UNIVERSITÀ CA' FOSCARI – VENEZIA
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Specialistica in Informatica

Tesi di Laurea

A Formal Study for Type System for XQuery Optimization

Federico Ulliana

Relatore: Prof. Michele Bugliesi Correlatore: Dott. Dario Colazzo

Controrelatore: Prof. Renzo Orsini

Anno Accademico 2008-2009
Via Torino 155 – 30173 Mestre-Venezia

April 4 2009

A tutti i miei Mentore.

Contents

Abstract

Type based XML projection (or pruning) is an optimization technique adopted in the context of main memory XML query engines, introduced by Benzaken, Castagna, Colazzo and Nguyen in a recent VLDB paper.

The main idea behind XML projection is quite simple: given a query over an XML data tree valid with respect to a DTD, all subtrees of the document not necessary to evaluate the query are pruned out, thus obtaining a smaller tree. The query is then executed over a smaller tree, avoiding to allocate memory for nodes that will never be reached by navigational specifications in the query.

What is typical of type-based projection, is that the decision whether a subtree have to be pruned or not is based on the information related to the type of the query and to the type of the input document. In the VLDB paper, authors defined a type system for XPath, while XQuery queries are dealt with by approximating them by a set of XPath expression, which constitutes then the input for the type-projection analysis. Soundness and completeness results were provided only for XPath queries, and no property were proved for XQuery.

In this dissertation we provide an extension of the previous type system. The new type system is able to directly deal with XQuery queries, without needing path-extraction intermediate steps. This has in turn required a formal and concise characterization of XQuery semantics, which is much more complex than that of XPath. As a result we give and prove a soundness theorem for type-projection for XQuery queries, and we also show that the type system is able to infer very precise type projectors for a wide class of cases.

Chapter 1

Introduction

1.1 Towards Projection through Types

The last ten years have seen the rapidly emerging of XML query and transformation languages because of the large class of applications where XML plays a central role. Examples are Web applications, data integration, and P2P distributed database systems. What is clear is that, when dealing with heterogeneous data sources, it's difficult to grant a regular structure of the raw data such as, for instance, in the relational model. Then the needed of semistructured data. Semistructured data means data structure highly irregular which can change over the time. XML is a language that allows easily to define such data. With the type languages DTD and XML Schema it is possible to have a fine-grain control over how much variation allow in the data: from a perfect regularity, a little or a lot of variation.

Designers of XML made as optional the presence of a schema instance to validate a document. Despite this, document types are absolutely essential to the development of coherent complex applications, and then to guarantee documents to be valid wrt a schema. Moreover, they are useful to reason about data and collections of data. As in relational system, schema allow software to decide about possible optimizations, for instance, performing static type checking of programs and queries.

This work is collocated in the context of optimization for Main Memory XQuery Processors, in particular it studies the *pruning* technique based on *Type Projectors* introduced by Benzaken et al. in [1]. This technique stands on the fact that each node belonging to an XML tree valid wrt a schema can be associated with the chains of types that states between the root-node and the node-self. This means that it is possible to determine the type of

all the nodes in the result of a query and the type of their ancestors. Once computed this set of *useful* types, it is possible to prune the document in the parsing phase of the document avoiding to load in main memory all the subtrees for with root node type not in the computed set, as in Figure (3.5). This often reduces drastically the amount of the memory required for the

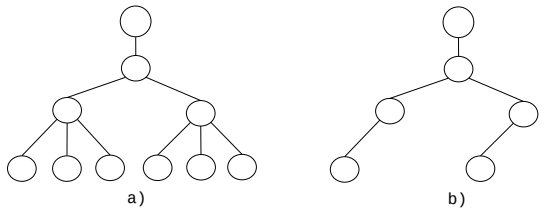

Figure 1.1: a) original document, b) pruned document

execution of a query, and it becomes especially important in main memory processors, because they are used to load all the document before executing a query.

