf{c.b}$*.*

Proof. By Lemma 4.1.4 we know that we can build a tree instance with a location l'' typed by a chain $\mathsf{c} \in \mathsf{C_d}$. If $\mathsf{a} <_{\mathsf{d(c)}} \mathsf{b}$ we claim that we can build a sequence of locations $l_1, \ldots, l_n$ such that the word $\mathsf{type}(l_1), \ldots, \mathsf{type}(l_n)$ belongs to the language of $\mathsf{d(c)}$, and for some locations l_i and l_j with $1 \leq i < j \leq n$ we have that $\mathsf{type}(l_i) = \mathsf{a}$ and $\mathsf{type}(l_j) = \mathsf{b}$. We show this by induction on the structure of $\mathsf{r} = \mathsf{d(c)}$.

Base.

(Cases $\mathsf{r}{=}\epsilon$ and $\mathsf{d(c)}{=}\mathsf{a}$) Trivial since they do not satisfy the hypothesis.

Induction.

(Case $\mathsf{r} = \mathsf{r}_1{,}\mathsf{r}_2$) If it holds that $\mathsf{a} <_{\mathsf{r}_1} \mathsf{b} \cup \mathsf{a} <_{\mathsf{r}_2} \mathsf{b}$ then we conclude by induction. Otherwise, because $\mathsf{a} <_{\mathsf{d(c)}} \mathsf{b}$ we have that $\mathsf{a} \in \mathsf{Sym}(\mathsf{r}_1)$ and $\mathsf{b} \in \mathsf{Sym}(\mathsf{r}_2)$. Therefore there exist two words $w_1 \in \mathsf{Lang}(\mathsf{r}_1)$ and $w_2 \in \mathsf{Lang}(\mathsf{r}_2)$ such that $\mathsf{a} \in w_1$ and $\mathsf{b} \in w_2$. At this point take a sequence of locations L such that $\mathsf{type}(L) = w_1 w_2$.

(Case $\mathsf{r} = \mathsf{r}_1|\mathsf{r}_2$) In this case it holds that $\mathsf{a} <_{\mathsf{r}_1} \mathsf{b} \cup \mathsf{a} <_{\mathsf{r}_2} \mathsf{b}$ and we conclude by induction.

(Cases $\mathsf{r} = \mathsf{r}^+$ and $\mathsf{r} = \mathsf{r}^*$ and $\mathsf{r} = \mathsf{r}^?$) Immediate since $<_{\mathsf{r}_1^+} = <_{\mathsf{r}_1,\mathsf{r}_1}$ and $\mathsf{r}^*{=}(\mathsf{r}^+|\epsilon)$ and $\mathsf{r}^?{=}(\mathsf{r}|\epsilon)$.

To conclude the thesis it suffices to complete the tree so as to obtain a valid instance. This can be done by applying the procedure described at point 3 of Lemma 4.1.4. □

9.1.2 Axis chain inference

In this section we prove that chain inference for XPath navigational is sound.

Lemma 9.1.1 (Soundness of Step Chain Inference)**.** *Let* d *be a DTD,* $t = (\sigma, l_t) \in \mathsf{d}$ *be a valid tree, and* $l \in \mathit{dom}(\sigma)$ *be a location of the tree. If*

$$\sigma, (\mathtt{x} := l_x) \vDash \mathtt{x/axis} :: \phi \;\Rightarrow\; \sigma, L$$

then for each location $l' \in L$ *we have that*

$$\mathsf{c}_l^\sigma \in \boldsymbol{T}_{\mathsf{C}}(\boldsymbol{A}_{\mathsf{C}}(\mathsf{c}_{l_x}^\sigma, \mathtt{axis}), \phi)$$

Proof. By case analysis on `axis`. Since `node()` is used as test filtering, we consider the correspondent axis evaluation and we leave as implicit the set-to-sequence conversion of rule by mean of document order.

[`axis = self`] We conclude immediately since by definition

$$[\![\mathtt{self}]\!]_{l_x}^\sigma = \{\, l_x \,\} \quad \text{and} \quad \mathbf{A}_{\mathsf{C_d}}(\mathsf{c}_{l_x}, \mathtt{self}) = \{\, \mathsf{c}_{l_x} \,\}$$

[$\texttt{axis} = \texttt{child}$] By definition

$$[\![\texttt{child}]\!]^{\sigma}_{l_x} = \mathit{Child}_{\sigma}(l_x)$$

Since the document is valid wrt the DTD d, for all $l \in \mathit{Child}_{\sigma}(l_x)$, let $\mathsf{a}_x = \mathsf{type}(l_x)$ and $\mathsf{a} = \mathsf{type}(l)$ we have that $\mathsf{a}_x \Rightarrow_{\mathsf{d}} \mathsf{a}$, with $\mathsf{a}_x \in \Sigma$ and $\mathsf{a} \in \Sigma_{\mathsf{S}}$. Therefore the type chain of the location l is $\mathsf{c}_l = \mathsf{c}_{l_x}.\mathsf{a}$ and by definition $\mathsf{c}_{l_x}.\mathsf{a} \in \mathsf{C}_{\mathsf{d}}$. We conclude since

$$\mathbf{A}_{\mathsf{C}_{\mathsf{d}}}(\mathsf{c}_{l_x}, \texttt{child}) = \{\ \mathsf{c}_{l_x}.\mathsf{a} \mid \mathsf{c}_{l_x}.\mathsf{a} \in \mathsf{C}_{\mathsf{d}}\ \}$$

[$\texttt{axis} = \texttt{descendant}$] By definition we have that

$$[\![\texttt{descendant}]\!]^{\sigma}_{l_x} = \mathit{Descendant}_{\sigma}(l_x)$$

For all $l \in \mathit{Descendant}_{\sigma}(l_x)$ let $L = (l_1, l_2, \ldots, l_n)$ be the sequence of nodes such that $(l_i, l_{i+1}) \in \mathit{Child}_{\sigma}$ for $i = 1..n{-}1$, with $l_1 = l_x$ and $l_n = l$. Since the document is valid wrt the DTD d, for all pair of locations $l_i, l_{i+1} \in L$, let $\mathsf{a}_i = \mathsf{type}(l_i)$, we have that $\mathsf{a}_i \Rightarrow_{\mathsf{d}} \mathsf{a}_{i+1}$.Therefore $\mathsf{a}_{l_x} \Rightarrow^{+}_{\mathsf{d}} \mathsf{a}_l$, and the type chain of the location l is $\mathsf{c}_l = \mathsf{c}_{l_x}.\mathsf{a}_2.\mathsf{a}_3 \ldots \mathsf{a} \in \mathsf{C}_{\mathsf{d}}$. We conclude since

$$\mathbf{A}_{\mathsf{C}_{\mathsf{d}}}(\mathsf{c}_{l_x}, \texttt{descendant}) = \{\ \mathsf{c}_{l_x}.\mathsf{c} \mid \mathsf{c}_{l_x}.\mathsf{c} \in \mathsf{C}_{\mathsf{d}}\ ,\ \mathsf{c} \neq \epsilon\ \}$$

Notice that case $\texttt{axis} = \texttt{descendant-or-self}$ is analogous.

[$\texttt{axis} = \texttt{parent}$] By definition

$$[\![\texttt{parent}]\!]^{\sigma}_{l_x} = \mathit{Parent}_{\sigma}(l_x)$$

Since the document is valid wrt the DTD d, for $l \in \mathit{Parent}_{\sigma}(l_x)$, let $\mathsf{a}_x = \mathsf{type}(l_x)$ and $\mathsf{a} = \mathsf{type}(l)$ we have $\mathsf{a} \Rightarrow_{\mathsf{d}} \mathsf{a}_x$. Therefore $\mathsf{c}_{l_x} = \mathsf{c}.\mathsf{a}_x$, where the type chain of the location l is $\mathsf{c}_l = \mathsf{c}$. We conclude since

$$\mathbf{A}_{\mathsf{C}_{\mathsf{d}}}(\mathsf{c}_l.\mathsf{a}_x, \texttt{parent}) = \{\ \mathsf{c}_l\ \}$$

[$\texttt{axis} = \texttt{ancestor}$] This case is similar to the case of $\texttt{descendant-or-self}$ axis.

By definition we have that

$$[\![\texttt{ancestor}]\!]^{\sigma}_{l_x} = \mathit{Ancestor}_{\sigma}(l_x)$$

Given $l \in \mathit{Ancestor}_{\sigma}(l_x)$, let $L = (l_1, l_2, \ldots, l_n)$ be the sequence of nodes such that $(l_i, l_{i+1}) \in \mathit{Child}_{\sigma}$ for $i = 1..n{-}1$ with $l_1 = l$ and $l_n = l_x$. Since the document is valid wrt the DTD d, for all pair of locations $l_i, l_{i+1} \in L$, let $\mathsf{a}_i = \mathsf{type}(l_i)$ we have that $\mathsf{a}_i \Rightarrow_{\mathsf{d}} \mathsf{a}_{i+1}$ and $\mathsf{a}_i \in \Sigma_{\mathsf{S}}$. Therefore $\mathsf{a}_l \Rightarrow^{+}_{\mathsf{d}} \mathsf{a}_{l_x}$ and the type chain of location l is $\mathsf{c}_{l_x} = \mathsf{c}_l.\mathsf{a}_2.\ldots.\mathsf{a} \in \mathsf{C}_{\mathsf{d}}$. We conclude since

$$\mathbf{A}_{\mathsf{C}_{\mathsf{d}}}(\mathsf{c}_{l_x}, \texttt{ancestor}) = \{\ \mathsf{c} \mid \mathsf{c}_{l_x} = \mathsf{c}.\mathsf{c}'\ ,\ \mathsf{c}', \mathsf{c}'' \neq \epsilon\ \}$$

Notice that case $\texttt{axis} = \texttt{ancestor-or-self}$ is analogous.

[$\texttt{axis} = \texttt{following-sibling}$] By definition we have that

$$[\![\texttt{following-sibling}]\!]^{\sigma}_{l_x} = \mathit{FollowingSibling}_{\sigma}(l_x)$$

Let l_p be the parent of l_x, i.e., $l_p \in \mathit{Parent}_{\sigma}(l_x)$. Since the document is valid wrt the DTD d, let $\mathsf{a}_x = \mathsf{type}(l_x)$ and $\mathsf{a}_p = \mathsf{type}(l_p)$ we have that $\mathsf{a}_p \Rightarrow_{\mathsf{d}} \mathsf{a}_x$. Therefore $\mathsf{c}_{l_x} = \mathsf{c}_{l_p}.\mathsf{a}_x \in \mathsf{C}_{\mathsf{d}}$. Since the document is valid wrt the DTD d, for $l_f \in \mathit{FollowingSibling}_{\sigma}(l_x)$, let $\mathsf{a}_f = \mathsf{type}(l_f)$ we have that $\mathsf{a}_p \Rightarrow_{\mathsf{d}} \mathsf{a}_f$ and $\mathsf{a} <_{\mathsf{d}(\mathsf{c}_{l_p})} \mathsf{b}$. Therefore $\mathsf{c}_{l_f} = \mathsf{c}_p.\mathsf{a}_f \in \mathsf{C}_{\mathsf{d}}$. We conclude since by definition

$$\mathbf{A}_{\mathsf{C}_{\mathsf{d}}}(\mathsf{c}_{l_x}, \texttt{following-sibling}) = \{\ \mathsf{c}_p.\mathsf{a}_f \in \mathsf{C}_{\mathsf{d}} \mid \mathsf{c}_{l_x} = \mathsf{c}_p.\mathsf{a}_x \ \wedge\ \mathsf{a}_x <_{d(\mathsf{c}_{l_p})} \mathsf{a}_f\ \}$$

Notice that case $\texttt{axis} = \texttt{preceding-sibling}$ is analogous.

□

Test chain inference

Soundness of chain inference for test filtering is stated by the following theorem. For simplicity, we present it together with completeness.

Lemma 9.1.2 (Test Filtering Soundness and Completeness)**.** *If* $\sigma, (\mathtt{x} := l_x) \vDash \mathtt{x/self} :: \phi \Rightarrow \sigma, L$ *then*

$$\boldsymbol{\mathcal{T}}_{\mathsf{C}_{\mathsf{d}}}(\mathsf{c}^{\sigma}_{l_x}, \phi) = \{\ \mathsf{c}^{\sigma}_{l_x} \mid l_x \in L\ \}$$

Proof. By case analysis on ϕ. Since $\texttt{self}$ is used as navigational axis, we directly translate the judgment into the proper test filtering $[\![\phi]\!]^{l_x}_{\sigma}$. Also note that the sequence L is equal to l_x alone when the test is evaluated to true and L is empty otherwise.

- [$\phi = \mathsf{a} \in \Sigma$] By definition of node test filtering we have that $[\![\mathsf{a}]\!]^{l_x}_{\sigma} = \{\ l_x \mid l_x \leftarrow \mathsf{a}[L] \in \sigma\ \}$ and we conclude immediately completeness since $\mathbf{T}_{\mathsf{C}_{\mathsf{d}}}(\mathsf{c}_{l_x}, \mathsf{a}) = \{\ \mathsf{c}_{l_x} \mid \mathsf{c}_{l_x} = \mathsf{c}.\mathsf{a}\ \}$.
- [$\phi = \texttt{text}()$] As before, completeness follows by definition since $[\![\mathsf{a}]\!]^{l_x}_{\sigma} = \{\ l_x \mid l_x \leftarrow$ "$\texttt{txt}$"$\ \}$ and $\mathbf{T}_{\mathsf{C}_{\mathsf{d}}}(\mathsf{c}_{l_x}, \mathsf{a}) = \{\ \mathsf{c}_{l_x} \mid \mathsf{c}_{l_x} = \mathsf{c}.\mathsf{String}\ \}$.
- [$\phi = \texttt{node}()$] By definition $[\![\phi]\!]^{\sigma}_{l_x} = l_x$ and $\mathbf{T}_{\mathsf{C}_{\mathsf{d}}}(\mathsf{c}_{l_x}, \texttt{node}()) = \{\mathsf{c}_{l_x}\}$.

□

9.1.3 Completeness of step chain inference

Lemma 9.1.3 (Completeness of Step Chain Inference)**.** *Let* d *be a DTD and* $\mathsf{c} \in \mathsf{C_d}$ *a chain. If*

$$\mathsf{c}' \in \boldsymbol{T}_{\mathsf{C_d}}\big(\boldsymbol{A}_{\mathsf{C_d}}(\mathsf{c}, \mathtt{axis}), \mathtt{node}()\big)$$

then there exists a valid tree $t = (\sigma, l_t) \in \mathsf{d}$ *and a location* $l_x \in dom(t)$ *such that* $\mathsf{c}_{l_x} = \mathsf{c}$ *and, assuming that* $\sigma, (\mathtt{x} := l_x) \vDash \mathtt{x}/\mathtt{axis} :: \mathtt{node}() \Rightarrow \sigma, L$ *we have that* $\mathsf{c}^{\sigma}_{l'} = \mathsf{c}'$*, for some* $l' \in L$*.*

Proof. As before, since `node()` is used as test filtering, we directly translate the judgment into the proper axis evaluation and we leave as implicit the set-to-sequence conversion by mean of document order. The we conclude by a case analysis on `axis`.

[`axis = self`] By Lemma 4.1.4, for all input chains c then there exists a valid tree $t = (\sigma, l_t) \in \mathsf{d}$ and a location $l_x \in dom(t)$ such that $\mathsf{c}_{l_x} = \mathsf{c}$. We conclude immediately since by definition we have that $\mathbf{A}_{\mathsf{C_d}}(\mathsf{c}_{l_x}, \mathtt{self}) = \{\ \mathsf{c}_{l_x}\ \}$ and $[\![\mathtt{self}]\!]^{\sigma}_{l_x} = \{\ l_x\ \}$.

[`axis = child`] By definition

$$\mathbf{A}_{\mathsf{C_d}}(\mathsf{c}, \mathtt{child}) = \{\ \mathsf{c.a} \mid \mathsf{c.a} \in \mathsf{C_d}\ \}$$

By Lemma 4.1.4, for all input rooted chains $\mathsf{c.a} \in \mathsf{C_d}$ then there exists a valid tree $t = (\sigma, l_t) \in \mathsf{d}$ and a location $l \in dom(t)$ such that $\mathsf{c}_l = \mathsf{c.a}$. To conclude it suffices to fix $\mathsf{c}_{l_x} = \mathsf{c}$.

[`axis = descendant`] By definition we have that

$$\mathbf{A}_{\mathsf{C_d}}(\mathsf{c}, \mathtt{descendant}) = \{\ \mathsf{c.c}' \mid \mathsf{c.c}' \in \mathsf{C_d}\,,\ \mathsf{c}' \neq \epsilon\ \}$$

By Lemma 4.1.4, for all input rooted chains $\mathsf{c.c}' \in \mathsf{C_d}$ then there exists a valid tree $t = (\sigma, l_t) \in \mathsf{d}$ and a location $l \in dom(t)$ such that $\mathsf{c}_l = \mathsf{c.c}'$. To conclude it suffices to fix $\mathsf{c}_{l_x} = \mathsf{c}$.

Case [`axis = descendant-or-self`] is analogous.

[`axis = parent`] By definition we have that

$$\mathbf{A}_{\mathsf{C_d}}(\mathsf{c.a}, \mathtt{parent}) = \{\ \mathsf{c}\ \}$$

By Lemma 4.1.4, for all input rooted chains $\mathsf{c} \in \mathsf{C_d}$ then there exists a valid tree $t = (\sigma, l_t) \in \mathsf{d}$ and a location $l \in dom(t)$ such that $\mathsf{c}_l = \mathsf{c}$. To conclude it suffices to fix $\mathsf{c}_{l_x} = \mathsf{c.a}$.

[`axis = ancestor`] By definition we have that

$$\mathbf{A}_{\mathsf{C_d}}(\mathsf{c}, \mathtt{ancestor}) = \{\ \mathsf{c}' \mid \mathsf{c}' \preceq \mathsf{c}\,,\ \mathsf{c} \neq \epsilon\ \}$$

By Lemma 4.1.4, for all input rooted chains $\mathsf{c.c}' \in \mathsf{C_d}$ then there exists a valid tree $t = (\sigma, l_t) \in \mathsf{d}$ and a location $l \in dom(t)$ such that $\mathsf{c}_l = \mathsf{c}$. To conclude it suffices to fix $\mathsf{c}_{l_x} = \mathsf{c.c}'$.