In this work we will cover some lacks in the VLDB paper to the aim of manage all the type projectors technique on XQuery and give all the formal foundations needed. We reserve a paragraph later for a detailed explanation.

1.2 Main-memory XQuery Processors

As already stated, applications that process XML documents as files suffer limitations of current main-memory processors. In other words, they cannot manage very huge documents due to the memory occupancy of the XML data model instance. Also, little devices with small CPU and low memory that need to process portion of XML need to save resources if possible. The interest of this work is on main-memory XQuery processor for the query language XQuery proposed by W3C in [7]. After several years of development by the W3C, XQuery is becoming more stable and starts being implemented and used. Originally designed to query XML databases, now is being considered as an alternative in the context of many other applications, such as streaming, information integration, services and full text querying. Main memory XQuery processors are often the primary choice for those applications that cannot afford to build secondary storage indexes or load a database before starting query processing.

Memory occupation by data model instances

The implementations of main memory XQuery engines load the complete document before processing it. Moreover is need to build a data model instance of the entire document in memory before query processing because, to a large extent, of the complexity of evaluating languages such as XSLT and XQuery. Processing XML only as a stream without building a data model instance is an active area of research, but such approaches only consider fragments of XPath, and cannot deal with most XQuery expressions. Indeed, many XQuery expressions (joins, type operations such as typeswitch, operations on document order, backward XPath axis, function calls, let expressions, namespaces, sorting, etc.) require to materialize part(s) of the document. This is typically done using one of the existing XML data models which provide information necessary for query processing such as node identity, type annotations resulting from validation, namespace, nodes, pointers to parent nodes, etc.

The complexity of XML data models accentuate the problems related to memory management in XQuery implementations. Benchmarks show that the size of a DOM representation in memory is typically 4-5 times larger than the original file. Some techniques can be used to build a more compact representation. The aim is to avoiding to build a complete data model instance.

Processing XQuery in Main-Memory

We show how a pruning technique fits in a typical main-memory XQuery processor. We use the Galax system as an illustration. Galax is integrated with the pruning technique based on *path analysis* introduced by Marian and Siméon in [2]. The technique is briefly described later, but it's not necessary to be known to understand what follows.

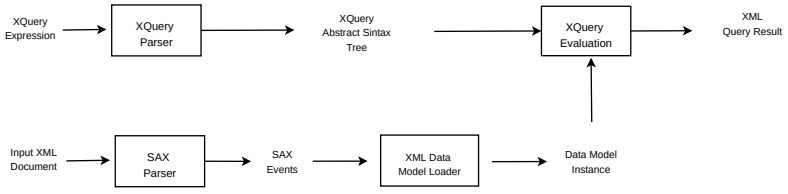

Figure 1.2: Galax XQuery processing architecture without projection paths

Architecture without pruning

Figure (1.2) shows the Galax processing architecture in the absence of projection. On the one hand, the XQuery expression is parsed to an abstract syntax tree, on the other hand, the input document is parsed into a streamed fashion using SAX, then loaded in memory as an XML data model instance. In the case of a streaming processor, the document is parsed directly from the network instead of a local file. Finally, the query is applied on the data model instance to yield a result.

Architecture with projection

The projection technique can be integrated in a main-memory XQuery processor with minimal effort, as showed in Figure (1.3). The figure show the technique developed in [2], then after the query is parsed, it is analysed to recognise *projection paths*. The result of this analysis is sent to the data model loader which uses it to build a *projected* data model instance which contains only the nodes specified by the projection paths. The figure explain the mechanism, but as we will see later the technique of projection paths suffers of overhead. Now we propose the architecture in presence of Type