Case `ancestor-or-self` is analogous.

[axis = following-sibling] By definition we have that

$$\mathbf{A}_{\mathsf{C_d}}(\mathsf{c.a}, \texttt{following-sibling}) = \{\ \mathsf{c.b} \in \mathsf{C_d}\ \mid\ \mathsf{a} <_{d(\mathsf{c})} \mathsf{b}\ \}$$

By Lemma 4.1.4, for all input rooted chains $\mathsf{c} \in \mathsf{C_d}$ then there exists a valid tree $t = (\sigma, l_t) \in \mathsf{d}$ and a location $l_p \in dom(t)$ such that $\mathsf{c}_{l_p} = \mathsf{c}$, and such that the sequence of children of l_p contains two locations l, l' such that $\mathsf{type}(l) = \mathsf{a}$ and $\mathsf{type}(l') = \mathsf{b}$ and $\mathsf{a} <_{\mathsf{d}(\mathsf{c}_{l_p})} \mathsf{b}$.

Case preceding-sibling is analogous.

□

Putting everything together, we can prove the theorems of soundness and completeness for chain inference of XPath Step expressions.

Lemma 4.3.1 (Step Chain Soundness). Let d be a DTD, $t = (\sigma, l_t) \in \mathsf{d}$ be a valid tree, and $l \in dom(\sigma)$ be a location of the tree. If

$$\sigma, (\mathtt{x} := l) \vDash \mathtt{x/axis} :: \phi \ \Rightarrow\ \sigma, L$$

then for each location $l' \in L$ we have that

$$\mathsf{c}^{\sigma}_{l'} \in \mathbf{T}_{\mathsf{C}}(\,\mathbf{A}_{\mathsf{C}}(\mathsf{c}^{\sigma}_{l}, \texttt{axis}), \phi\,)$$

Proof. By Lemma 9.1.1 and Lemma 9.1.2. □

Lemma 4.3.1 (Step Chain Completeness). *Let* d *be a DTD and* $\mathsf{c} \in \mathsf{C_d}$ *a chain. If*

$$\mathsf{c}' \in \boldsymbol{T}_{\mathsf{C_d}}(\boldsymbol{A}_{\mathsf{C_d}}(\mathsf{c}, \texttt{axis}), \phi)$$

then there exists a valid tree $t = (\sigma, l_t) \in \mathsf{d}$ *and a location* $l \in dom(t)$ *such that* $\mathsf{c}_l = \mathsf{c}$ *and, assuming that* $\sigma, (\mathtt{x} := l) \vDash \mathtt{x/axis} :: \phi \ \Rightarrow\ \sigma, L$ *we have that* $\mathsf{c}^{\sigma}_{l'} = \mathsf{c}'$*, for some* $l' \in L$*.*

Proof. By Lemma 9.1.3 and Lemma 9.1.2. □

9.1.4 Soundness of Query Chain Inference

In this section we prove the correctness of chain inference for query expressions. We introduce first some auxiliary definitions and properties that we will use later in our formal development.

Auxiliary definitions

The notion of correctness we stated (Theorem 4.3.4) relies on tree projections (Section 4.3.1).

Let σ be a store and L_1, L_2 be two sequences of locations belonging to $dom(\sigma)$. The merge of L_1, L_2, denoted by $L_1 \uplus L_2$, is defined as the sequence L such that

- $l \in L_1 \cdot L_2$ iff $l \in L$;
- for all $l, l' \in L$ we have that $l \neq l'$;
- $(l_i, l_{i+1}) \in L$ implies $(l_i, l_{i+1}) \in docOrder_\sigma$.

Let σ be a store and σ_1, σ_2 be two of its projections. The stores obtained by only common and uncommon locations between σ_1 and σ_2 are defined as follows.

$$common(\sigma_1, \sigma_2) \stackrel{def}{=} \{\, l \leftarrow \mathsf{a}[\, L_1 \uplus L_2\,] \mid l \leftarrow \mathsf{a}[L_1] \in \sigma_1\,,\; l \leftarrow \mathsf{a}[L_2] \in \sigma_2 \,\}$$

$$uncommon(\sigma_1, \sigma_2) \stackrel{def}{=} \{\, l \leftarrow \mathsf{a}[L] \mid l \notin dom(\sigma_1) \cap dom(\sigma_2) \,\}$$

Definition 9.1.4 (Merge of Projections)**.** *Let σ be a store and σ_1, σ_2 be two of its projections. The merge of σ_1 and σ_2, denoted by $\sigma_1 \uplus \sigma_2$, is defined as*

$$\sigma_1 \uplus \sigma_2 \stackrel{def}{=} common(\sigma_1, \sigma_2) \cup uncommon(\sigma_1, \sigma_2)$$

Let $l \in dom(\sigma_1) \cap dom(\sigma_2)$ be a location, the merge of the trees $t_1 = (\sigma_1, l)$ and $t_2 = (\sigma_2, l)$, denoted by $t_1 \uplus t_2$, is defined as $t_1 \uplus t_2 \stackrel{def}{=} (\sigma_1 \uplus \sigma_2, l)$.

We outline some simple properties of the merge operator.

Proposition 9.1.5. *Given a tree t and two of its projections, $t_1, t_2 \preceq t$, the following properties hold.*

1. $t_1 \preceq t_1 \uplus t_2$
2. $t_1 \uplus t_2 \preceq t$
3. $t_1' \preceq t_1$ *and* $t_2' \preceq t_2$ *implies* $t_1' \uplus t_2' \preceq t_1 \uplus t_2$

Proof. 1) Follows from the definition of merge.
2) Follows from the definition of merge and from $t_i \preceq t$.
3) Follows since, by the point 1), we have that $t_1' \preceq t_1 \uplus t_2$ and $t_2' \preceq t_1 \uplus t_2$. Therefore $t_1' \uplus t_2' \preceq t_1 \uplus t_2$. □

We prove now an auxiliary lemma concerning return chains.

Proposition 9.1.6. *Let* d *be a DTD,* $t = (\sigma, l_t) \in \mathsf{d}$ *a valid tree,* q *a query,* γ *a dynamic environment and* Γ *a static environment such that* $\sigma \vDash_{\mathsf{d}} \gamma : \Gamma$*. Provided* $\sigma, \gamma \vDash \mathsf{q} \Rightarrow \sigma_{new}, L$ *and* $\Gamma \vdash_{\mathsf{C_d}} \mathsf{q} : (\mathsf{r}, \mathsf{v}, \mathsf{e})$*, if* $l \in L \cap dom(t)$ *then* $\mathsf{c}_l^{\sigma} \in \mathsf{r_q}$*.*

Proof. By induction on the structure of q.

Base.

[$\mathsf{q} = ()$] Immediate since $\mathsf{r_q} = \varnothing$.

[$\mathsf{q} =$ "`txt`"] Immediate since $l \notin dom(t)$.

Induction

[$\mathsf{q} = \mathsf{q}_1, \mathsf{q}_2$] By induction on $\mathsf{q}_1, \mathsf{q}_2$ and by noticing that $\mathsf{r_q} = \mathsf{r}_{\mathsf{q}_1} \cup \mathsf{r}_{\mathsf{q}_2}$.

[$\mathsf{q} =$ <a>q'] Immediate since $l \notin dom(t)$.

[$\mathsf{q} =$ `x/step`] By correctness of step chain inference (Theorem 4.3.1), and since resulting chains are in $\mathsf{r_q}$.

[$\mathsf{q} =$ `for x in` q_1 `return` q_2] First we notice that $L_{\mathsf{q}} = L_{\mathsf{q}_2}$. We conclude by inductive hypothesis on q_2 and because $\mathsf{r_q} = \mathsf{r}_{\mathsf{q}_2}$.

[$\mathsf{q} =$ `let x :=` q_1 `return` q_2] Analogous to the `for` case.

[$\mathsf{q} =$ `if` (q_0) `then` q_1 `else` q_2] First we notice that either $L_{\mathsf{q}} = L_{\mathsf{q}_1}$ or $L_{\mathsf{q}} = L_{\mathsf{q}_2}$. We conclude by inductive hypothesis on $\mathsf{q}_1, \mathsf{q}_2$ and because $\mathsf{r_q} = \mathsf{r}_{\mathsf{q}_1} \cup \mathsf{r}_{\mathsf{q}_2}$.

□

Theorem 4.3.4 ((Soundness of Query Chains)) *Let* d *be a DTD,* $t = (\sigma, l_t) \in \mathsf{d}$ *a valid tree,* q *a query,* γ *a dynamic environment ad* Γ *a static environment such that* $\sigma \vDash_{\mathsf{d}} \gamma : \Gamma$. *Provided* $\sigma, \gamma \vDash \mathsf{q} \Rightarrow \sigma_{new}, L$ *and* $\Gamma \vdash_{\mathsf{C_d}} \mathsf{q} : (\mathsf{r}, \mathsf{v}, \mathsf{e})$ *the following holds.*

1. *if* t^{min} *is a minimal* q*-projection of* t *then* $t^{min} \preceq t_{|\mathcal{L}^t_{\overline{\mathsf{r}} \cup \mathsf{v}}} \preceq t$
2. *if* $t^{new} = (\sigma_{new}, l_{new})$ *with* $l_{new} \in L \setminus dom(\sigma)$ *is a fresh subtree then* $dom(t^{new}) = \mathcal{L}^{t^{new}}_{\mathsf{e}}$

For the sake of conciseness we adopt the following abbreviations. The sequence L_{q} denotes the result q evaluation, that is $\sigma, \gamma \vDash \mathsf{q} \Rightarrow \sigma_{\mathsf{q}}, L_{\mathsf{q}}$. The sets of chains $\mathsf{r_q}, \mathsf{v_q}, \mathsf{e_q}$ denote the sets of return, used and element chains inferred for q, that is $\Gamma \vdash_{\mathsf{C_d}} \mathsf{q} : (\mathsf{r_q}, \mathsf{v_q}, \mathsf{e_q})$.

We prove the point 1.

Proof. By structural induction on q.

Base.

[$\mathsf{q} = (), \mathsf{q} =$ "$\mathtt{txt}$"] We conclude immediately since the q-projection of t is empty.

Induction.

[$\mathsf{q} = \mathsf{q}_1, \mathsf{q}_2$] A minimal q-projection t' is the merge of a minimal q_1-projection t'_1 with a minimal q_2-projection t'_2. This follows from the semantics rule (SQ-Concat). By inductive hypothesis we have that

$$t'_1 \preceq t_{|\overline{\mathsf{r}}_{\mathsf{q}_1} \cup \mathsf{v}_{\mathsf{q}_1}} \qquad \textit{and} \qquad t'_2 \preceq t_{|\overline{\mathsf{r}}_{\mathsf{q}_2} \cup \mathsf{v}_{\mathsf{q}_2}}$$

By Proposition 9.1.5

$$t' = t'_1 \uplus t'_2 \preceq t_{|\overline{\mathsf{r}}_{\mathsf{q}_1} \cup \mathsf{v}_{\mathsf{q}_1}} \uplus t_{|\overline{\mathsf{r}}_{\mathsf{q}_2} \cup \mathsf{v}_{\mathsf{q}_2}}$$

By chain inference rule (Concat) we have that $\mathsf{r_q} = \mathsf{r}_{\mathsf{q}_1} \cup \mathsf{r}_{\mathsf{q}_2}$ and $\mathsf{v_q} = \mathsf{v}_{\mathsf{q}_1} \cup \mathsf{v}_{\mathsf{q}_2}$. Therefore we conclude

$$t' \preceq t'_1 \uplus t'_2 \preceq t_{|\overline{\mathsf{r}}_{\mathsf{q}} \cup \mathsf{v}_{\mathsf{q}}}$$

[$\mathsf{q} =$ <a>q'] In this case, a tree projection is minimal for q if and only if it is minimal for q'. This follows from the fact that element construction always produces fresh locations that do not belong to the original store. This is specified by semantics rules (SQ-Elt) and (SQ-Copy). By inductive hypothesis, if t' is a minimal projection for q' then $t' \preceq t_{|\overline{\mathsf{r}}_{\mathsf{q}'} \cup \mathsf{v}_{\mathsf{q}'}}$. Following rule (Elt) we have that $\mathsf{r_q} = \varnothing$ and

$$\mathsf{v_q} = \overline{\mathsf{r}}_{\mathsf{q}'} \cup \mathsf{v}_{\mathsf{q}'}$$

We conclude since $t' \preceq t_{|\mathsf{v_q}}$.

[q = x/step] Semantics of step navigation is defined by rule (SQ-STEP). We assume that x is bound to a single location l_x in γ. The general case where x is bound to a sequence of locations goes by induction and concludes this case.

We distinguish between two main cases depending on the axis steps being forward or not.

(axis is forward) Since axis is forward, for all location $l \in L_{\mathtt{q}}$ we have that either l is a descendant of l_x, or $l = l_x$ if axis = self. Let t' be a minimal q-projection of t. Projection t' allows to correctly evaluate q, hence it is given by all locations in $L_{\mathtt{q}}$ together with their ancestors and their descendants. Formally, $t' = (\sigma_{|\mathcal{L}}, l_t)$ where

$$\mathcal{L} = \bigcup_{l \in L_{\mathtt{q}}} \{l\} \cup [\![\mathtt{ancestor}]\!]^{\sigma}_{l} \cup [\![\mathtt{descendant}]\!]^{\sigma}_{l}$$

Notice that t' is both minimal and unique. Notice also that it may be the case that $L_{\mathtt{q}} = ()$ and thus t' is the empty forest. Because $\sigma \vDash_{\mathtt{d}} \gamma : \Gamma$ we have that $\mathtt{c}_{l_x} \in \Gamma(\mathtt{x})$. By Theorem 4.3.1 step chain inference is correct. Therefore, following rule (STEPF) we can conclude that if $l \in L_{\mathtt{q}}$ then $\mathtt{c}_l \in \mathtt{r}_{\mathtt{q}}$. Notice also that $\mathtt{c}_l$ types any location l' that is an ancestors of l. The descendants of l are typed by the prolongation of $\mathtt{c}_l$, that belongs to $\overline{\mathtt{r}}_{\mathtt{q}}$. By rule (STEPF) no used chain is inferred. Therefore we conclude

$$t' \preceq t_{|\overline{\mathtt{r}}_{\mathtt{q}}}$$

(axis is backward or horizontal) In this case, since the axis is either backward or horizontal, all resulting locations $l \in L_{\mathtt{q}}$ are not descendants of l_x. Notice however that it may be the case that $l = l_x$ if axis = ancestor-or-self. Let t' be a minimal q-projection of t. Because t' allows to correctly evaluate q, it must contain: *i*) all locations in the result sequence $L_{\mathtt{q}}$, together with their respective ancestors and descendants and *ii*) the input location l_x together with its ancestors, but only in the case that $L_{\mathtt{q}} \neq ()$. We stress that we may not need to include the descendants of l_x in the projection, for instance if we are evaluating an horizontal axis. Formally, $t' = (\sigma_{|\mathcal{L}}, l_t)$ where $\mathcal{L}$ is

$$\bigcup_{l \in L_{\mathtt{q}}} \{l\} \cup [\![\mathtt{ancestor}]\!]^{\sigma}_{l} \cup [\![\mathtt{descendant}]\!]^{\sigma}_{l} \cup \{\, l_x \mid L_{\mathtt{q}} \neq () \,\}$$

Notice that for backward and horizontal axis we have $[\![\mathtt{ancestor}]\!]^{\sigma}_{l} \subseteq [\![\mathtt{ancestor}]\!]^{\sigma}_{l_x}$. Also, t' is unique. Because $\sigma \vDash_{\mathtt{d}} \gamma : \Gamma$ we have that $\mathtt{c}_{l_x} \in \Gamma(\mathtt{x})$. By Theorem 4.3.1 step chain inference is correct. Therefore, for all $l \in L_{\mathtt{q}}$ we have that $\mathtt{c}_l \in \mathtt{r}_{\mathtt{q}}$. Furthermore, if $L \neq ()$ then $\mathtt{c}_{l_x} \in \mathtt{v}_{\mathtt{q}}$. We conclude that

$$t' \preceq t_{|\overline{\mathtt{r}}_{\mathtt{q}} \cup \mathtt{v}_{\mathtt{q}}}$$

When $\gamma(x) = L$ query evaluation is performed by means of iteration, as defined by rules (SQ-Iter) and (SQ-IterBase). Let t' be the minimal $\mathtt{q}$-projection for $\gamma = \{\, l \in L \,\}$. By inductive hypothesis, chain inference is correct for each $l \in L$. Let t'_l be the minimal query $\mathtt{q}$-projection for $\gamma = \{\, l \,\}$. We conclude since by Lemma 9.1.5

$$t' \;=\; \biguplus_{l \in L} t'_l \;\preceq\; t_{\overline{\mathsf{r}}_\mathtt{q} \cup \mathsf{v}_\mathtt{q}}$$

$[\mathtt{q} = \mathtt{for\ x\ in\ q_1\ return\ q_2}]$ Semantics of iteration is defined by rule (SQ-For). Query $\mathtt{q}_2$ is evaluated once for each location $l \in L_{\mathtt{q}_1}$ binding $\mathtt{x}$ with l in γ. We denote by t_2^l a minimal $\mathtt{q}_2$-projection of t obtained by assuming that $\mathtt{x} \mapsto l \in \gamma$. We denote by $\mathcal{L}_1$ the set of result locations of $\mathtt{q}_1$ that are also productive for $\mathtt{q}_2$, that is $\mathcal{L}_1 = \{\, l \in L_1 \mid t_2^l \neq () \,\}$.

A minimal $\mathtt{q}$-projection of t is defined as $t' = t_1 \uplus t_2$ where t_1 is a minimal $\mathtt{q}_1$-projection of t that ensures only locations in $\mathcal{L}_1$ in the result sequence of $\mathtt{q}_1$, and t_2 is a minimal $\mathtt{q}_2$-projection of t obtained by the merge of all $\mathtt{q}_2$-minimal projections of t where $\mathtt{x}$ is bound to a location $l \in \mathcal{L}_1$.

We want to show that

$$t' \;=\; t_1 \uplus t_2 \;\preceq\; t_{|\overline{\mathsf{r}}_\mathtt{q} \cup \mathsf{v}_\mathtt{q}}$$

We show first $t_2 \preceq t_{|\overline{\mathsf{r}}_\mathtt{q} \cup \mathsf{v}_\mathtt{q}}$. By inductive hypothesis, provided that $l_i \in \mathcal{L}_1$ and $\Gamma[\mathtt{x} \mapsto \{\mathsf{c}_{l_i}\}] \vdash_\mathsf{C} \mathtt{q}_2 : (\mathsf{r}_i, \mathsf{v}_i, \mathsf{e}_i)$ it follows that

$$t_2^l \;\preceq\; t_{|\overline{\mathsf{r}_i} \cup \mathsf{v}_i}$$

By Lemma 9.1.5 we have also that

$$t_2 \;=\; \biguplus_{l \in \mathcal{L}_1} t_2^l$$

In general, if the result of $\mathtt{q}_2$ is non-empty then $\mathsf{r}_i \cup \mathsf{e}_i \neq \varnothing$.

By rule (For)

$$\mathsf{r}_\mathtt{q} = \bigcup_{\mathsf{c}_{l_i} \in \mathsf{r}_{\mathtt{q}_1}} \mathsf{r}_i \quad \text{and} \quad \mathsf{v}_\mathtt{q} \supseteq \bigcup_{\mathsf{r}_i \cup \mathsf{e}_i \neq \varnothing} \{\mathsf{v}_i\}$$

from which it follows

$$t_2 \;\preceq\; t_{|\overline{\mathsf{r}}_\mathtt{q} \cup \mathsf{v}_\mathtt{q}}$$

We show $t_1 \preceq t_{|\mathsf{v}_\mathtt{q}}$. By inductive hypothesis,

$$t_1 \preceq t_{|\overline{\mathsf{r}}_{\mathtt{q}_1} \cup \mathsf{v}_{\mathtt{q}_1}}$$

however since t' does not need the descendants of result locations of $\mathtt{q}_1$, we have also that $t_1 \preceq t_{|\mathsf{r}_{\mathtt{q}_1} \cup \mathsf{v}_{\mathtt{q}_1}}$. Then, because we need only result locations in $\mathcal{L}_1$ we have that $t_1 \preceq t_{|\mathsf{r}'_1 \cup \mathsf{v}_{\mathtt{q}_1}}$ where

$$\mathsf{r}'_1 \;=\; \bigcup_{\substack{\mathsf{c}_{l_i} \in \mathsf{r}_{\mathtt{q}_1} \\ \mathsf{r}_i \cup \mathsf{e}_i \neq \varnothing}} \{\, \mathsf{c}_{l_i} \,\}$$

By rule (FOR) we have that

$$\mathsf{r}_{\mathtt{q}_1} \cup \mathsf{v}_{\mathtt{q}_1} \subseteq \mathsf{v}_{\mathtt{q}}$$

and therefore we conclude

$$t_1 \preceq t_{|\mathsf{v}_{\mathtt{q}}}$$

Putting everything together, by Lemma 9.1.5

$$t_1 \uplus t_2 \preceq t_{|\overline{\mathsf{r}}_{\mathtt{q}} \cup \mathsf{v}_{\mathtt{q}}}$$

[$\mathtt{q}$ = `let x := q1 return q2`] The semantics of the let binding is defined by rule (SQ-LET). Let t_i be a minimal $\mathtt{q}_i$-projection for $\mathtt{q}_i$. Notice that in the evaluation of $\mathtt{q}_2$, `x` is bound to the resulting sequence of $\mathtt{q}_1$ in γ (no iteration is performed by the rule).

By inductive hypothesis

$$t_1 \preceq t_{|\overline{\mathsf{r}}_{\mathtt{q}_1} \cup \mathsf{v}_{\mathtt{q}_1}} \qquad \text{and} \qquad t_2 \preceq t_{|\overline{\mathsf{r}}_{\mathtt{q}_2} \cup \mathsf{v}_{\mathtt{q}_2}}$$

Following rule (LET) we have that

$$\mathsf{r}_{\mathtt{q}_1} \cup \mathsf{v}_{\mathtt{q}_1} \subseteq \mathsf{v}_{\mathtt{q}}$$

Therefore

$$t_1 \preceq t_{|\mathsf{v}_{\mathtt{q}}}$$

Furthermore, by rule (LET) we have that

$$\overline{\mathsf{r}}_{\mathtt{q}_2} \cup \mathsf{v}_{\mathtt{q}_2} \subseteq \overline{\mathsf{r}}_{\mathtt{q}} \cup \mathsf{v}_{\mathtt{q}}$$

Therefore

$$t_2 \preceq t_{|\overline{\mathsf{r}}_{\mathtt{q}} \cup \mathsf{v}_{\mathtt{q}}}$$

Now $t' = t_1 \uplus t_2$ is a minimal $\mathtt{q}$-projection of t. By Proposition 9.1.5 we conclude

$$t_1 \uplus t_2 \preceq t_{|\overline{\mathsf{r}}_{\mathtt{q}} \cup \mathsf{v}_{\mathtt{q}}}$$

[$\mathtt{q}$ = `if (q0) then q1 else q2`] The semantics of the conditional expression is defined by rules (SQ-IF). Let t', t_0, t_1 and t_2 minimal projection of t for $\mathtt{q}, \mathtt{q}_0, \mathtt{q}_1$ and $\mathtt{q}_2$, respectively. By inductive hypothesis

$$t_i \preceq t_{|\overline{\mathsf{r}}_{\mathtt{q}_i} \cup \mathsf{v}_{\mathtt{q}_i}} \qquad \text{for } i = 0..2$$

According to the semantics of the conditional expression

$$t' \preceq \biguplus_{i=0..2} t_i$$

and then we conclude by rule (IF) since

$$\biguplus_{i=0..2} t_i \preceq t_{|\overline{\mathsf{r}}_{\mathtt{q}} \cup \mathsf{v}_{\mathtt{q}}}$$

□

We prove the point 2.

Proof. By structural induction on q.

Base.

[$\mathsf{q} = ()$] The empty query does not construct any element.

[$\mathsf{q} =$ "$\mathtt{txt}$"] The query constructs a fresh text-node and by rule (TEXT) we have that $\mathsf{String} \in \mathsf{e}$.

Induction

[$\mathsf{q} = \mathsf{q}_1, \mathsf{q}_2$] The constructed elements returned by q are obtained by the concatenation of those returned by q_1 and q_2, respectively. By inductive hypothesis the statemet holds for $\mathsf{q}_1, \mathsf{q}_2$ and since $\mathsf{e}_\mathsf{q} = \mathsf{e}_{\mathsf{q}_1} \cup \mathsf{e}_{\mathsf{q}_2}$ we conclude.

[$\mathsf{q} =< a > \mathsf{q}' < /a >$] Following the rule for query semantics (SQ-CONSTR), the query q adds to the input store σ a new location, say l_a, such that $(l_\mathsf{a} \leftarrow \mathsf{a}[L^{\mathsf{copy}}_{\mathsf{q}'}])$, where $L^{\mathsf{copy}}_{\mathsf{q}'}$ denotes the copy of $L_{\mathsf{q}'}$. The resulting store is denoted by σ_q. We distinguish four cases depending on wether the subexpression q' returns *i*) only locations belonging to the input tree, *ii*) only constructed elements, *iii*) both constructed and input elements, or *iv*) the empty sequence.

i) We assume that $l \in L_{\mathsf{q}'}$ implies $l \in dom(t)$, where $t = (\sigma, l_t)$ denotes the input tree. By Proposition 9.1.6 we know that for all $l \in L_{\mathsf{q}'}$ we have $\mathsf{c}_l \in \mathsf{r}_{\mathsf{q}'}$. By rule (ELT), if $\mathsf{c}_l = \mathsf{c.b}$ and $\mathsf{c.b.c}' \in \overline{\mathsf{r}}_{\mathsf{q}'}$ then $\mathsf{a.b.c}' \in \mathsf{e}_\mathsf{q}$. Therefore let $t' = \sigma_\mathsf{q} @ l_\mathsf{a}$ we conclude

$$t' \preceq t'_{|\mathsf{e}_\mathsf{q}}$$

ii) We assume that $l \in L_{\mathsf{q}'}$ implies $l \notin dom(t)$, where $t = (\sigma, l_t)$ denotes the input tree. By inductive hypothesis, for all tree $t'' = \sigma_\mathsf{q} @ l_\mathsf{a}$, with $l \in L_{\mathsf{q}'}$, we have that

$$t'' \preceq t''_{|\mathsf{e}_{\mathsf{q}'}}$$

By rule (ELT), if $\mathsf{c} \in \mathsf{e}_{\mathsf{q}'}$ then $a.\mathsf{c} \in \mathsf{e}_\mathsf{q}$. Let $t' = \sigma_\mathsf{q} @ l_a$, we conclude that all locations of t' are typed by a prefix of a chain in e_q. Formally we have

$$t' \preceq t'_{|\mathsf{e}_\mathsf{q}}$$

iii) In the case where q' provides both input and fresh data, we conclude by reasoning as in *i*) and *ii*).

iv) We assume $L_{\mathsf{q}'} = ()$. In this case, the result of q is a tree composed by a single node labeled a. By inductive hypothesis we assume the system to be *correct* on q', but it may not be complete, thus we do not know if $\mathsf{r}_{\mathsf{q}'} \cup \mathsf{e}_{\mathsf{q}'} = \varnothing$ or not. So we distinguish between the two cases. If $\mathsf{r}_{\mathsf{q}'} \cup \mathsf{e}_{\mathsf{q}'} = \varnothing$ (hence the system is

also complete for $\mathtt{q}'$) we conclude since by rule (ELT) $\mathtt{a} \in \mathtt{e}_{\mathtt{q}}$. If $\mathtt{r}_{\mathtt{q}'} \cup \mathtt{e}_{\mathtt{q}'} \neq \varnothing$ (hence the system is correct but not complete for $\mathtt{q}'$) then rule (ELT) produces element chains as described in *i*) and *ii*). Since all these chains will start with label `a`, we can conclude that the resulting element of `q` is always typed.

[$\mathtt{q}$ = `x/step`] No element is constructed by the query.

[$\mathtt{q}$ = `for x in` $\mathtt{q}_1$ `return` $\mathtt{q}_2$] The elements constructed and returned by `q` are those constructed and returned by $\mathtt{q}_2$. We conclude by inductive hypothesis on $\mathtt{q}_2$.

[$\mathtt{q}$ = `let x :=` $\mathtt{q}_1$ `return` $\mathtt{q}_2$] Analogous to the `for` case.

[$\mathtt{q}$ = `if (`$\mathtt{q}_0$`) then` $\mathtt{q}_1$ `else` $\mathtt{q}_2$] The elements constructed and returned by `q` are those constructed and returned either by $\mathtt{q}_1$ or $\mathtt{q}_2$. We conclude by inductive hypothesis on $\mathtt{q}_1$ and $\mathtt{q}_2$.

□

Summing up, we proved that query chain inference is sound. This means two things. First, that locations in the input document accessed by the query are typed by some inferred chains. Second, that locations created and returned by the query are typed by some inferred chains. Completeness (whether inference yields to minimal sets of chains) is still open, and we leave it as a future work.

9.1.5 Soundness of Update chain Inference

In this section we prove that chain inference for update expressions is correct. As updates are defined in terms of queries, the proof relies on correctness of chain inference for query expressions (Theorem 4.3.4). Chain inference for updates is sound in the sense that all chains required to type locations involved by an update are inferred.

THEOREM 4.3.6 (SOUNDNESS OF UPDATE CHAIN INFERENCE) *Let* d *be a DTD,* t *a valid tree* $t = (\sigma, l_t) \in \mathsf{d}$*, and* $\mathtt{u}$ *an update with at most one free variable* $\mathtt{x}$*,* $\gamma = \{\, \mathtt{x} \mapsto l_t \,\}$ *a dynamic environment, and* $\Gamma = \{\, \mathtt{x} \mapsto \mathsf{s}_\mathsf{d} \,\}$ *a static environment. Provided that*

$$\sigma, \gamma \vDash \mathtt{u} \;\Rightarrow\; \sigma_\omega, \omega \qquad \sigma_\omega \vDash \omega \rightsquigarrow \sigma_\mathtt{u} \qquad \Gamma \vdash_{\mathsf{C}_\mathsf{d}} \mathtt{u} : \mathsf{U}$$

it holds that

- *if* $l \in dom(\sigma)$ *is a location in* t*, and the update* $\mathtt{u}$ *involves* l *then there exists* $\mathsf{c} : \mathsf{c}' \in \mathsf{U}$ *such that* $\mathsf{c}^\sigma_l = \mathsf{c} : \mathsf{c}'$*, where* $\mathsf{c}' \neq \epsilon$*.*
- *if* l *is a location in* $\mathtt{u}(t)$*, i.e.* $l \in dom(\sigma_\mathtt{u} @ l_t)$*, and the update* $\mathtt{u}$ *involves* l *then there exists* $\mathsf{c} : \mathsf{c}' \in \mathsf{U}$ *such that* $\mathsf{c}^{\sigma_\mathtt{u}}_l = \mathsf{c} : \mathsf{c}'$*, where* $\mathsf{c}' \neq \epsilon$*.*

Proof. By structural induction on $\mathtt{u}$.

Base.

[$\mathtt{u} = ()$] No location is involved by the empty update. As defined by rules (SU-EMPTY) and (SAU-EMPTY), $\mathtt{u}$ yields an empty update pending list, that performs no operation. Notice also that following rule (EMPTY) no chain is inferred.

[$\mathtt{u} = \mathtt{delete}\ \mathtt{q}_0$] Update $\mathtt{u}$ involves all locations in $L_{\mathtt{q}_0}$ (that is the resulting sequence of $\mathtt{q}_0$). As defined by rule (SU-DELETE), there should be exactly one location in $L_{\mathtt{q}_0}$. We assume $L_{\mathtt{q}_0} = (l_0)$, with $l_0 \in dom(t)$. By rule (SAU-DELETE) l_0 is detached from its parent. By rule (SANITYCHECK) this operation is correct since l_0 is mutable and thus can be updated. By Proposition 9.1.6 $\mathsf{c}^\sigma_{l_0} \in \mathsf{r}_{\mathtt{q}_0}$. The statement follows by definition since by following rule (DELETE) we have that given $\mathsf{c}^\sigma_{l_0} = \mathsf{c}.\mathsf{a}_0$ then $\mathsf{c}{:}\mathsf{a}_0 \in \mathsf{U}$.

[$\mathtt{u} = \mathtt{rename}\ \mathtt{q}_0\ \mathtt{as}\ \mathtt{b}$] Update $\mathtt{u}$ involves all locations in $L_{\mathtt{q}_0}$. As defined by rule (SU-RENAME) there is at most one location $l_0 \in dom(t)$ in the resulting sequence of $\mathtt{q}_0$. We assume $L_{\mathtt{q}_0} = (l_{\mathtt{q}_0})$. By rule (SAU-RENAME) the label of l_0 is renamed in label $\mathtt{b}$. This is correct since by rule (SANITYCHECK) l_0 is mutable, and thus can be updated, and the location is not the target of several `rename` expressions. By Proposition 9.1.6 $\mathsf{c}^\sigma_{l_0} \in \mathsf{r}_{\mathtt{q}_0}$. In this case, we have that the renamed location belongs both to the original and to the updated store. We conclude now that in both cases the location is typed. Let $\mathsf{c}^\sigma_{l_0} = \mathsf{c}.\mathsf{a}_0$ by rule (RENAME) we have $\mathsf{c}{:}\mathsf{a}_0 \in \mathsf{U}$. Let $\mathsf{c}^{\sigma_\mathtt{u}}_{l_0} = \mathsf{c}.b$ by rule (RENAME) we have that $\mathsf{c} : b \in \mathsf{U}$.

[u = `insert q (into|into as first|into as last) q0`] The locations involved by u are copies of source locations in L_{q}, together with their descendants. As defined by rules (SAU-InsertInto), (SAU-InsertFirst) and (SAU-InsertLast), these locations are in turn inserted as children of location L_{q_0}. By rule (SU-Insert1) there is at most one target location in the resulting sequence of q_0, hence $L_{\mathsf{q}_0} = (l_0)$. Source locations in L_{q} are typed by inferred chains, as stated by Theorem 4.3.4. We recall this fact, distinguishing between fresh and input source locations.

- If $l \in L_{\mathsf{q}}$ belongs to the input document then by Proposition 9.1.6 we have that $\mathsf{c}^{\sigma}_{l} \in \mathsf{r}_{\mathsf{q}}$. Furthermore, for all l' descendant of l we have $\mathsf{c}^{\sigma}_{l'} \in \overline{\mathsf{r}}_{\mathsf{q}}$.
- If $l \in L_{\mathsf{q}}$ is a fresh location, by Theorem 4.3.4, we have that l is typed by some chains in e_{q}, formally $\mathsf{c}^{\sigma_w}_{l}.\mathsf{e} \in \mathsf{e}_{\mathsf{q}}$. Furthermore, for all l' descendant of l we have $\mathsf{c}^{\sigma_w}_{l'}.\mathsf{e}' \in \mathsf{e}_{\mathsf{q}}$. Recall that σ_w is an intermediate store containing source locations that are arguments of elementary update commands, to be applied further.

Locations involved by u belongs to the updated store $\sigma_{\mathtt{u}}$ only. We conclude showing that chains typing these locations are in U. By Proposition 9.1.6 $\mathsf{c}^{\sigma}_{l_0} \in \mathsf{r}_{\mathsf{q}_0}$. By rule (Insert1) we have that

- if $\mathsf{c.a} \in \mathsf{r}_{\mathsf{q}}$ and $\mathsf{c.a.c}' \in \overline{\mathsf{r}}_{\mathsf{q}}$ then $\mathsf{c}^{\sigma}_{l_0} : \mathsf{a.c}' \in \mathsf{U}$
- if $\mathsf{e} \in \mathsf{e}_{\mathsf{q}}$ then $\mathsf{c}^{\sigma}_{l_0} : \mathsf{e} \in \mathsf{U}$

[u = `insert q (after|before) q0`] This case is similar to the previous one, except for the fact that chains for after/before expressions have a different definition. The locations involved by u are copies of source locations in L_{q}, together with their descendants. As defined by rules (SAU-InsertAfter), and (SAU-InsertBefore), these locations are in turn inserted as children of location L_{q_0}. By rule (SU-Insert2) there is at most one target location in the resulting sequence of q_0, hence $L_{\mathsf{q}_0} = (l_0)$. Source locations in L_{q} are typed as stated by Theorem 4.3.4. We recall this fact, distinguishing between fresh and input source locations.