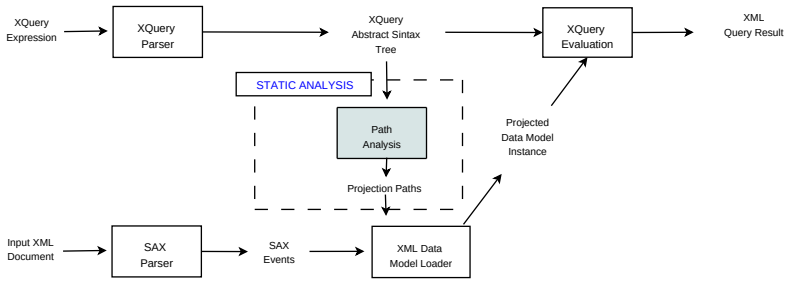

Figure 1.3: Galax XQuery processing architecture with projection paths

Projectors, the technique developed in [1]. We can see in Figure (1.4) that the main difference wrt the projeetion paths approach is the contribute of the schema to the type projection inference. This is what distinguished the two techniques. The figure shows clearly that XQuery is treated with an apart module, that is the path extraction function.
Finally, in Figure (1.5) we present the improvement that simplify the model and that is the aim of this work: static analyse directly XQuery avoiding the phase of path extraction.

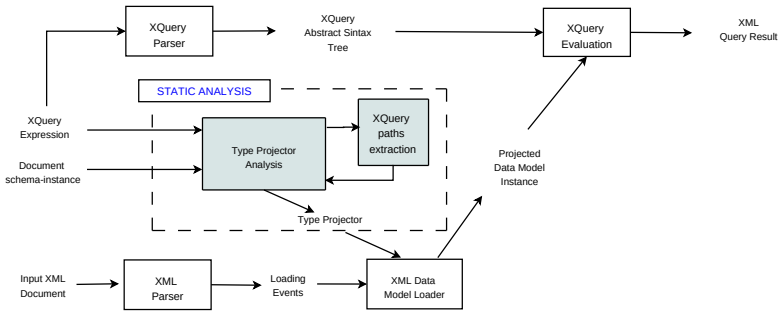

Figure 1.4: Architecture with type projectors

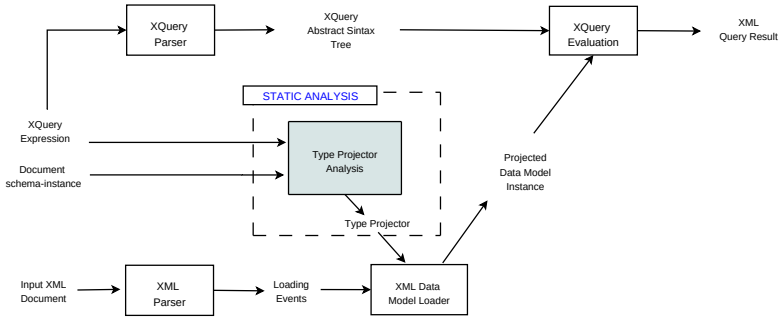

Figure 1.5: Architecture with type projectors simplified

1.3 State of the art

Three main pruning approaches are known: untyped query analysis, dataguide and typed query analysis.

[1] Marian and Siméon developed the technique based on *projection paths*. They propose that the part of data that is necessary to the execution of the query `q` is determined by statically extracting all paths in `q`. These paths are then applied to the document `d` at load time, in the SAX-event based fashion, in order to prune unneeded subtrees of the document, as in Galax. This technique is powerful since it can be applied to a set of queries over the same document and it does not require any apriori knowledge of the structure of the document `d` because only the query is analyzed. However, this technique suffers some limitations. First, the document loader-pruner is not able to manage backward axes nor path expressions with predicates. Also, since no information about `d` is used, the technique does not behave efficiently in terms of loading time and pruning precision (hence, memory allocation) when "//" occurs in paths. Indeed, when is present in a projection path, the pruning process requires to visit all descendants of a node in order to decide whether the node contains a useful descendant. What is worst is that pruning time tends to be quite high and it drastically increases (together with memory consumption) with the number of augments in the pruning path-set. As a matter of facts, in this technique pruning corresponds to computing a further query, whose time and memory occupation may be comparable to those required to compute the original query. In particular, every occurrence of "//" may yield a full exploration of the tree in this technique. Therefore, pruning execution overhead and its high memory footprint may jeopardize the gains obtained by using the pruned document. Finally, since no predicates in paths are treated, the precision of pruning degrades for queries containing the XPath expressions `descendant :: node[cond]`, which are very useful and used in practice.