- If $l \in L_{\mathsf{q}}$ belongs to the input document then By Proposition 9.1.6 we have that $\mathsf{c}^{\sigma}_{l} \in \mathsf{r}_{\mathsf{q}}$. Furthermore, for all l' descendant of l we have $\mathsf{c}^{\sigma}_{l'} \in \overline{\mathsf{r}}_{\mathsf{q}}$.
- If $l \in L_{\mathsf{q}}$ is a fresh location, by Theorem 4.3.4 we have that l is typed by some chains in e_{q}, formally $\mathsf{c}^{\sigma_w}_{l}.\mathsf{e} \in \mathsf{e}_{\mathsf{q}}$. Furthermore, for all l' descendant of l we have $\mathsf{c}^{\sigma_w}_{l'}.\mathsf{e}' \in \mathsf{e}_{\mathsf{q}}$. Recall that σ_w is an intermediate store containing source locations that are arguments of elementary update commands not yet applied.

Locations involved by u belongs to the updated store $\sigma_{\mathtt{u}}$. We conclude showing that chains for those locations are in U. By Proposition 9.1.6 $\mathsf{c}^{\sigma}_{l_0} \in \mathsf{r}_{\mathsf{q}_0}$. Now we can conclude since by rule (Insert1), if $\mathsf{c}^{\sigma}_{l_0} = \mathsf{c.a}$ we have that

- if $\mathsf{c.a} \in \mathsf{r}_{\mathsf{q}}$ and $\mathsf{c.a.c}' \in \overline{\mathsf{r}}_{\mathsf{q}}$ then $\mathsf{c} : \mathsf{c}' \in \mathsf{U}$

- if $\mathsf{e} \in \mathsf{e_q}$ then $\mathsf{c} : \mathsf{e} \in \mathsf{U}$

[$\mathtt{u}$ = `replace` q_0 `with` $\mathtt{q}$] In this case soundness follows as for delete and insert cases. The locations involved by $\mathtt{u}$ are copies of source locations in $L_{\mathtt{q}}$ together with their descendants, and target locations returned by $\mathtt{q}_0$. By rule (SU-Replace) there is only one target location l_0 in the resulting sequence of $\mathtt{q}_0$, hence $L_{\mathtt{q}_0}$=(l_0). We first treat the typing of source locations and the the typing of target locations. Source locations in $L_{\mathtt{q}}$ are typed as stated by Theorem 4.3.4. We recall this fact, distinguishing between fresh and input source locations.

- If $l \in L_{\mathtt{q}}$ belongs to the input document then By Proposition 9.1.6 we have that $\mathsf{c}_l^{\sigma} \in \mathsf{r_q}$. Furthermore, for all l' descendant of l we have $\mathsf{c}_{l'}^{\sigma} \in \overline{\mathsf{r}}_{\mathsf{q}}$.
- If $l \in L_{\mathtt{q}}$ is a fresh location, by Theorem 4.3.4 we have that l is typed by some chains in $\mathsf{e_q}$, formally $\mathsf{c}_l^{\sigma_w}.\mathsf{e} \in \mathsf{e_q}$. Furthermore, for all l' descendant of l we have $\mathsf{c}_{l'}^{\sigma_w}.\mathsf{e}' \in \mathsf{e_q}$. Recall that σ_w is an intermediate store containing source locations that are arguments of elementary update commands not yet applied.

By Proposition 9.1.6 $\mathsf{c}_{l_0}^{\sigma} \in \mathsf{r}_{\mathtt{q}_0}$. Now we can conclude that source locations are typed since by rule (Replace) let $\mathsf{c}_{l_0}^{\sigma} = \mathsf{c.a}$ we have that

- if $\mathsf{c.a} \in \mathsf{r_q}$ and $\mathsf{c.a.c}' \in \overline{\mathsf{r}}_{\mathsf{q}}$ then $\mathsf{c} : \mathsf{a.c}' \in \mathsf{U}$
- if $\mathsf{e} \in \mathsf{e_q}$ then $\mathsf{c} : \mathsf{e} \in \mathsf{U}$

Concerning target locations, the statement follows since let $\mathsf{c}_{l_0}^{\sigma} = \mathsf{c.a}$ we have that $\mathsf{c} : \mathsf{a} \in \mathsf{U}$.

Induction.

[$\mathtt{u} = \mathtt{u}_1, \mathtt{u}_2$] The locations involved by the concatenation of two updates is the union of the locations involved by each expression. We conclude by inductive hypothesis on $\mathtt{u}_1, \mathtt{u}_2$.

[$\mathtt{u}$ = `for x in q return` $\mathtt{u}_1$] The locations involved by the update are those involved by $\mathtt{u}_1$ when $\mathtt{x}$ is bound to each node in the resulting sequence of $\mathtt{q}$. By Proposition 9.1.6 the chain corresponding to all return location of $\mathtt{q}$ are inferred in $\mathsf{r_q}$. By inductive hypothesis the statement holds for the update $\mathtt{u}_1$ when $\mathtt{x}$ is bound to a return chain inferred for $\mathtt{q}$. This is described by rule (For), and therefore we conclude that the statement holds for $\mathtt{u}$.

[$\mathtt{u}$ = `let x :=  q return` $\mathtt{u}_1$] The locations involved by the update are those involved by $\mathtt{u}_1$ when applied with $\mathtt{x}$ bound to the resulting sequence of $\mathtt{q}$. By Proposition 9.1.6 the chain corresponding to all return location of $\mathtt{q}$ are inferred in $\mathsf{r_q}$. By inductive hypothesis the statement holds for the update $\mathtt{u}_1$ when $\mathtt{x}$ is bound to the resulting sequence of $\mathtt{q}$.

This is described by rule (LET), and therefore we conclude that the statement holds for $\mathtt{u}_1$.

[$\mathtt{u} = \mathtt{if\ (q)\ then\ u}_1\ \mathtt{else\ u}_2$] The chains involved by $\mathtt{u}$ are those involved by $\mathtt{u}_1$ or $\mathtt{u}_2$. By inductive hypothesis the statement holds for $\mathtt{u}_1, \mathtt{u}_2$ and we conclude that it holds also for $\mathtt{u}$.

□

In this section we proved the soundness of update chain inference. We showed that involved locations are always typed by some update chains, and that these update chains are correctly inferred. As for queries, completeness remains open and it is left as a future work. It is worth noticing however that completeness of chain inference for updates requires (as a necessary condition) chain inference for queries to be complete.

9.1.6 Soundness of Chain-based Independence

Proposition 9.1.7. *If* $\mathtt{q} \perp_{\mathsf{C_d}} \mathtt{u}$*, then we have:*

$$\mathcal{I}_{\mathsf{U}}^{t} \cap \mathcal{L}_{\overline{\mathsf{r}}\cup\mathsf{v}}^{t} \;=\; \mathcal{I}_{\mathsf{U}}^{\mathtt{u}(t)} \cap \mathcal{L}_{\overline{\mathsf{r}}\cup\mathsf{v}}^{\mathtt{u}(t)} \;=\; \varnothing$$

Proof. We treat separately parts of the statement concerning t and $\mathtt{u}(t)$.

We show $\mathcal{I}_{\mathsf{U}}^{t} \cap \mathcal{L}_{\overline{\mathsf{r}}\cup\mathsf{v}}^{t} = \varnothing$, aiming at obtaining a contradiction. Assume there is an involved location l_0 belonging to a minimal $\mathtt{q}$-projection, hence $l \in \mathcal{I}_{\mathsf{U}}^{t} \cap \mathcal{L}_{\overline{\mathsf{r}}\cup\mathsf{v}}^{t}$. We recall that the set of nodes of t typed by update chains in U is defined as

$$\mathcal{I}_{\mathsf{U}}^{t} \;=\; \{\, l \in dom(t) \mid \mathsf{c}_l = \mathsf{c} : \mathsf{c}' \in \mathsf{U} \quad \mathsf{c}' \neq \epsilon \,\}$$

and the set of nodes in a tree t typed by chains in $\overline{\mathsf{r}} \cup \mathsf{v}$ is defined as

$$\mathcal{L}_{\overline{\mathsf{r}}\cup\mathsf{v}}^{t} \;=\; \{\, l \in dom(t) \;\mid\; \mathsf{c}_{l_0}^{\sigma}.\mathsf{c} \in \overline{\mathsf{r}} \cup \mathsf{v} \,\}$$

From these definitions we conclude that $\mathsf{c}_{l_0}^{\sigma} \in \mathsf{U}$ and $\mathsf{c}_{l_0}^{\sigma} \in \overline{\mathsf{r}} \cup \mathsf{v}$. Hence,

$$\mathsf{U} \cap (\overline{\mathsf{r}} \cup \mathsf{v}) \neq \varnothing$$

By hypothesis $\mathtt{q} \perp_{\mathsf{C_d}} \mathtt{u}$ and then

$$\mathsf{confl}(\mathsf{r}, \mathsf{U}) = \mathsf{confl}(\mathsf{U}, \mathsf{r}) = \mathsf{confl}(\mathsf{U}, \mathsf{v}) = \varnothing$$

Assume $\mathsf{c}{:}\alpha.\mathsf{c}' \in \mathsf{U}$. Since $\mathsf{confl}(\mathsf{r}, \mathsf{U}) = \varnothing$ there cannot exists $\mathsf{c}_\mathsf{q} \in \mathsf{r}$ such that $\mathsf{c}_\mathsf{q} \preceq \mathsf{c}{:}\alpha.\mathsf{c}'$. Since $\mathsf{confl}(\mathsf{U}, \mathsf{r}) = \varnothing$ there cannot exists $\mathsf{c}_\mathsf{q} \in \mathsf{r}$ such that $\mathsf{c}.\alpha.\mathsf{c}' \preceq \mathsf{c}_\mathsf{q}$. Hence we conclude that $\mathsf{c}.\alpha.\mathsf{c}' \notin \overline{\mathsf{r}}$. Finally, since $\mathsf{confl}(\mathsf{U}, \mathsf{v}) = \varnothing$ there cannot exists $\mathsf{c}_\mathsf{q} \in \mathsf{v}$ such that $\mathsf{c} \preceq \mathsf{c}_\mathsf{q}$. Hence we conclude $\mathsf{c}.\alpha.\mathsf{c}' \notin \mathsf{v}$. Therefore from $\mathtt{q} \perp_{\mathsf{C_d}} \mathtt{u}$ follows

$$\mathsf{U} \cap (\overline{\mathsf{r}} \cup \mathsf{v}) = \varnothing$$

a contradiction.

The same reasoning holds for proving $\mathcal{I}_{\mathsf{U}}^{\mathtt{u}(t)} \cap \mathcal{L}_{\overline{\mathsf{r}}\cup\mathsf{v}}^{\mathtt{u}(t)} \;=\; \varnothing$.

□

Theorem 4.4.2 (Soundness of $\mathsf{C_d}$ independence) *Let* d *be a DTD,* $t = (\sigma, l_t) \in \mathsf{d}$ *a valid tree,* $\mathtt{q}$ *a query,* $\mathtt{u}$ *an update,* γ *a dynamic environment and* Γ *a static environment such that* $\sigma \vDash_{\mathsf{d}} \gamma : \Gamma$*. Then,*

$$\mathtt{q} \perp_{\mathsf{C_d}} \mathtt{u} \quad \textit{implies} \quad \mathtt{q} \between_{\mathsf{d}} \mathtt{u}$$

Proof. If $\mathtt{q} \perp_{\mathsf{C_d}} \mathtt{u}$ then by Proposition 9.1.7 we have that locations involved by the update are disjoint from location in a query projection. Since a query projection ensures preservation of query semantics under evaluation we can conclude $\mathtt{q} \between_{\mathsf{d}} \mathtt{u}$.

□

9.2 Soundness of the Chain Analysis for the Finite Case

In this section we prove that the independence analysis based on chains in possibly infinite case can be equivalently performed on a finite subset of k-chains. The main statement we prove is that a query and an update are conflicting if and only if they are conflicting on a finite set of k-chains. As a consequence of this, two expressions are independent if they do not conflict on a specific set of k-chains.

The proof is constituted of two main parts. We first prove that each chain inferred from an expression can be *folded* to a k-chain inferred for the same expression, yet preserving all informations concerning the behavior of the query. Then we show that this folding can be done also for a pair of conflicting expressions thus preserving all interactions between a query and update also in the finite analysis.

We develop our proof with the aid of *linear-path queries*, a subclass of queries defined in Chapter 2 that simplify the formal development yet allowing us to represent all query and update chains.

9.2.1 Folding Lemma for a Single Expression

In this section we prove Lemma 4.4.4 that states the existence of a folding relation between query chains and k-query-chains. We will reduce the problem to that of showing a folding relation for a subclass of expressions considered in this paper, called linear-path queries (lp queries).

Linear-path queries are defined in order to represent all navigational paths of a query that are superposed at the syntactic level, and are defined as follows.

Definition 9.2.1 (Linear-path queries)**.** *A linear-path query, denoted by q, is an expression matching the following grammar*

$$q \;::=\; \texttt{<}a\texttt{>}\; q \;\texttt{</}a\texttt{>} \mid \bar{q}$$

$$\bar{q} \;::=\; \texttt{for x in x/axis} :: \phi \;\texttt{return}\; \bar{q} \mid \texttt{x/axis} :: \phi \mid \text{“}\texttt{txt}\text{”}$$

Linear-path queries are the equivalent of navigational XPath expression without branching, plus both element and text node construction. These expressions are called linear because the grammar allows only for linear recursion. In fact, an iteration possibly nests only another iteration (on its right branch), and an element construction possibly nests either another element construction or an iteration. Also, element constructions are allowed only at the outermost level of the query.

A query $\mathtt{q}$ is always reformulated in a set of *lp* queries $\tau_{\mathtt{q}}$. To illustrate the process, consider the query $\mathtt{q}$ defined as

$$\begin{array}{l}\mathtt{for}\ \mathtt{x}_1 := \mathtt{x}//author, \mathtt{x}//editor\\ \mathtt{return}\\ \quad \mathtt{x}_1/\mathtt{self} :: author\end{array}$$

is translated into

$$\tau_{\mathtt{q}} = \left\{ \begin{array}{l}\mathtt{for}\ \mathtt{x}_1\ \mathtt{in}\ \mathtt{x}//author\\ \mathtt{return}\\ \quad \mathtt{x}_1/\mathtt{self} :: author\end{array} \quad , \quad \begin{array}{l}\mathtt{for}\ \mathtt{x}_1\ \mathtt{in}\ \mathtt{x}//editor\\ \mathtt{return}\\ \quad \mathtt{x}_1/\mathtt{self} :: author\end{array} \quad , \quad \mathtt{x}//author, \quad \mathtt{x}//editor \right\}$$

It is easy to see that this reformulation provides an over-approximation of all navigational paths embodied in $\mathtt{q}$, and hence all chains inferred for the original query are contained in the set of chains inferred for the reformulated ones. At the same time, we have that $k_{\mathtt{q}} = 2$ and for all $q \in \tau_{\mathtt{q}}$ it holds that $k_q \leq k_{\mathtt{q}}$, hence all chains inferred for a reformulated expression can be folded to $k_{\mathtt{q}}$-chains.

These two properties together makes sufficient to show a folding on the set of *lp*-queries, to scale on the whole language considered in this work. By showing a $k_{\mathtt{q}}$-folding for a superset of $\mathtt{q}$ chains, we immediately conclude the $k_{\mathtt{q}}$-folding for all $\mathtt{q}$ chains. This simplification is without loss of generality. Queries *lp* are a sort of normalized form for the problem of chain inference that simplify the formal development. In previous versions of this work, we developed proofs directly dealing with the general case. These resulted to be prolix, because they required involved properties, without clarifying the crux of the question. Folding for updates expressions will be given later, by means of linear-path queries.

Reformulating a query in a set of *lp*-queries

The set of linear-paths induced by a query is defined in Table 9.1. We describe the reformulation process. Given a query $\mathtt{q}$, the set of *lp* queries induced by $\mathtt{q}$ is obtained by splitting paths in concatenations (",") and conditional expressions, by approximating let-bindings, by linearizing all iterations of the query and, finally, by rewriting element constructions.

Let-bindings are approximated by means of iteration. By doing this in principle we obtain a correct multiplicity values for the expression, however there is a gap to fill in terms of inferred chains. In fact, iterations discard chains bound to variables if they do not satisfy navigational specifications, while rule (Let) retains all chains bound to a variable $\mathtt{x}$ no regardless. For this reason, we put the iteration in union with all expression bound with a new let-variable. These are return chains instead of used ones, but the nature of the chains is irrelevant for the end of showing a folding relation.

To illustrate path splitting and let-binding approximation, consider the following example.

$$\mathtt{LPQ}\left(\begin{array}{l}\mathtt{let}\ \mathtt{x}_1 := \mathtt{x}//author, \mathtt{x}//editor\\ \mathtt{return}\\ \ \mathtt{x}_1/\mathtt{self} :: author\end{array}\right) =$$

$$\left\{\begin{array}{l}\mathtt{for}\ \mathtt{x}_1\ \mathtt{in}\ \mathtt{x}//author\\ \mathtt{return}\\ \ \mathtt{x}_1/\mathtt{self} :: author\end{array}\right., \left.\begin{array}{l}\mathtt{for}\ \mathtt{x}_1\ \mathtt{in}\ \mathtt{x}//editor\\ \mathtt{return}\\ \ \mathtt{x}_1/\mathtt{self} :: author\end{array}\right., \mathtt{x}//author, \mathtt{x}//editor\Bigg\}$$

Here the chains pointing to *editor* elements are discarded by the iteration. This is correct from the semantics point of view but it is not compliant with the (LET) rule that also produces chains pointing to editor as used chains.

The functions $R^{\bullet}_{\mathtt{for}}$ and $R^{\bullet}_{e}$ are introduced for the linearization of the iterations and for lifting element constructions.