The second main technique by Bressan in [20] introduce a different and quite precise XML pruning technique for a subset of XQuery FLOWR expressions. The technique is based on the apriori knowledge of a *data-guide* for `d`. A data guide is an auxiliary data structure that aid navigation of the data. The document `d` is first matched against an abstract representation of `q`. Pruning is then performed at run time, it is very precise, and, thanks to the use of some indexes over the data-guide, it ensures good improvements in terms of query execution time. However the technique is one-query oriented, in the sense that it cannot be applied to multiple queries, it does not handle XPath predicates, and cannot handle backward axes. Also, the

approach requires the construction and management of the data-guide and of adequate indexes.

Third approach, the one followed by this work, is by Benzaken et al. [1]. Authors present a new pruning approach which is applicable in the presence of typed XML data. Most applications require that data are valid with respect to some external schema, DTD or XML Schema. Their technique, that is well described in this dissertation, it's more closed with the work of Marian and Siméon but relaxing some limitations. First of all, unlike [2] and [20] the approach accounts for backward axes, performs a fine-grained analysis of predicates, allows for dealing with bunches of queries. The solution provides comparable or greater precision than the other approaches, while it requires always negligible or no pruning overhead. One of the particular features is that the pruning algorithm is characterized by a constant (and low) memory consumption and by an execution time linear in the size of the document to prune. More precisely, a pruning based on type projectors is equivalent to a single bufferless one-pass traversal of the parsed document that simply discards elements not generated by any of the names in the *projector*. So if embedded in query processors, pruning can be executed during parsing and/or validation and brings no overhead.

1.4 Thesis contribution

In this work we aim to overcome the following limitations of [1]:

- the lack of a formal treatment of all XPath horizontal navigational axes that are approximated as composition of other axes.
- the lack of a formal treatment of the most powerful constructs of XQuery that because of their complexity are approximated using a path extraction function that reduces them to a set of XPath paths.
- the availability of proofs only for XPath because XQuery is approximated to XPath.

We propose the following features:

- the formalization of the semantics of FLWR-XQuery, the language that contains all XPath, FLWR expressions and element construction.
- a set of typing rules for FLWR-XQuery and the proof of Soundness and Completeness for the type inference.

- a set of rules for type projection inference for FLWR-XQuery and the proof of Soundness for the type projection inference.

What gives originality to our work is the fact that the language that we treat formally is a relevant extension of the one proposed by the authors in [1]. We move in two ways: first covering a lack in XPath introducing features as dynamic and static analysis for axes `following–sibling`, `preceding–sibling`, `following` and `preceding` (that are approximated in the original work), and second managing the most powerful constructs of XQuery, that are, FLWR expressions and *element construction*. In the original work the latter constructs were considered using a special path-extraction function and then reducing the problem in the scope of path navigation static analysis because the formal treatment of XQuery seemed too hard. For the language that we consider, a subset of XQuery we dub FLWR-XQuery[1], semantics is defined in agreement with W3C specifications, and gives both an easy understandable way for the dynamic evaluations of the language and expressivity and flexibility in formal proof. This is far from being obvious, especially if we compare our works to the others in literature on the same branch of research. We define a static *type system* to type the result of a query, and a static *type projector* system to compute type projectors, for all FLWR-XQuery. We give a measure of the precision of our system since the soundness and the completeness of our approach for a large class of queries and DTDs are formally proved. Then considering as starting point the theorems that we will state a lot of new applications of the optimization technique can be studied.

1.5 Structure of the document

The thesis is organised as follows. Chapter 3 introduces basic definitions and notations: data model, DTD, validation. In Chapter 4 we define FLWR-XQuery and its semantics. In Chapter 5 we present the static type system and state its formal properties. In Chapter 6 we present projection, type projector, type projectors inference system and state its formal properties. In the last Chapter we proof in detail all the stated theorems.