We describe now the linearization of iterations. Given a `for` construct, we denote by $R_{\mathtt{for}}$ the function that linearizes expressions of the form `for x in` $\mathtt{q}_1$ `return` $\mathtt{q}_2$. $R_{\mathtt{for}}$ is defined as the identity on any input query, except for the ones matching the following case.

$$R_{\mathtt{for}}\left(\begin{array}{l}\mathtt{for}\ \mathtt{x}_1\ \mathtt{in}\\ \ (\mathtt{for}\ \mathtt{x}_2\ \mathtt{in}\ \mathtt{q}_2\ \mathtt{return}\ \mathtt{q}_1)\\ \mathtt{return}\ \mathtt{q}_0\end{array}\right) \stackrel{def}{=} \begin{array}{l}\mathtt{for}\ \mathtt{x}_2\ \mathtt{in}\ R_{\mathtt{for}}(\mathtt{q}_2)\\ \ \mathtt{for}\ \mathtt{x}_1\ \mathtt{in}\ R_{\mathtt{for}}(\mathtt{q}_1)\\ \ \mathtt{return}\ R_{\mathtt{for}}(\mathtt{q}_0)\end{array}$$

Notice that the rule applies locally on a query `q` and it has to be iterated until fixpoint for linearizing all subexpressions, until $R_{\mathtt{for}}(\mathtt{q}) = \mathtt{q}$. We denote this by $R^{\bullet}_{\mathtt{for}}(\mathtt{q})$. Also, we denote by $R^{\bullet}_{\mathtt{for}}(\mathtt{Q})$ the rewriting of a set of queries `Q`.

The fixpoint rewriting works as in the following example.

$$\begin{array}{l}\mathtt{for}\ \mathtt{x}_1\ \mathtt{in}\\ \ (\mathtt{for}\ \mathtt{x}_2\ \mathtt{in}\\ \ \ \mathtt{for}\ \mathtt{x}_3\ \mathtt{in}\ \mathtt{x}//book\\ \ \ \mathtt{return}\ \mathtt{x}_3//author\\ \ \mathtt{return}\ \mathtt{x}_2//name)\\ \mathtt{return}\ \mathtt{x}_1//first\end{array} \xrightarrow{R^{\bullet}_{\mathtt{for}}} \begin{array}{l}\mathtt{for}\ \mathtt{x}_2\ \mathtt{in}\\ \ (\mathtt{for}\ \mathtt{x}_3\ \mathtt{in}\ \mathtt{x}//book\\ \ \ \mathtt{return}\ \mathtt{x}_3//author)\\ \mathtt{return}\\ \ \mathtt{for}\ \mathtt{x}_1\ \mathtt{in}\ \mathtt{x}_2//name\\ \ \ \mathtt{return}\ \mathtt{x}_1//first\end{array}$$

$$\xrightarrow{R^{\bullet}_{\mathtt{for}}} \begin{array}{l}\mathtt{for}\ \mathtt{x}_3\ \mathtt{in}\ \mathtt{x}//book\\ \ \mathtt{for}\ \mathtt{x}_2\ \mathtt{in}\ \mathtt{x}_3//author\\ \ \ \mathtt{for}\ \mathtt{x}_1\ \mathtt{in}\ \mathtt{x}_2//name\\ \ \ \ \mathtt{return}\ \mathtt{x}_1//first\end{array}$$

$$
\begin{array}{rcl}
\texttt{LPQ}(()) \stackrel{def}{=} \varnothing & \qquad & \texttt{LPQ}(\text{“}\texttt{txt}\text{”}) \stackrel{def}{=} \{\ \text{“}\texttt{txt}\text{”}\ \} \\
\texttt{LPQ}(\texttt{q}_1, \texttt{q}_2) \stackrel{def}{=} \texttt{LPQ}(\texttt{q}_1) \cup \texttt{LPQ}(\texttt{q}_2) & & \\
\texttt{LPQ}(\texttt{x/step}) \stackrel{def}{=} \{\texttt{x/step}\} & & \\
\texttt{LPQ}(\texttt{q}) \stackrel{def}{=} R_e^{\bullet}(\texttt{LPQ}(\texttt{q})) & & \\
\texttt{LPQ}(\texttt{for x in q}_1\ \texttt{return q}_2) \stackrel{def}{=} \bigcup_{q_i \in \texttt{LPQ}(\texttt{q}_i)} R^{\bullet}_{\texttt{for}}(\texttt{for x in } q_1\ \texttt{return } q_2) & & \\
\texttt{LPQ}(\texttt{let x := q}_1\ \texttt{return q}_2) \stackrel{def}{=} \texttt{LPQ}(\texttt{for x in q}_1\ \texttt{return q}_2) \cup \texttt{LPQ}(\texttt{q}_1) & & \\
\texttt{LPQ}(\texttt{if (q}_0\texttt{) then q}_1\ \texttt{else q}_2) \stackrel{def}{=} \bigcup_i \texttt{LPQ}(\texttt{q}_i) & & \\
\texttt{LPQ}(\texttt{<}a\texttt{>q</}a\texttt{>}) \stackrel{def}{=} \bigcup_{q \in \texttt{LPQ(q)}} \texttt{<}a\texttt{>}q\texttt{</}a\texttt{>} & &
\end{array}
$$

Table 9.1: Linear-path query extraction

For *lp* queries, element construction is allowed only at the outermost level of the query. [1] Expressions that do not satisfy this property are rewritten in proper ones by lifting element construction. For example, the query

$$\texttt{for x in y//}b\ \texttt{return <}new\texttt{>x</}new\texttt{>}$$

is rewritten in

$$\texttt{<}new\texttt{>for x in y//}b\ \texttt{return x</}new\texttt{>}$$

First notice that the semantics of the two expressions is different: the former returns a forest of n trees, where n is the number of b elements in a tree, while the latter returns a single tree. Also, that there may be no type definition in the schema employing b as a label. In this case the chain inference process outputs the empty set for the former expression, while it outputs the element chain *new* for the latter. However, while the semantics of the expressions may differ, the rewriting is still sound for the purpose of chain inference. In this case, the set of used, return and element chains inferred for the original and the rewritten expression coincide. In general, it is easy to show that the set of *element* chains inferred for a *lp* query set a sound approximation of the element chains of the original query.

We denote by $R_e(\texttt{q})$ the function that rewrites a query by lifting element constructions. $R_e(\texttt{q})$ is defined as the identity for all input query, except for the ones matching

[1] It is worth recalling that we assume also that query do not construct elements in the left branch of an iteration or let binding.

one of the following two cases.

$$R_e(\texttt{for x in q}_1 \texttt{ return <}a\texttt{>q}_2\texttt{</}a\texttt{>}) \stackrel{def}{=} \texttt{<}a\texttt{>for x in q}_1 \texttt{ return } R_e(\texttt{q}_2)\texttt{</}a\texttt{>}$$

$$R_e(\texttt{<}a\texttt{>q</}a\texttt{>}) \stackrel{def}{=} \texttt{<}a\texttt{>}R_e(\texttt{q})\texttt{</}a\texttt{>}$$

The function is intentionally defined to work only on that specific form of iteration. It is meant to be applied over queries that are (almost) *lp* queries, up to element construction. Notice that the rules have to applied until fixpoint is reached, i.e. $R_e(\texttt{q}) = \texttt{q}$. We denote this by $R_e^\bullet(\texttt{q})$. We will use the same notation also when speaking about set of queries. The fixpoint is unique and always reached by the translation.

As an example consider the previous query

$$\texttt{for x in y//}b\texttt{ return <}new\texttt{>q</}new\texttt{>}$$

where $\texttt{q} = \texttt{for x}_2\texttt{ in y//}a\texttt{ return <}sub\texttt{/>}$. We have that

$$\begin{aligned}
& R_e^\bullet(\texttt{for x in y//}b\texttt{ return <}new\texttt{>q</}new\texttt{>}) \\
= \; & R_e^\bullet(\texttt{<}new\texttt{>for x in y//}b\texttt{ return } R_e(\texttt{q})\texttt{</}new\texttt{>}) \\
= \; & R_e^\bullet(\texttt{<}new\texttt{>for x in y//}b\texttt{ return} \\
& \quad \texttt{<}sub\texttt{>} \\
& \quad \texttt{for x}_2\texttt{ in y//}a\texttt{ return ()} \\
& \quad \texttt{</}sub\texttt{>} \\
& \texttt{</}new\texttt{>}) \\
= \; & \texttt{<}new\texttt{>} \\
& \texttt{<}sub\texttt{>} \\
& \texttt{for x in y//}b\texttt{ return} \\
& \texttt{ for x}_2\texttt{ in y//}a\texttt{ return ()} \\
& \texttt{</}sub\texttt{>} \\
& \texttt{</}new\texttt{>}
\end{aligned}$$

This translation is reminiscent of path extraction, however there are two difference. The first is that in the definition of the set of *lp* queries associated to an expression, we do not take into account variable binding. We avoid this process because it is not needed in order to have a sound k-folding. The second difference, and most important, is that this process is described formally, but actually never implemented. It is just used to show that the k-analysis is sound in principle.

Correctness wrt chains of the `LPQ()` transformation of queries is stated as follows.

Proposition 9.2.2. *Let* $\mathtt{d}$ *be a DTD,* $\mathtt{q}$ *a query, and* Γ *a static environment. Let us denote by* $\tau_{\mathtt{q}}$ *the set of used, return and element chains inferred for* $\mathtt{q}$ *wrt* Γ *and by* $\tau_{\mathtt{LPQ(q)}}$ *the union of used, return, and element chains inferred for each expression in* $\mathtt{LPQ(q)}$*. Then,* $\tau_{\mathtt{q}} \subseteq \tau_{\mathtt{LPQ(q)}}$*.*

Proof. By induction on the structure of $\mathtt{q}$. □

By over approximating the set of chains inferred for an expression, we have that also the result of a query is always over approximated by its associated set of linear-path queries. However, what is important for us is not only the resulting locations of a query, but also the whole set of locations accessed. This can be showed as a corollary of Proposition 9.2.2.

Computing the Multiplicity Value for *lp*-queries

In the general case (Section 2.3), the definition of the k value is (necessarily) structured, because several navigational paths are superposed at the syntactic level. Differently, linear-path allow us to directly compute the multiplicity value for a given query expression defined in Chapter 4, without inductive definition. This will be useful in the proof in order to show that the finite on k-chains is equivalent to the possibly infinite analysis.

Proposition 9.2.3. *Let* q *be an lp query without element construction then*

$$k_q = \max_{a \in \Sigma} \left\{ \sum_{\mathtt{step} \in q} \mathcal{F}(a, \mathtt{step}) + \mathcal{R}(\mathtt{step}) \right\}$$

Furthermore, in the case that q *is nested inside an element construction, we have*

$$k_{<a>q} = \begin{cases} 1 + k_q & \text{if } a \text{ is the most frequent label in } q \\ k_q & \text{otherwise} \end{cases}$$

Where a *is the most frequent label in* $\mathtt{q}$ *if, for all* $b \in \Sigma_{\mathtt{S}}$*, if we have that*

$$\sum_{\mathtt{step} \in q} \mathcal{F}(b, \mathtt{step}) \leq \sum_{\mathtt{step} \in q} \mathcal{F}(a, \mathtt{step})$$

Proof. Straightforward by induction on q. □

The following lemma shows that all *lp*-queries inferred for an expression have a multiplicity value bounded by that of the original query.

Proposition 9.2.4. *Let* $\mathtt{d}$ *be a DTD,* $\mathtt{q}$ *a query, and* Γ *a static environment. Let us denote by* $\tau_{\mathtt{q}}$ *and* $\tau_{\mathtt{LPQ(q)}}$ *the union of the set of used, return, and element chains inferred for* $\mathtt{q}$ *wrt* Γ*, and for each expression in* $\mathtt{LPQ(q)}$*, respectively. Then, for all query* $q \in \mathtt{LPQ(q)}$ *we have that* $k_q \leq k_{\mathtt{q}}$*.*

Proof. By induction on the structure of $\mathtt{q}$. □

The existence of a folding relation between query chains and k query chains is stated by the following theorem.

Lemma 9.2.5 (Folding of linear-path queries)**.** *Let* d *be a DTD,* $\mathtt{q}$ *a query with at most only one free variable* $\mathtt{x}$*, and* $\Gamma = \{\ \mathtt{x} \mapsto \mathsf{s_d}\ \}$ *a static environment. For all* $q \in \mathtt{LPQ(q)}$*, provided that* $\Gamma \vdash_{\mathsf{C_d}} q : (\mathsf{r}, \mathsf{v}, \mathsf{e})$ *and* $\tau = \mathsf{r} \cup \mathsf{v} \cup \mathsf{e}$*, for each chain* $\mathsf{c} \in \tau$ *there exists a chain* $\mathsf{c}' \in \tau$ *such that* $\mathsf{c} \hookrightarrow^*_{\mathsf{d}} \mathsf{c}'$ *and* c' *is a* $k_{\mathtt{q}}$*-chain.*

Thanks to the above lemma, we can finally conclude that the folding relation exists for chains inferred for all queries.

Lemma 4.4.4 (Folding) *Let* d *be a DTD,* $\mathtt{q}$ *a query with at most only one free variable* $\mathtt{x}$*, and* $\Gamma = \{\ \mathtt{x} \mapsto \mathsf{s_d}\ \}$ *a static environment. Assume that* $\Gamma \vdash_{\mathsf{C_d}} \mathtt{q} : (\mathsf{r}, \mathsf{v}, \mathsf{e})$ *and* $\tau = \mathsf{r} \cup \mathsf{v} \cup \mathsf{e}$*. For each chain* $\mathsf{c} \in \tau$ *there exists a chain* $\mathsf{c}' \in \tau$ *such that* $\mathsf{c} \hookrightarrow^*_{\mathsf{d}} \mathsf{c}'$ *and* c' *is a* $k_{\mathtt{q}}$*-chain.*

Proof. Let $\mathsf{c} \in \tau_{\mathtt{q}}$ be a chain. By Lemma 9.2.2 we have that $\mathsf{c} \in \tau_{\mathtt{LPQ(q)}}$. We conclude since by Lemma 9.2.5 c can be folded to a $k_{\mathtt{q}}$-chains. □

In the the remainder of this section we prove Lemma 9.2.5.

Folding Lemma for *lp*-queries

The height of an *lp*-query q, denoted by $h(q)$, is defined as

$$\begin{aligned} h(\,()\,) &= 0 \\ h(\text{“}\mathtt{txt}\text{”}) &= 0 \\ h(\mathtt{x/axis::}\phi) &= 0 \\ h(\mathtt{for\ x\ in}\ q_1\ \mathtt{return}\ q_2) &= 1 + \max\{h(q_1), h(q_2)\} \\ h(\texttt{<}a\texttt{>}q\texttt{</}a\texttt{>}) &= 1 + h(q) \end{aligned}$$

The following lemma states that any chain can be folded to a 1-chain.

Lemma 9.2.6. *Let* d *be a DTD. For all chain* $\mathsf{c} \in \mathsf{C_d}$ *there exists a 1-chain* $\mathsf{c}' \in \mathsf{C_d}$ *such that* $\mathsf{c} \hookrightarrow^*_{\mathsf{d}} \mathsf{c}'$*.*

Proof. Given $\mathsf{c} \in \mathsf{C_d}$, if c is a 1-chain then we are done. Otherwise c is of the form $\mathsf{c} = \mathsf{c}_1.w.\mathsf{c}_2$, where w is a cycle $w = \mathsf{a}.\mathsf{a}_1 \ldots \mathsf{a}_n.\mathsf{a}$. By definition of the folding relation we have that

$$\mathsf{c}_1.w.\mathsf{c}_2 \ \hookrightarrow_{\mathsf{d}}\ \mathsf{c}_1.\mathsf{a}.\mathsf{c}_2$$

Now if $\mathsf{c}_1.\mathsf{a}.\mathsf{c}_2$ is a 1-chain then we are done, otherwise we repeat the folding process until we obtain a chain c' with no cycles, such that $\mathsf{c} \hookrightarrow^*_{\mathsf{d}} \mathsf{c}'$. This process terminates because the number of symbols in c is finite. □

Lemma 9.2.5 (Folding of linear-path queries) *Let* d *be a DTD,* $\mathtt{q}$ *a query with at most only one free variable* $\mathtt{x}$*, and* $\Gamma = \{\ \mathtt{x} \mapsto \mathsf{s_d}\ \}$ *a static environment. For all* $q \in \mathtt{LPQ(q)}$*, provided that* $\Gamma \vdash_{\mathsf{C_d}} q : (\mathsf{r}, \mathsf{v}, \mathsf{e})$ *and* $\tau = \mathsf{r} \cup \mathsf{v} \cup \mathsf{e}$*, for each chain* $\mathsf{c} \in \tau$ *there exists a chain* $\mathsf{c}' \in \tau$ *such that* $\mathsf{c} \hookrightarrow^*_{\mathsf{d}} \mathsf{c}'$ *and* c' *is a* $k_{\mathtt{q}}$*-chain.*

Proof. By induction on $h(q)$.

Base ($h = 0$) In this case q has one of the following three forms

$$() \quad | \quad \text{“}\mathtt{txt}\text{”} \quad | \quad \mathtt{x/axis :: }\phi$$

We conclude the proof by a case analysis.

[$q = ()$] Straightforward since $\tau_q = \varnothing$.

[$q =$ “txt”] Recall that because the query “txt” is independent of any update, in Chapter 2 we assumed that “txt” is nested into a for expression. Furthermore, because we do not navigate on constructed sequences, we have that “txt” should be in the right branch of the for expression. Then, in order to do chain inference for “txt” we must have a navigational step in the left branch of q that makes $k_q > 0$. In the case where $k_q = 0$ it means that no operation is performed by the left branch of the query and thus chains for “txt” are not inferred by the type system rules. We conclude since $\tau_q = \{\mathsf{String}\}$ and String is a 1-chain.

[$q = \mathtt{x/axis :: }\phi$] Because $\mathtt{x} \mapsto \mathsf{s_d}$ we have that $\tau_q \neq \varnothing$ only if axis is one of self, child, descendant, or descendant−or−self. We assume $\phi = \mathtt{a} \in \Sigma$ and we proceed with a case analysis on axis. The proof for the case $\phi \in \{\mathtt{node()}, \mathtt{text()}\}$ is analogous, since node() can be simulated by ranging over all labels in Σ, and text() by considering only type String.

[axis = child] In this case $k_q = 1$ and we conclude since either $\tau_q = \varnothing$ or $\tau_q = \{\ \mathsf{s_d.a}\ \}$, where $\mathsf{s_d} \neq \mathtt{a}$ by hypothesis of non-recursive schema root (see Chapter 2).

[axis = self] In this case $k_q = 1$ and we conclude since either $\tau_q = \varnothing$ or $\tau_q = \{\ \mathsf{s_d}\ \}$.

[axis = descendant] In this case $k_q = 1$ since $\mathcal{R}(\mathtt{descendant :: a}) = 1$. We conclude since by Lemma 9.2.6 any chain can be folded to a 1−chain.

[axis = descendant−or−self] Analogous to the self and descendant cases.

Induction ($h > 0$)

In this In this case q has one of two forms.

$$\texttt{<a>}\, q' \,\texttt{} \quad | \quad \mathtt{for\ x\ in}\ q_1\ \mathtt{return}\ q_2$$

[$q =$ <a>q'] We prove this case by making a case analysis wrt the set of used, return and element chains inferred for q.

used chains By rule (ELT), $\mathsf{v}_q = \overline{\mathsf{r}}_{q'} \cup \mathsf{v}_{q'}$. All chains inferred for the inner query q' are obtained by a derivation tree of height $h(q') < h(q)$. Then by inductive hypothesis on q' we conclude the thesis for all chains in $\mathsf{r}_{q'} \cup \mathsf{v}_{q'}$, since they are $k_{q'}$-chains and provided that $k_{q'} \leq k_q$ they are k_q-chains.

return chains By rule (ELT), $\mathsf{r}_q = \varnothing$ hence no return chain is inferred.

element chains We have two main cases to consider.

- If $\mathsf{r}_{q'} \cup \mathsf{v}_{q'} = \varnothing$ then $\mathsf{e}_q = \{a\}$. We conclude since $k_q = 1$.
- If $\mathsf{r}_{q'} \cup \mathsf{v}_{q'} \neq \varnothing$ then $\mathsf{e}_q = \mathsf{e}^e_q \cup \mathsf{e}^r_q$ where

$$\mathsf{e}^e_q = \{\ a.\mathsf{c} \mid \mathsf{c} \in \mathsf{e}_{q'}\ \} \qquad \mathsf{e}^r_q = \{\ a.\mathsf{a}.\mathsf{c}' \mid \mathsf{c}.\mathsf{a} \in \mathsf{r}_{q'}, \mathsf{c}.\mathsf{a}.\mathsf{c}' \in \overline{\mathsf{r}}_{q'}\ \}$$

  The two sets are built from element and return chains of q', respectively. Let us consider them separately.

  - If $a.\mathsf{c}$ belongs to e^e_q then $\mathsf{c} \in \mathsf{e}_{q'}$. By inductive hypothesis c is a $(k_{q'})$-chain. If $k_q = 1 + k_{q'}$ then a is the most frequent label in c with frequency upper bounded by $k_{q'}$. Therefore $a.\mathsf{c}$ is a $(1 + k_{q'})$-chain and hence a k_q-chain. In the case that $k_q = k_{q'}$ then a is not the most frequent label in c. Thus the frequency of a in c is strictly upper bounded by $k_{q'}$. Therefore $a.\mathsf{c}$ is still a $k_{q'}$-chain and thus a k_q-chain.
  - If $a.\mathsf{a}.\mathsf{c}'$ belongs to e^r_q then $\mathsf{c}.\mathsf{a} \in \mathsf{r}_{q'}$ and $\mathsf{c}.\mathsf{a}.\mathsf{c}' \in \overline{\mathsf{r}}_{q'}$. By inductive hypothesis $\mathsf{c}.\mathsf{a}.\mathsf{c}'$ can be folded to a $(1 + k_{q'})$-chain $\mathsf{c}_1.\mathsf{a}.\mathsf{c}'_1$. Thus $a.\mathsf{a}.\mathsf{c}'$ can be folded to $a.\mathsf{a}.\mathsf{c}'_1$. Provided that $\mathsf{a}.\mathsf{c}'_1$ is a $(1 + k_{q'})$-chain, we have that $a.\mathsf{a}.\mathsf{c}'$ is again a $(1 + k_{q'})$-chain or the prefix $\mathsf{a}.\mathsf{c}'$ can be folded to a $(k_{q'})$-chain, thus obtaining a $(1 + k_{q'})$-chain.

[$q =$ `for x in` q_1 `return` q_2] Here $\mathtt{q}_2$ possibly nests an arbitrary number n of iterations. In particular, we assume that the rightmost subexpression of q is a navigational step. It may also be that the rightmost subexpression is a text-element construction, but in that case the proof is a particular instance of the one given below where one does not make any use of the static variable environment. Any derivation tree of chain inference for q must end by applying the (FOR) rule, i.e., it looks like this (for simplicity we do not detail all rule operations)

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash_{\mathsf{C}} \texttt{step}_1 \,:\, \tau_1}{\texttt{x}_1 \mapsto \tau_1, \Gamma \vdash_{\mathsf{C}} \texttt{step}_2 \,:\, \tau_2}}{\vdots} \qquad \{\texttt{x}_i \mapsto \tau_i\}_{i \leq n}, \Gamma \vdash_{\mathsf{C}} \texttt{step}_{n+1} \,:\, \tau_{n+1}}{\Gamma \vdash_{\mathsf{C}} \texttt{for}\ \texttt{x}_1\ \texttt{in}\ \texttt{step}_1 \ldots \texttt{x}_n\ \texttt{in}\ \texttt{step}_n\ \texttt{return}\ \texttt{step}_{n+1} \,:\, \tau_q}$$

where $\texttt{step}_i$ is of the form $\texttt{x}_i/\texttt{axis}_i :: \phi_i$, with $\texttt{x}_i$ either a free-variable of q bound to the root-type in Γ, or a bound-variable of q previously defined in $\{\texttt{x}_1, \ldots, \texttt{x}_{i-1}\}$. Also, the resulting set of chains $\tau_q \subseteq \tau_n \cup \tau_{n+1}$, as specified by rule (For).

We denote by h the height of the derivation tree for q. For each variable $\texttt{x}_i$, chains in τ_i are obtained by a derivation of height $h_i < h$. By inductive hypothesis, the chains bound to $\texttt{x}_i$ can be folded to k_i-chains, where k_i is

$$k_i = \max_{a \in \Sigma} \{ \sum_{\texttt{step}=\texttt{step}_1}^{\texttt{step}_i} \mathcal{F}(a, \texttt{step}) + \mathcal{R}(\texttt{step}) \}$$

Since $\tau_q \subseteq \tau_n \cup \tau_{n+1}$, and all chains in τ_n are k_q-chains, what is left to show is that all chains inferred for $\texttt{step}_{n+1}$ can be folded to k_{n+1} chains, provided that the static environment Γ binds variables with chains that can be folded to k_n chains. This allows us to conclude the proof since $k_{n+1} = k_q$.

We proceed with a case analysis on $\texttt{step}_i$, assuming as before that $\phi_i = \mathsf{a} \in \Sigma$. We focus on the cases where $\tau_q \neq \varnothing$, i.e, the inference yields a non-empty result for some chains bound to $\texttt{x}$ in Γ. When $\tau_q = \varnothing$ the thesis follows straightforwardly.

$[\texttt{step}_{n+1} = \texttt{x}/\texttt{child} :: \mathsf{a}]$ In this case

$$\begin{aligned} \tau_{n+1} &= \bigcup_{\mathsf{c} \in \Gamma(\texttt{x})} \mathbf{T}_{\mathsf{C_d}}(\, \mathbf{A}_{\mathsf{C_d}}(\, \mathsf{c}\,,\, \texttt{child}\,)\,,\, \mathsf{a}\,) \\ &= \{\, \mathsf{c.a} \mid \mathsf{c} \in \Gamma(\texttt{x}) \,\} \end{aligned}$$

By inductive hypothesis, for each $\mathsf{c} \in \Gamma(\texttt{x})$ there exists a k_n-chain $\mathsf{c}' \in \Gamma(\texttt{x})$ such that $\mathsf{c} \hookrightarrow_{\mathsf{d}}^{*} \mathsf{c}'$. Provided this, it follows $\mathsf{c.a} \hookrightarrow_{\mathsf{d}}^{*} \mathsf{c}'.\mathsf{a}$. We need to show that $\mathsf{c}'.\mathsf{a}$ is a k_{n+1}-chain. We do this by distinguishing two cases, depending on wether $k_n < k_{n+1}$ or $k_n = k_{n+1}$.

If $k_n < k_{n+1}$, c' being a k_n-chain, it follows that $\mathsf{c}'.\mathsf{a}$ is a k_{n+1}-chain.

If $k_n = k_{n+1}$, then $\mathsf{c}'.\mathsf{a}$ is a k_n-chain because a appears strictly less than k_n times in c'. In other words, a is not the most frequent label in $\mathtt{step}_1, \ldots, \mathtt{step}_n$. This can be shown as follows by considering that

$$\sum_{i=1}^{n} \mathcal{F}(\mathsf{a}, \mathtt{step}_i) + \mathcal{R}(\mathtt{step}_i) \quad < \quad \sum_{i=1}^{n+1} \mathcal{F}(\mathsf{a}, \mathtt{step}_i) + \mathcal{R}(\mathtt{step}_i)$$

and provided that

$$\max_{a \in \Sigma}\{\sum_{i=1}^{n} \mathcal{F}(a, \mathtt{step}_i) + \mathcal{R}(\mathtt{step}_i)\} \quad = \quad \max_{a \in \Sigma}\{\sum_{i=1}^{n+1} \mathcal{F}(a, \mathtt{step}_i) + \mathcal{R}(\mathtt{step}_i)\}$$

we conclude the case because

$$\sum_{i=1}^{n} \mathcal{F}(\mathsf{a}, \mathtt{step}_i) + \mathcal{R}(\mathtt{step}_i) < k_n$$

In the following we denote by c_a the fact that a is the last symbol of the chain c, that is $\mathsf{c} = \mathsf{c}'.\mathsf{a}$.

$[\mathtt{step}_{n+1} = \mathtt{descendant} :: \mathsf{a}]$ In this case

$$\begin{array}{rcl} \tau_{n+1} & = & \bigcup\limits_{\mathsf{c} \in \Gamma(\mathsf{x})} \mathbf{T}_{\mathsf{C_d}}(\, \mathbf{A}_{\mathsf{C_d}}(\, \mathsf{c}\,, \mathtt{descendant}\,)\,, \mathsf{a}\,) \\ & = & \{\, \mathsf{c}.\mathsf{c}_\mathsf{a} \in \mathsf{C_d} \mid \mathsf{c} \in \Gamma(\mathsf{x}) \,\} \end{array}$$

Given $\mathsf{c}.\mathsf{c}_\mathsf{a} \in \tau_{n+1}$, by inductive hypothesis there exists a k_n-chain $\mathsf{c}' \in \Gamma(\mathsf{x})$ such that $\mathsf{c} \hookrightarrow_\mathsf{d}^* \mathsf{c}'$. By Lemma 9.2.6 the suffix c_a can be folded to a 1-chain. [2] At this point we have that $\mathsf{c}.\mathsf{c}_\mathsf{a} \hookrightarrow_\mathsf{d}^* \mathsf{c}'.\mathsf{c}'_\mathsf{a}$. Provided that c' is a k_n-chain and c'_a is a 1-chain, $\mathsf{c}'.\mathsf{c}'_\mathsf{a}$ is $(1{+}k_n)$-chain. We conclude since $\mathcal{R}(\mathtt{descendant}) = 1$ and thus $k_{n+1} = k_n + 1$.

$[\mathtt{step}_{n+1} = \mathtt{descendant\text{-}or\text{-}self} :: \mathsf{a}]$ Analogous to the case `descendant`.

$[\mathtt{step}_{n+1} = \mathtt{parent} :: \mathsf{a}]$ In this case

$$\begin{array}{rcl} \tau_{n+1} & = & \bigcup\limits_{\mathsf{c} \in \Gamma(\mathsf{x})} \mathbf{T}_{\mathsf{C_d}}(\, \mathbf{A}_{\mathsf{C_d}}(\, \mathsf{c}\,, \mathtt{parent}\,)\,, \mathsf{a}\,) \\ & = & \{\, \mathsf{c}_\mathsf{a} \mid \mathsf{c} = \mathsf{c}_\mathsf{a}.\mathsf{b} \in \Gamma(\mathsf{x}) \,\} \end{array}$$

[2] Notice however that $\mathsf{c}_2.\mathsf{a} \hookrightarrow_\mathsf{d}^* \mathsf{c}'_2$ does not imply $\mathsf{c}_2 \hookrightarrow_\mathsf{d}^* \mathsf{c}'_2$. For instance, $\mathsf{b}.\mathsf{a}.\mathsf{b} \hookrightarrow_\mathsf{d} \mathsf{b}$, yet $\mathsf{b}.\mathsf{a} \not\hookrightarrow_\mathsf{d} \mathsf{b}$.

where $\mathsf{b} \in \Sigma \cup \{\mathsf{String}\}$.

We want to show a folding for $\mathsf{c_a.b}$ and for $\mathsf{c_a}$. By inductive hypothesis, for each $\mathsf{c_b} = \mathsf{c_a.b} \in \Gamma(\mathtt{x})$ there exists a k_n-chain $\mathsf{c'_b}$ in $\Gamma(\mathtt{x})$ such that $\mathsf{c_b} \hookrightarrow^*_\mathsf{d} \mathsf{c'_b}$. We have two cases to consider, depending on wether a and b are mutually recursive or not.

If a and b are not mutually recursive ($\mathsf{a} \Rightarrow_\mathsf{d} \mathsf{b}$ but $\mathsf{b} \not\Rightarrow^*_\mathsf{d} \mathsf{a}$) then none of the chains belonging to $\mathsf{C_d}$ has a cycle involving the two types. So by folding $\mathsf{c_b}$ to $\mathsf{c'_b}$ the suffix $\mathsf{a.b}$ is preserved. Let $\mathsf{c'_b} = \mathsf{c'_a.b}$, we conclude since $\mathsf{c_a} \hookrightarrow^*_\mathsf{d} \mathsf{c'_a}$.

If a and b are mutually recursive ($\mathsf{a} \Rightarrow^*_\mathsf{d} \mathsf{b}$ and $\mathsf{b} \Rightarrow^*_\mathsf{d} \mathsf{a}$) then the are two cases to consider. If the prefix $\mathsf{a.b}$ is preserved by some foldings then we conclude. We discuss the case where the prefix $\mathsf{a.b}$ is never preserved by any folding to $\mathsf{c'_b}$. This happens if the schema enforces that b is always defined in terms of a (i.e., a is always a descendant of b) not the other way.

Notice that in this case $\mathsf{b} \neq \mathsf{String}$ and necessarily $\mathsf{b} \in \Sigma$. This implies that the first occurrence of b precedes any occurrence of b in the input chain $\mathsf{c_b}$, hence $\mathsf{c_b}$ is of the form

$$\mathsf{c_b} = \mathsf{c}_1.\underbrace{\mathsf{b.b}_1 \,..\, \mathsf{b}_n\mathsf{.a.b}}_{w}$$

and the cycle w is folded in order to respect the multiplicity value k_n, thus obtaining $\mathsf{c'_b} = \mathsf{c'_1.b}$. [3]

Then we do the following construction. Take $w' = \mathsf{b}.w_\mathsf{a}.\mathsf{b}$ as the folding of w, where $last(w_\mathsf{a}) = \mathsf{a}$ and $w_\mathsf{a}.\mathsf{b}$ is a 1-chain. Concatenate $\mathsf{c'_1}$ with w', thus obtaining a $(1{+}k_n)$-chain

$$\mathsf{c'} = \mathsf{c'_1}.w'$$

Now it remains to show that $k_{n+1} = 1 + k_n$. We show it by contradiction. Assume $k_{n+1} = k_n$. Then, as shown for the case $\mathtt{axis} = \mathtt{child}$, a is not the most frequent label in $\mathtt{step}_1, \ldots, \mathtt{step}_n$. Since a belongs to w_a, then also this symbols are not the most frequent in $\mathtt{step}_1, \ldots, \mathtt{step}_n$. Thus there exists a chain $\mathsf{c'_a}$ preserving the suffix $b.a$. A contradiction.

$[\mathtt{step}_{n+1} = \mathtt{ancestor} :: \mathsf{a}]$ In this case

$$\begin{aligned}\tau_{n+1} &= \bigcup_{\mathsf{c} \in \Gamma(\mathtt{x})} \mathsf{T_{C_d}}(\mathsf{A_{C_d}}(\mathsf{c}, \mathtt{ancestor}), \mathsf{a}) \\ &= \{\, \mathsf{c_a} \mid \mathsf{c} = \mathsf{c_a.c'} \in \Gamma(\mathtt{x}) \,\}\end{aligned}$$

with $\mathsf{c'} \neq \epsilon$. We want to show a folding for $\mathsf{c_a}$.

We have two cases to consider, depending on wether a is recursive or not.

[3] This happens for instance for schemas such as $\mathsf{d}_1 = \{\mathsf{s} \leftarrow \mathsf{a};\ \mathsf{a} \leftarrow \mathsf{b}?\}$ or $\mathsf{d}_2 = \{\mathsf{s} \leftarrow (\mathsf{a}, \mathsf{b});\ \mathsf{a} \leftarrow \mathsf{b}?;\ \mathsf{b} \leftarrow \mathsf{a}*\}$ or $\mathsf{d}_3 = \{\mathsf{s} \leftarrow \mathsf{a};\ \mathsf{a} \leftarrow \mathsf{b}_1?; \ldots \mathsf{b}_{n-1} \leftarrow \mathsf{b}_n; \mathsf{b}_n \leftarrow a\}$ and a query such as $q = //a/\mathtt{parent} :: \mathsf{a}$

If a is not recursive, then no chain in $\mathsf{C_d}$ has a cycle involving a. The type a is in a path between two distinct strongly-connected components of the dependency graph induced by the schema. Therefore $\mathsf{c} = \mathsf{c_a}.\mathsf{c}_2$ is of the form

$$\mathsf{c} = \mathsf{c}_1.\,\mathsf{a}.\,\mathsf{c}_2$$

where c_1 and c_2 do not share any label. By inductive hypothesis, there exists a k_n-chain $\mathsf{c}' \in \Gamma(\mathtt{x})$ such that $\mathsf{c} \hookrightarrow^*_{\mathsf{d}} \mathsf{c}'$. Since c_1 and c_2 do not share any label, it is easy to see that

$$\mathsf{c}' = \mathsf{c}'_1.\,\mathsf{a}.\,\mathsf{c}'_2$$

where $\mathsf{c}_i \hookrightarrow^*_{\mathsf{d}} \mathsf{c}'_i$, and c'_i is a k_n-chain, $(i = 1, 2)$. Then $\mathsf{c}' = \mathsf{c}'_1.\mathsf{a}$ is a k_n-chain folding of $\mathsf{c_a}$, and a k_{n+1}-chain as well.