[1]It's not accident that we omit "O" in the name of our language because we didn't treat `order by` clause.

Chapter 2

Data and Types

The purpose of this chapter is to introduce the language XML and its type languages, in order to define formally the data model and the type language model on which we based our work. For the typing, we focus on the DTD language that is illustrated both from a descriptive and a formal point of view. Moreover, from a theoretical perspective, it will be showed that grammars generated by DTD and XML Schema schemas corresponds to two particular subclasses of the *Regular Tree Grammars* (RTG).

2.1 XML and XML Documents

We start with a brief description of XML and its characteristics. XML, *Extensible Markup Language* [3] is a markup language that has become the de-facto standard for data exchange over the web. This is due to the high flexibility of XML, that makes it able to model the various kind of data format that are present over the web: HTML data, relational and object database data, structured and unstructured textual data, audio and video data, and so on as described in [13]. There is no fixed collection of markup tags in XML that let us define our own tags and structure, tailored for the kind of information that we wish to present. The XML specification says nothing about the semantics of the markups tags, this is left to the specifications of the particular XML application.

2.1.1 Textual Representation

An XML Document is written as a Unicode text with markup tags and other meta-information representing the elements, attributes, and other nodes.

Text nodes are simply written as the text they represent. This text is also called *character data*. Element nodes are denoted by markup tags as in the following example:

```
<book  year = '1998'>
  <title>
    Types and Programming Languages
  </title>
  <author>
    B.Pierce
  </author>
  <hardcover/>
</book>
```

Here, <title> is an element *start tag* with name `title`, and </title> is a matching *end tag*. The text between, which generally may contain markup in turn, is the content of the element. Attributes are written inside the element start tag. In this case, the element has a single attribute named `year` with value `1998`. Within a start tag, the ordering of attributes and all whitespace surrounding them are always insignificant. Attribute values are enclosed by ''or "". Elements without content are called *empty*, and such elements may be written with a short-hand notation <hardcover/> as an alternative to <hardcover></hardcover>.

2.1.2 Well-formedness

An XML document in its textual form must be *well-formed*. This essentially means that it defines a tree structure, corresponding to the conceptual model presented earlier. To be well formed tags must match and nest properly. The following fragments are not well formed.

- <book> </title>
- <book> <title> </book> </title>

In the first, the start and the end tags do not match; in the second `book` and `title` does not nest properly. Moreover XML is case sensitive. For example,

```
<title>
  Types and Programming Languages
</TiTlE>
```

is *not* well formed. A well-formed document must have exactly one root element. Since a non-well-formed document has no meaning, we always implicitly mean *well formed XML document* when we say *XML document*, unless otherwise stated.

2.1.3 Tree Representation

All the XML Data Tree are modeled with a tree structure. Conceptually, an XML document is a hierarchical structure called *XML tree*, which consists of *nodes* of various kind ordered and arranged as a tree, as in Figure (2.1). We