If a is recursive, then we distinguish two sub cases cases. If $\mathtt{step}_1, \ldots, \mathtt{step}_n$ do not employ any recursive axis, then no $\mathsf{c} \in \Gamma(\mathtt{x})$ need to be folded. Therefore the proof is concluded straightforwardly by inductive hypothesis on the derivation of the expression that generated the chains for $\mathtt{x}$.

Otherwise, if some steps in $\mathtt{step}_1, \ldots, \mathtt{step}_n$ employ a recursive axis then, analogously to the case of $\mathtt{axis} = \mathtt{parent}$, the label a may not be preserved by the folding relation. If a is preserved then the conclusion is immediate. We assume this is not the case.

By inductive hypothesis, there exists a k_n-chain $\mathsf{c}' \in \Gamma(\mathtt{x})$ such that $\mathsf{c} \hookrightarrow^*_{\mathsf{d}} \mathsf{c}'$, and c' is of the form

$$\mathsf{c}' = \mathsf{b}_1.\mathsf{b}_2 \ldots \mathsf{b}_m$$

and $\mathsf{b}_i \neq \mathsf{a}$ for all $i \in \{1, .., m\}$.

We describe a procedure for constructing the desired k_{n+1}-chain.

For all $i = 1..m$, if

$$\mathsf{b}_i \Rightarrow^*_{\mathsf{d}} \mathsf{a} \qquad \text{and} \qquad \mathsf{a} \Rightarrow^*_{\mathsf{d}} \mathsf{b}_{i+1}$$

then construct a new chain from c', by injecting a word u between b_i, such as to obtain a chain of the form

$$\mathsf{c}''_i = \mathsf{b}_1 ..\mathsf{b}_i.u.\mathsf{b}_{i+1} ..\mathsf{b}_m$$

such that $u = \mathsf{a}_1 \ldots \mathsf{a}_k$ is a 1-chain with $\mathsf{a} \in u$, and $\mathsf{b}_i \Rightarrow_{\mathsf{d}} \mathsf{a}_1$ and $\mathsf{a}_n \Rightarrow_{\mathsf{d}} \mathsf{b}_{i+1}$.

Now, to check that c''_i we obtained from the construction is a chain matching the semantics of $\mathtt{step}_n$ it suffices to check that $\mathsf{c}''_i \in \Gamma(\mathtt{x})$. If this is not the case, then we chose another b_i and we repeat the process. If no other b_i exists than the step is unsatisfiable. If $\mathsf{c}''_i \in \Gamma(\mathtt{x})$ then we define $\mathsf{c}''_i = \mathsf{c}''_{\mathsf{a}}.\mathsf{c}''$, where $\mathsf{c}''_{\mathsf{a}}$ is the prefix of c''_i terminating with a. According to chain inference for the ancestor axis, we have that $\mathsf{c}''_{\mathsf{a}} \in \tau_{n+1}$ with $\mathsf{c}''_{\mathsf{a}}$ is a $(1{+}k_n)$-chain. Because $\mathcal{R}(\mathtt{ancestor}) = 1$, $k_{n+1} = 1 + k_n$ and we conclude. It remains to show that some c''_i constructed as above belong

to $\Gamma(\mathtt{x})$. Assume that for all $\mathtt{c}_i''$ we have that $\mathtt{c}_\mathtt{a}'' \notin \Gamma(\mathtt{x})$ then we conclude that $\mathtt{step}_1, \ldots, \mathtt{step}_n$ do not employ recursive navigations. A contradiction.

$[\mathtt{step}_{n+1} = \mathtt{following\text{-}sibling} :: \mathtt{a}]$ In this case

$$\begin{aligned} \tau_{n+1} &= \bigcup_{\mathtt{c} \in \Gamma(\mathtt{x})} \mathbf{T}_{\mathtt{C_d}}(\, \mathbf{A}_{\mathtt{C_d}}(\, \mathtt{c}\,, \mathtt{following\text{-}sibling}\,)\,, \mathtt{a}\,) \\ &= \{\, \mathtt{c}_p.\mathtt{a} \mid \mathtt{c} = \mathtt{c}_p.\mathtt{b} \in \Gamma(\mathtt{x})\ , \mathtt{b} <_{\mathtt{d}(\mathtt{c}_p)} \mathtt{a}\ \} \end{aligned}$$

We want to show a folding for $\mathtt{c}_p.\mathtt{a}$. By case $\mathtt{axis} = \mathtt{parent}$, we know that there exists a $(1+k_n)$-chain $\mathtt{c}'_p.\mathtt{a} \in \tau_n$ such that $\mathtt{c}_p.\mathtt{a} \hookrightarrow_\mathtt{d}^* \mathtt{c}'_p.\mathtt{a}$. Also, in this case the last label of $\mathtt{c}'_p$ may be $\mathtt{a}$ as well. Now independently from whether $\mathtt{a} = \mathtt{a}$ or not $\mathtt{c}'_p.\mathtt{a}$ is a $1+k_n$-chain, as well as $\mathtt{c}'_p.\mathtt{a}$. If $k_{n+1} = 1 + k_n$, then we conclude. Otherwise, if $k_{n+1} = k_n$ then, analogously to the case $\mathtt{axis} = \mathtt{child}$, we can show that $\mathtt{a}$ is not the most frequent label in $\mathtt{step}_1, \ldots, \mathtt{step}_n$, and thus that $\mathtt{c}'_p.\mathtt{a}$ is a k_n-chain.

□

Remark 9.2.7. *As a direct corollary of the lemma we can approximate the problem of* query unsatisfiability in presence of a schema *just looking on query k-chains. Indeed, if no k-chain is inferred for the input query expression, then no data match query needs. Moreover, for the subset of queries and schemas that make our chain inference system enjoying completeness, we can exactly determine wether a query is satisfiable or not in presence of a schema.*

9.2.2 Folding Lemma with conflict preservation

In this section we prove Lemma 4.4.5, by showing the existence of a folding relation that preserves also conflicts among chains. We show that such a folding exists for a pair of expressions if the multiplicity value we take is the sum of the multiplicity value of each expression.

Lemma 9.2.8. *Let* $\mathtt{d}$ *be a DTD,* $\mathtt{q}_1, \mathtt{q}_2$ *two queries with at most only one free variable* $\mathtt{x}$*, and* $\Gamma = \{\, \mathtt{x} \mapsto \mathtt{s_d}\ \}$ *a static environment. For all* $\mathtt{q}'_i \in \mathtt{LPQ}(\mathtt{q}_i)$*, provided that* $\Gamma \vdash_{\mathtt{C_d}} \mathtt{q}'_i : (\mathtt{r}_i, \mathtt{v}_i, \mathtt{e}_i)$ *and* $\tau_i = \mathtt{r}_i \cup \mathtt{v}_i \cup \mathtt{e}_i$*, for each chain* $\mathtt{c} \in \tau$ *there exists a chain* $\mathtt{c}' \in \tau$ *such that* $\mathtt{c} \hookrightarrow_\mathtt{d}^* \mathtt{c}'$ *and* $\mathtt{c}'$ *is a* $k_\mathtt{q}$*-chain. For each pair of chains* $(\mathtt{c}_1, \mathtt{c}_2) \in \tau_1 \times \tau_2$ *such that* $\mathtt{c}_1 \preceq \mathtt{c}_2$*, there exists* $(\mathtt{c}'_1, \mathtt{c}'_2) \in \tau_1 \times \tau_2$ *such that* $\mathtt{c}'_1 \preceq \mathtt{c}'_2$ *and* $\mathtt{c}_i \hookrightarrow_\mathtt{d}^* \mathtt{c}'_i$ *with* $\mathtt{c}'_i$ *a* $(k_{\mathtt{q}'_1} + k_{\mathtt{q}'_2})$*-chain* $(i{=}1,2)$*.*

Proof. We consider first the case when $\mathtt{c}_1, \mathtt{c}_2$ are either used or return chains. Since $\mathtt{c}_1 \preceq \mathtt{c}_2$, we have that $\mathtt{c}_2$ is the longest chain and we make a case analysis on $\mathtt{q}_2$. We distinguish three main cases depending on whether $\mathtt{q}_2$ contains some recursive axis or not. If $\mathtt{q}_2$ does not contains (forward or backward) recursive axis, $\tau_{\mathtt{q}_2}$ is finite. Moreover,

all chains in $\tau_{\mathtt{q}_2}$ have height upper bounded by $|\mathtt{q}_2|$ and label frequency upper bounded by

$$k_{\mathtt{q}_2} = \max_{a \in \Sigma} \{ \sum_{\mathtt{step} \in \mathtt{q}_2} \mathcal{F}(a, \mathtt{step}) \}$$

Since $\mathtt{c}_1 \preceq \mathtt{c}_2$, it follows that $\mathtt{c}_1$ is a $k_{\mathtt{q}_2}$-chain as well. Let $k{=}k_{\mathtt{q}_1} + k_{\mathtt{q}_2}$ we conclude that all pairs of conflicting chains are k-chains, as desired. Notice that if $\mathtt{q}_2$ also uses recursive backward navigation the statement still hold. As a matter of facts, a backward navigation would just backtrack on the input chain, without affecting the property that chains inferred for $\mathtt{q}_2$ are $k_{\mathtt{q}_2}$ chains.

If $\mathtt{q}_2$ uses at least one recursive *forward* axis, then an upper bound of the label frequency of chains witnessing the conflict is

$$k = k_{\mathtt{q}_1} + k_{\mathtt{q}_2}$$

The case requiring the largest k value occur when each prefix of $\mathtt{q}_2$ navigates some descendant of nodes selected by $\mathtt{q}_1$, in the sense that $\mathtt{c}_2 \in \tau_{\mathtt{q}_2}$, when $\mathtt{q}_2$ has only one free variable $\mathtt{x}$, and the chains are derived with the judgment

$$\Gamma[\mathtt{x} \mapsto \mathtt{c}_1] \vdash_{\mathtt{C_d}} \mathtt{q}_2 : \tau_{\mathtt{q}_2}$$

In this case let $\mathtt{q}_1 \circ \mathtt{q}_2$ be the composition of the two expressions, that is [4]

$$\mathtt{q}_1 \circ \mathtt{q}_2 \stackrel{def}{=} \mathtt{for}\ \bar{\mathtt{x}}\ \mathtt{in}\ \mathtt{q}_1\ \mathtt{return}\ \mathtt{q}_2$$

By definition we have that $k_{\mathtt{q}_1 \circ \mathtt{q}_2} = k_{\mathtt{q}_1} + k_{\mathtt{q}_2}$. By Lemma 4.4.4 we can fold $\mathtt{c} \in \tau_{\mathtt{q}_1 \circ \mathtt{q}_2}$ to a $k_{\mathtt{q}_1 + \mathtt{q}_2}$-chain $\mathtt{c}'$, such that $\mathtt{c}'$ is of the form $\mathtt{c}'_1.\mathtt{c}''$, with $\mathtt{c}'_1$ a $k_{\mathtt{q}_1}$-chain for $\mathtt{q}_1$ and $\mathtt{c}'$ a $(k_{\mathtt{q}_1}{+}k_{\mathtt{q}_2})$-chain for $\mathtt{q}_2$, thus exhibiting a pair of folded chains as desired.

In the remaining case where some $\mathtt{q}'_2$ prefix of $\mathtt{q}_2$ does not navigate descendants of nodes returned by $\mathtt{q}_1$, then either $\mathtt{q}'_2$ is navigating some ancestors of types returned by $\mathtt{q}_1$, or $\mathtt{q}'_2$ is navigating in parallel a node which is also navigated by $\mathtt{q}_1$ (and of course in order to have a conflict, a sibling navigation should follow in the sequel). In both cases we need a multiplicity value upper bounded by the one given by the case shown above, when each $\mathtt{q}'_2$ prefix of path $\mathtt{q}_2$ navigates some descendant of $\mathtt{c}_1$.

Correctness when either $\mathtt{c}_1$ or $\mathtt{c}_2$ are element chains follows, since element chains are either completely specified, and then folding is not needed (seen that label frequency is precisely kept into account by $\mathcal{F}(,)$), or they have an head completely specified and tail taken from a return chain, for which we assume the thesis holds by induction. □

[4] Notice that $\mathtt{q}_1 \circ \mathtt{q}_2$ may not be linear anymore, thus with a little abuse of notation we denote by $\mathtt{q}_1 \circ \mathtt{q}_2$ also $R^{\bullet}_{\mathtt{for}}(\mathtt{q}_1 \circ \mathtt{q}_2)$

Notice that the lemma states something stronger than what is needed for query-update independence purpose. In fact, it does not only concern used and return chains, but it also takes into account element chains, even if they are not involved in the definition of independence.

As before, we generalize from *lp* queries to the whole language.

Lemma 4.4.5 (*k*-FOLDING AND CONFLICT PRESERVATION) *Let* d *be a DTD,* $\mathsf{q}_1, \mathsf{q}_2$ *two queries with at most only one free variable* $\mathtt{x}$*, and* $\Gamma = \{ \mathtt{x} \mapsto \mathsf{s}_\mathsf{d} \}$ *a static environment. Provided that* $\Gamma \vdash_{\mathsf{C}_\mathsf{d}} \mathsf{q}_i : (\mathsf{r}_i, \mathsf{v}_i, \mathsf{e}_i)$ *and* $\tau_i = \mathsf{r}_i \cup \mathsf{v}_i \cup \mathsf{e}_i$*, for each pair of chains* $(\mathsf{c}_1, \mathsf{c}_2) \in \tau_{\mathsf{q}_1} \times \tau_{\mathsf{q}_2}$ *such that* $\mathsf{c}_1 \preceq \mathsf{c}_2$*, there exists* $(\mathsf{c}'_1, \mathsf{c}'_2) \in \tau_{\mathsf{q}_1} \times \tau_{\mathsf{q}_2}$ *such that* $\mathsf{c}'_1 \preceq \mathsf{c}'_2$ *and* $\mathsf{c}_i \rightsquigarrow^*_\mathsf{d} \mathsf{c}'_i$ *with* c'_i *a* $(k_{\mathsf{q}_1}{+}k_{\mathsf{q}_2})$*-chain* $(i{=}1,2)$*.*

Proof. Let $\mathtt{LPQ}(\mathsf{q}_i)$ be the set of linear-path queries associated to q_i, and $k'_i = \max\{k_{\mathsf{q}'_i} | \mathsf{q}'_i \in \mathtt{LPQ}(\mathsf{q}_i)\}$. By Lemma 9.2.2 we have that $\tau_{\mathtt{LPQ}(\mathsf{q}_i)} \supseteq \tau_{\mathsf{q}_i}$. Moreover, by Lemma 9.2.8 all pair of chains $(\mathsf{c}_1, \mathsf{c}_2) \in \tau_{\mathtt{LPQ}(\mathsf{q}_1)} \times \tau_{\mathtt{LPQ}(\mathsf{q}_2)}$ can be folded to a pair of $(k'_1{+}k'_2)$-chains preserving the conflict, and by Proposition 9.2.4 $k'_i \leq k_{\mathsf{q}_i}$. Thus all chains in $\tau_{\mathsf{q}_1} \times \tau_{\mathsf{q}_2}$ can be folded into $k_{\mathsf{q}_1}{+}k_{\mathsf{q}_2}$-chains. □

Remark 9.2.9. *As a direct corollary of Lemma 4.4.5 we can approximate the problem of* queries disjunction in presence of a schema *just looking on query k-chains. Indeed, if no pair of conflicting k-chain is found for the input query expressions, then no nodes are selected by both input query expressions on any valid document instance. Furthermore, for the subset of queries and schemas that make our chain inference system enjoying completeness, we can exactly solve the query intersection problem in presence of a schema.*

We conclude this section by showing the equivalence of the chain analysis in the infinite and finite case.

Theorem 9.2.10. *Let* d *be a DTD,* q *a query and* $\mathtt{u}$ *an update, with at most only one free variable* $\mathtt{x}$ *and* $\Gamma = \{ \mathtt{x} \mapsto \mathsf{s} \}$ *a static environment. Then,* $\mathsf{q} \perp_{\mathsf{C}_\mathsf{d}} \mathtt{u}$ *iff* $\mathsf{q} \perp_{\mathsf{C}^k_\mathsf{d}} \mathtt{u}$*.*

Proof. Direction $\Rightarrow$) is straightforward since $\mathsf{C}^k_\mathsf{d} \subseteq \mathsf{C}_\mathsf{d}$. We show now direction $\Leftarrow$) by showing that $\mathsf{q} \not\perp_{\mathsf{C}_\mathsf{d}} \mathtt{u}$ implies $\mathsf{q} \not\perp_{\mathsf{C}^k_\mathsf{d}} \mathtt{u}$.

By structural induction on $\mathtt{u}$.
Base.

[$\mathtt{u} = ()$] No chain is inferred for the empty update, hence it is independent wrt any query.

[u = `delete` q_0] In this case $\mathsf{q} \not\perp_{\mathsf{C_d}} \mathsf{u}$ implies

$$\mathsf{confl}(\mathsf{r_q}, \mathsf{U_u}) \cup \mathsf{confl}(\mathsf{U_u}, \mathsf{r_q}) \cup \mathsf{confl}(\mathsf{U_u}, \mathsf{v_q}) \neq \varnothing$$

Since $\mathsf{U}_u = \mathsf{r}_{\mathsf{q}_0}$ (up to ":" separator), let $k = k_{\mathsf{q}} + k_{\mathsf{q}_0}$.
By lemma 4.4.5

$$\mathsf{confl}(\mathsf{r_q}, \mathsf{r}_{\mathsf{q}_0}) \neq \varnothing \quad \textit{implies} \quad \mathsf{confl}(\mathsf{r}^k_{\mathsf{q}}, \mathsf{r}^k_{\mathsf{q}_0}) \neq \varnothing$$

$$\mathsf{confl}(\mathsf{r}_{\mathsf{q}_0}, \mathsf{r_q}) \neq \varnothing \quad \textit{implies} \quad \mathsf{confl}(\mathsf{r}^k_{\mathsf{q}_0}, \mathsf{r}^k_{\mathsf{q}}) \neq \varnothing$$

$$\mathsf{confl}(\mathsf{r}_{\mathsf{q}_0}, \mathsf{v_q}) \neq \varnothing \quad \textit{implies} \quad \mathsf{confl}(\mathsf{r}^k_{\mathsf{q}_0}, \mathsf{v}^k_{\mathsf{q}}) \neq \varnothing$$

and therefore we conclude $\mathsf{q} \not\perp_{\mathsf{C}^k_{\mathsf{d}}} \mathsf{u}$.