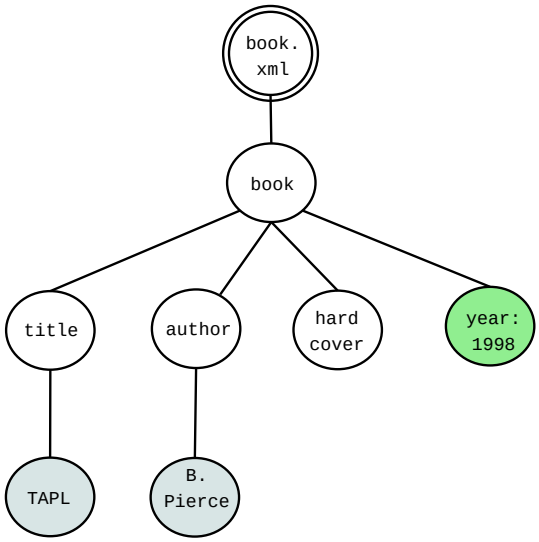

Figure 2.1: Tree representation of the document

can distinguish various kind of nodes: text nodes, element nodes, attribute nodes, comment nodes, processing instruction nodes and root nodes. In Figure (2.1) nodes are drawn as circles. The topmost node (marked with double border) is called the *root-node* and represents the entire document. The edges shows a parent-child relation between the nodes, for example, the node with label `title` is *child* of the node with label `book`, and viceversa the second is the *parent* of the first. The *content* of each node is the sequence of its child nodes. Nodes with no children are called *leaves* such as the text nodes or the attribute nodes (respectively on the left and on the right of the figure). We miss processing instruction nodes.

Every element node has a *name*, also called *label*, a word that describes the grouping and that corresponds to an *element-tag*. This is strictly related

with typing which describes the element-tags that we can find as children of each element-tag. A *type* is the element-tag of a node and the structure of its content. Later in this dissertation we will refer to *type* meaning one approximation of it: the element-tag and the name contained in its content model (we will not consider completely the structure of the content model). Element nodes are the most important to the aim of optimization because they define the structure of the document. We know that also attribute nodes can give optimization, based on non-structural condition, but to the sake of simplicity of the formal data model they are not considered in this work, as we will see later. Another thing to specify is about the root-node of the tree. The children of the root-node consist of any number of comment and processing instruction nodes together with *exactly one* element node, which is called the *root-element* (a common mistake is to confuse the root node with the root element of a document). For the sake of simplicity in our model we will refer always to the root-element while formally treating XML Trees. The *siblings* of a node are the other children of the parent of the node. The *ancestors* of a node consist of its parent, the parent of the parent, and so on, including the root node. The *descendants* of a node is the set consisting of its children, the children of the children, and so on.

Among all the nodes in an XML tree there is a total ordering called *document order*. Document order corresponds to the order in which the nodes appear if in textual representation. For example, in Figure (2.1) the node with label `author` appears after the node `title` and before the *text* node that contains the string `B.Pierce`. There's no order between attributes of a node. Document order between nodes in different XML trees is implementation-defined but must be consistent; that is, all the nodes in one tree must be ordered either before or after all the nodes in another tree.

2.1.4 XML Data model

Now we define the XML document data model we use in this document. The core of the formalization is once again the definition of node.

Definition 2.1 (Node) *A node is a pair* $\mathbf{n} = (\texttt{id}, X)$ *where* `id` *is an unique identifier and* X *is the type of the node.*

We use $\mathbf{n}_{\text{id}}, \mathbf{n}_X$ to indicate respectively the projection of the first and the second element of the pair. In all the approaches [1] , [2] and [6] the information about the type of the node is often demanded to an external function that maps every node into its type and other information. In this disserta-

tion we inglobe the type information in the datamodel instance. We refer the reader to the suggested literature for more details.

An instance of the XML document data model can be generated by the following grammar.

$$\begin{array}{rl} \textbf{Trees} & t ::= l_{\mathbf{n}}[f] \mid s_{\mathbf{n}} \\ \textbf{Forests} & f ::= t \mid f\,,\, f \mid () \end{array}$$

For example, for the tree in Figure (2.1) we have this production:

$$\texttt{book}_{\mathbf{n}_1}[\,\texttt{title}_{\mathbf{n}_2}[\,s_{\mathbf{n}_3}\,],\,\texttt{author}_{\mathbf{n}_4}[\,s_{\mathbf{n}_5}\,],\,\texttt{hardcover}_{\mathbf{n}_6}[\,()\,]\,]$$

Essentially, this grammar produces an ordered sequence of labeled ordered trees (ranged over by t), that is an ordered forest (ranged over by f) and where () denotes the empty forest. Tree nodes are labeled by *element tags* while, without loss of generality, we consider only leaves that are text nodes (that is strings ranged over by s) or empty trees (that is, elements that label the empty forest). As stated before, to model the content of each node we have based our definition on the notion of *forest*. This is essential for the model in order to be a good XQuery data model.