[u = `rename` q_0 `as` a] In this case $\mathsf{q} \not\perp_{\mathsf{C_d}} \mathsf{u}$ implies

$$\mathsf{confl}(\mathsf{r_q}, \mathsf{U_u}) \cup \mathsf{confl}(\mathsf{U_u}, \mathsf{r_q}) \cup \mathsf{confl}(\mathsf{U_u}, \mathsf{v_q}) \neq \varnothing$$

Recall that $\mathsf{U_u} = \mathsf{U}_1 \cup \mathsf{U}_2$ where

$$\mathsf{U}_1 = \{\ \mathsf{c{:}b} \mid \mathsf{c.b} \in \mathsf{r}_{\mathsf{q}_0}\ \} \quad \text{and} \quad \mathsf{U}_2 = \{\ \mathsf{c{:}a} \mid \mathsf{c.b} \in \mathsf{r}_{\mathsf{q}_0}\ \}$$

For all conflicting update chain belonging to U_1 we conclude as for the `delete` case.

For all conflicting update chain belonging to U_2 we make a case analysis on the kind of conflict.

$\mathsf{confl}(\mathsf{r_q}, \mathsf{U}_2)$ For all $(\mathsf{c_q}, \mathsf{c_u}) \in \mathsf{r_q} \times \mathsf{U}_2$ such that $\mathsf{c_q} \preceq \mathsf{c_u}$ we have two cases to consider: $\mathsf{c_q} \prec \mathsf{c_u}$ and $\mathsf{c_q} = \mathsf{c_u}$. If $\mathsf{c_q} \prec \mathsf{c_u}$ then $\mathsf{confl}(\mathsf{r_q}, \mathsf{U}_2) \neq \varnothing$ if and only if $\mathsf{confl}(\mathsf{r_q}, \mathsf{U}_1) \neq \varnothing$ because the conflict arises on a label preceding the renamed one. Then reasoning as for U_1 we conclude $\mathsf{q} \not\perp_{\mathsf{C}^k_{\mathsf{d}}} \mathsf{u}$. If $\mathsf{c_q} = \mathsf{c_u}$, we distinguish again if $\mathsf{confl}(\mathsf{r_q}, \mathsf{U}_1) \neq \varnothing$ or not. If $\mathsf{confl}(\mathsf{r_q}, \mathsf{U}_1) \neq \varnothing$ then we conclude $\mathsf{q} \not\perp_{\mathsf{C}^k_{\mathsf{d}}} \mathsf{u}$ as before. If $\mathsf{confl}(\mathsf{r_q}, \mathsf{U}_1) = \varnothing$ then in order for the update to be valid wrt the schema a is required to be a sibling of a type returned by q_0. Since $\mathsf{confl}(\mathsf{r_q}, \mathsf{U}_1) = \varnothing$ then $\mathsf{c.b} \notin \mathsf{r}_0$ and this means that q_0 specifies a label $\mathsf{b} \neq \mathsf{a}$ in the step generating $\mathsf{c.b}$. We modify q_0 substituting b with a and we conclude by Lemma 4.4.5 on q, q_0 noticing that adding a a tag in q_0 raises a label frequency of 1 at most.

$\mathsf{confl}(\mathsf{U}_2, \mathsf{r_q})$ For all $(\mathsf{c_u}, \mathsf{c_q}) \in \mathsf{U}_2 \times \mathsf{r_q}$ such that $\mathsf{c_u} \preceq \mathsf{c_q}$ we only consider: $\mathsf{c_u} \prec \mathsf{c_q}$ since $\mathsf{c_q} = \mathsf{c_u}$ has been adressed in the case of $\mathsf{confl}(\mathsf{r_q}, \mathsf{U}_2)$. With a case analysis on the emptiness of $\mathsf{confl}(\mathsf{U}_1, \mathsf{r_q})$ and by substituting a with b in q_0, we conclude $\mathsf{q} \not\perp_{\mathsf{C}^k_{\mathsf{d}}} \mathsf{u}$ as before.

$\mathsf{confl}(\mathsf{U}_2, \mathsf{v_q})$ Identical to $\mathsf{confl}(\mathsf{U}_2, \mathsf{r_q})$.

[$\mathtt{u} = \mathtt{insert}\ \mathsf{q}_1\ \mathtt{pos}\ \mathsf{q}_0$] In this case $\mathsf{q} \not\perp_{\mathsf{C_d}} \mathtt{u}$ implies

$$\mathsf{confl}(\mathsf{r_q}, \mathsf{U_u}) \cup \mathsf{confl}(\mathsf{U_u}, \mathsf{r_q}) \cup \mathsf{confl}(\mathsf{U_u}, \mathsf{v_q}) \neq \varnothing$$

We consider the case $\mathtt{pos} = \mathtt{into}$ since other cases are similar. Recall that $\mathsf{U} = \mathsf{U}_1 \cup \mathsf{U}_2$ where

$$\begin{aligned} \mathsf{U}_1 &= \{\ \mathsf{c}{:}\mathsf{e} \mid \mathsf{c} \in \mathsf{r}_{\mathsf{q}_0} \quad \mathsf{e} \in \mathsf{e}_{\mathsf{q}_1}\ \} \\ \mathsf{U}_2 &= \{\ \mathsf{c}{:}\alpha.\mathsf{c}' \mid \mathsf{c} \in \mathsf{r}_{\mathsf{q}_0} \quad \mathsf{c}_1.\alpha \in \mathsf{r}_{\mathsf{q}_1} \mathsf{c}_1.\alpha.\mathsf{c}' \in \mathsf{C_d}\ \} \end{aligned}$$

We consider first conflicts generated by U_1

$\mathsf{confl}(\mathsf{r_q}, \mathsf{U}_1)$ For all $(\mathsf{c_q}, \mathsf{c_u}) \in \mathsf{r_q} \times \mathsf{U}_1$ such that $\mathsf{c_q} \preceq \mathsf{c_u}$, let $\mathsf{c_u} = \mathsf{c} : \mathsf{e}$ we have two cases to consider depending on wether $\mathsf{c_q} \preceq \mathsf{c}$ or $\mathsf{c} \prec \mathsf{c_q}$.

If $\mathsf{c_q} \preceq \mathsf{c}$, by Lemma 4.4.5 on q, q_0 we can fold $\mathsf{c_q}$ to c'_q and c to c' with $\mathsf{c}'_\mathsf{q}, \mathsf{c}'$ both $(k_\mathsf{q}{+}k_{\mathsf{q}_0})$-chains. Since e is created by element construction it is completely specified and it is easy to see that $\mathsf{c}'.\mathsf{e}$ is a $(k_\mathsf{q}{+}k_\mathsf{u})$-chain.

If $\mathsf{c} \prec \mathsf{c_q}$ by Lemma 4.4.5 on q, q_0 we can fold $\mathsf{c_q}$ to c'_q and c to c' with $\mathsf{c}'_\mathsf{q}, \mathsf{c}'$ both $(k_\mathsf{q}{+}k_{\mathsf{q}_0})$-chains. Let $\overline{\mathsf{e}} \preceq \mathsf{e}$ such that $\mathsf{c}.\overline{\mathsf{e}} = \mathsf{c_q}$, since e is completely specified by query syntax by in the element construction then we have $\mathsf{c}''_\mathsf{q} = \mathsf{c}''.\overline{\mathsf{e}}$ as a folding of $\mathsf{c_q}$ that preserves $\overline{\mathsf{e}}$ as a suffix, with c'' a folding of c as prefix, and c''_q is a $(k_\mathsf{q}{+}k_{\mathsf{q}_0})$-chain. Also, of course $\mathsf{c}''.\mathsf{e}$ is a $(k_\mathsf{q}{+}k_\mathsf{u})$-chain and we conclude $\mathsf{q} \not\perp_{\mathsf{C_d^k}} \mathtt{u}$.

$\mathsf{confl}(\mathsf{U}_1, \mathsf{r_q})$ Analogous to $\mathsf{confl}(\mathsf{r_q}, \mathsf{U}_1)$.

$\mathsf{confl}(\mathsf{U}_1, \mathsf{v_q})$ Analogous to $\mathsf{confl}(\mathsf{r_q}, \mathsf{U}_1)$.

Concerning U_2, we want to show that we can restrict the conflict analysis on a subset of U_2 namely U'_2 defined as

$$\mathsf{U}'_2 = \{\ \mathsf{c}{:}\alpha \mid \mathsf{c} \in \mathsf{r}_{\mathsf{q}_0} \quad \mathsf{c}_1.\alpha \in \mathsf{r}_{\mathsf{q}_1}\ \}$$

- If $\mathsf{confl}(\mathsf{r_q}, \mathsf{U}_2) = \varnothing$ then either $\mathsf{confl}(\mathsf{r_q}, \mathsf{U}'_2)$ or $\mathsf{confl}(\mathsf{U}'_2, \mathsf{r_q})$.
- If $\mathsf{confl}(\mathsf{U}_2, \mathsf{r_q}) = \varnothing$ then $\mathsf{confl}(\mathsf{U}'_2, \mathsf{r_q})$.
- If $\mathsf{confl}(\mathsf{U}_2, \mathsf{v_q}) = \varnothing$ then $\mathsf{confl}(\mathsf{U}'_2, \mathsf{v_q})$.

The proof for $\mathsf{confl}(\mathsf{r}, \mathsf{U}'_2)$ follows as for the `rename` case.

[$\mathtt{u} = \mathtt{replace}\ \mathsf{q}_0\ \mathtt{with}\ \mathsf{q}$] The statement follows as a combination of `delete` and `insert` cases.

Induction.

[$\mathtt{u} = \mathtt{u}_1, \mathtt{u}_2$] By inductive hypothesis on $\mathtt{u}_1, \mathtt{u}_2$.

[$\mathtt{u} = \mathtt{for\ x\ in\ q\ return\ u}_1$] By inductive hypothesis on the length of the derivation for $\mathtt{q}$ and then for $\mathtt{u}$. Analogous to the case of Lemma 9.2.5.

[$\mathtt{u} = \mathtt{let\ x\ :=\ q\ return\ u}_1$] Analogous to the for case.

[$\mathtt{u} = \mathtt{if\ (q)\ then\ u}_1\ \mathtt{else\ u}_2$] By inductive hypothesis on the length of the derivation for $\mathtt{q}, \mathtt{u}_1$ and $\mathtt{u}_2$.

□

Theorem 9.2.11 (Soudness of $\mathsf{C}^k_\mathtt{d}$ Independence)**.** *Let* $\mathtt{d}$ *be a DTD,* $\mathtt{q}$ *a query and* $\mathtt{u}$ *an update, with at most only one free variable* $\mathtt{x}$*, which is always bound, during static analysis, to the root type* $\mathtt{s_d}$ *and, during query evaluation, to the root location* l_t *of a valid tree* $t \in \mathtt{d}$*. Let* $k = k_\mathtt{q} + k_\mathtt{u}$ *as defined above, then* $\mathtt{q} \perp_{\mathsf{C}^k_\mathtt{d}} \mathtt{u}$ *implies* $\mathtt{q} \perp\!\!\!\perp_\mathtt{d} \mathtt{u}$

Proof. Straightforward since by Theorem 9.2.10 $\mathtt{q} \perp_{\mathsf{C}_\mathtt{d}} \mathtt{u}$ iff $\mathtt{q} \perp_{\mathsf{C}^k_\mathtt{d}} \mathtt{u}$, and by Theorem 4.4.2 $\mathtt{q} \perp_{\mathsf{C}_\mathtt{d}} \mathtt{u}$ implies $\mathtt{q} \perp\!\!\!\perp_\mathtt{d} \mathtt{u}$. □

Bibliography

[AAD+12] Serge Abiteboul, Yael Amsterdamer, Daniel Deutch, Tova Milo, and Pierre Senellart. Finding optimal probabilistic generators for XML collections. In *ICDT*, 2012.

[AFL02] Marcelo Arenas, Wenfei Fan, and Leonid Libkin. On verifying consistency of XML specifications. In *PODS*, 2002.

[AYCLS01] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of tree pattern queries. In *SIGMOD*, 2001.

[BBC+11] Mohamed Amine Baazizi, Nicole Bidoit, Dario Colazzo, Noor Malla, and Marina Sahakyan. Projection for XML update optimization. In *EDBT*, 2011.

[BC09a] Michael Benedikt and James Cheney. Schema-based independence analysis for XML updates. In *VLDB*, 2009.

[BC09b] Michael Benedikt and James Cheney. Semantics, types and effects for XML updates. In *DBPL*, 2009.

[BC10] Michael Benedikt and James Cheney. Destabilizers and independence of XML updates. *PVLDB*, 3(1), 2010.

[BCCN06] Véronique Benzaken, Giuseppe Castagna, Dario Colazzo, and Kim Nguyen. Type-based XML projection. In *VLDB*, 2006.

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *ICFP*, 2003.

[Ben10] Michael Benedikt. Analysis of declarative updates: invited talk. In *EDBT/ICDT Workshops*, 2010.

[BGMM09] Henrik Björklund, Wouter Gelade, Marcel Marquardt, and Wim Martens. Incremental XPath evaluation. In *ICDT*, 2009.

[BK06] Michael Benedikt and Christoph Koch. Interpreting tree-to-tree queries. In *ICALP*, 2006.

[BK09] Michael Benedikt and Christoph Koch. From XQuery to relational logics. *ACM TODS*, 34(4), 2009.

[BNSV10] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. Inference of concise regular expressions and DTDs. *ACM TODS*, 2010.

[BPSM+06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, FranÐois Yergeau, and John Cowan. Extensible Markup Language (XML) 1.1 (Second Edition). Technical report, W3C Consortium, 2006.

[BTCU10a] Nicole Bidoit-Tollu, Dario Colazzo, and Federico Ulliana. Detecting XML query-update independence. *26ème journée des Bases des données Avancées*, 2010.

[BTCU10b] Nicole Bidoit-Tollu, Dario Colazzo, and Federico Ulliana. Detecting XML query-update independence. *International Formal Methods Workshop*, 2010.

[BTCU12] Nicole Bidoit-Tollu, Dario Colazzo, and Federico Ulliana. Type-based detection of XML query-update independence. *PVLDB*, 5(9), 2012.

[CCF+06] Don Chamberlin, Michael Carey, Daniela Florescu, Donald Kossmann, and Jonathan Robie. XQueryP: Programming with XQuery. In *XIME-P*, 2006.

[CGM11] Federico Cavalieri, Giovanna Guerrini, and Marco Mesiti. Updating XML schemas and associated documents through exup. In *ICDE*, 2011.

[CGMS06] Dario Colazzo, Giorgio Ghelli, Paolo Manghi, and Carlo Sartiani. Static analysis for path correctness of XML queries. *Journal of Functional Programming*, 16, 2006.

[CGS11] Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Schemas for Safe and Efficient XML Processing, 2011. Tutorial - ICDE.

[Che08] James Cheney. FLUX: FunctionaL Updates for XML. In *ICFP*, 2008.

[Che09] James Cheney. XQuery Update analysis tools version 0.1, 2009. http://homepages.inf.ed.ac.uk/jcheney/programs/.

[CM01] James Clark and Makoto Murata. RELAX NG specification, 2001.

[DFF+10] D Draper, P Fankhauser, M Fernandez, A Malhotra, K Rose, M Rys, J Siméon, and P Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Technical report, World Wide Web Consortium, 2010.

[DM06] Bruno Dutertre and Leonardo De Moura. The Yices SMT solver. Technical report, 2006.

[FCG04] Wenfei Fan, Chee Yong Chan, and Minos N. Garofalakis. Secure XML querying with security views. In *SIGMOD*, 2004.

[Fra05] M. Franceschet. XPathMark - An XPath benchmark for XMark generated data. In *XSym*, 2005.

[GL08] Pierre Genevès and Nabil Layaïda. XML reasoning solver user manual. Technical report, R. Report 6726, INRIA, 2008.

[GLS07] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of xml paths and types. *PLDI*, 2007.

[GORS08] Giorgio Ghelli, Nicola Onose, Kristoffer Rose, and Jerome Simeon. XML query optimization in the presence of side effects. In *SIGMOD*, 2008.

[Gra03] Hervé Grall. *Deux critères de sécurité pour l'exécution de code mobile*. Thése, École des Ponts ParisTech, 2003.

[GRS06] Giorgio Ghelli, Christopher Ré, and Jérôme Simèon. XQuery!: An XML query language with side effects. In *EDBT Workshops*, 2006.

[GRS08] Giorgio Ghelli, Kristoffer Høgsbro Rose, and Jérôme Siméon. Commutativity analysis for XML updates. *ACM TODS*, 2008.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[Hid03] Jan Hidders. Satisfiability of XPath expressions. In *DBPL*, 2003.

[HKL05] Beda Christoph Hammerschmidt, Martin Kempa, and Volker Linnemann. On the intersection of XPath expressions. In *IDEAS*, 2005.

[HU00] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation, Second Edition*. Addison-Wesley, 2000.

[HVP05] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1), 2005.

[JGL12] Muhammad Junedi, Pierre Genevès, and Nabil Layaïda. XML query-update independence analysis revisited. *DocEng*, 2012.

[KM11] Nils Klarlund and Anders Moller. MONA version 1.4 user manual. Technical report, BRICS, 2011.

[MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.

[MS03] A. Marian and J. Siméon. Projecting XML documents. In *VLDB*, 2003.

[MS04] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of XPath. *Journal of ACM*, 2004.

[PAG10] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. nSPARQL: A navigational language for RDF. *Journal of Web Semantics*, 2010.

[PV00] Yannis Papakonstantinou and Victor Vianu. DTD inference for views of XML data. In *PODS*, 2000.

[RCD+11] J Robie, D Chamberlin, M Dyck, D Florescu, J Milton, and J Simeon. XQuery update facility 1.0. Technical report, W3C Consortium, 2011.

[RS06] Mukund Raghavachari and Oded Shmueli. Conflicting XML updates. In *EDBT*, 2006.

[SCF+07] Jérôme Siméon, Don Chamberlin, Daniela Florescu, Scott Boag, Mary F. Fernández, and Jonathan Robie. XQuery 1.0: An XML query language. W3C recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xquery-20070123/.

[SM73] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *STOC*, 1973.

[SWK+02] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, 2002.

[TBMM04] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures Second Edition. Technical report, World Wide Web Consortium, Oct 2004. W3C Recommendation.

[Vor96] Sergei G. Vorobyov. An improved lower bound for the elementary theories of trees. In *CADE*, 1996.