Definition 2.2 (Subtree Selection) *Given a forest f and a node $\mathbf{n}$ belonging to the forest, we write $f@\mathbf{n}$ to denote the unique subtree of f such that $f@\mathbf{n} = l_{\mathbf{n}}[f]$ or $f@\mathbf{n} = s_{\mathbf{n}}$.*

With abuse of notation we use $\mathbf{n}$ to identify the precise information given by $\mathbf{n}_{\mathtt{id}}$. Moreover given a forest f we can define the set of all the nodes of the forest as

$$\mathrm{Nodes}(f) = \{\mathbf{n} \mid f@\mathbf{n} = l_{\mathbf{n}}[f] \ \text{ or } \ f@\mathbf{n} = s_{\mathbf{n}}\}$$

and the set of identifiers of all the nodes of the forest

$$\mathrm{Ids}(f) = \{\mathbf{n}_{\mathrm{id}} | \mathbf{n} \in \mathrm{Nodes}(f)\}$$

Definition 2.3 (Good Formation) *A forest f is well formed if for every $\mathbf{n} \in \mathrm{Nodes}(f)$ we have that the identifier $\mathbf{n}_{\mathrm{id}}$ occurs in $\mathrm{Ids}(f)$ almost once.*

Henceforth we will consider only well-formed forests, and later we will define also good formation wrt types. It's also easy to see the following proposition.

Proposition 2.4 *Given a well formed forest f and a tree t, if $t \in f$ then t is a well formed tree.*

We recall that the XML document data model makes a difference between the *root-node*, which represents the entire document, and the *root-element* (the second is the unique element node child of the first). Despite this, in our abstract definition of data tree we define and we refer to the *root* of a tree as the root-element, as follows:

Definition 2.5 (Root Node) *Given a tree t, if $t = s_{\mathbf{n}}$ or $t = l_{\mathbf{n}}[f]$ then* $\text{rootNode}(t) = \mathbf{n}$.

2.2 Schema Languages and Validation

Now we proceed defining the type language model. A schema, at the matter of facts, is a grammar that defines the structure of valid XML documents. From a theoretical point of view is interesting to note that grammars correspondent to DTD and XML Schema are two particular subclasses of the Regular Tree Grammars (RTG). So, we start introducing the class of tree grammars called *regular*, then we proceed with two restricted classes called *local* and *single type* that corresponds to DTD and XML Schema respectively.

2.2.1 Tree grammars

Tree Grammars define languages of terms representing finite labeled ordered trees. They are used because they provide a good compromise between abstraction, expressively and tractability for defining and reasoning about structural constraints over XML trees.

Regular Tree Grammars and Languages

We use the definitions in [16] where RTGs describe unranked trees (any node can have any number of children) which allows trees with infinite arity (any node can have any number of children), and the right-hand side of a production rule to have a regular expression over non-terminals generated by the following grammar:

$$r ::= \quad r|r \quad | \quad (r,r) \quad | \quad r^* \quad | \quad r^+ \quad | \quad r^? \quad | \quad Y \quad | \quad () \qquad (2.1)$$

where the operators "|"and ","are the union and the concatenation between two regular expressions, "?"defines optionality, "$*$"and "$+$"repetition or the regular expression respectively at least 0 or 1 times and Y denotes a non-terminal metavariable.

Definition 2.6 (Regular Tree Grammar) *A regular tree grammar (RTG) is a 4-uple* $G = (N, T, S, E)$ *where*

- N *is a finite set of* non-terminals
- T *is a finite set of* terminals
- S *is a set of* start symbols*, where* S *is a subset of* N
- E *is a finite set of production rules of the form* $Y \rightarrow l[r]$*, where* $Y \in N$*,* $l \in T$*, and* r *is a regular expression over* N *defined as in* (2.1)

Y *is the left-hand side,* $l[r]$ *is the right-hand side and* r *is the* content model *of this production rule.*

Example 2.7 The following grammar $G_1 = (N, T, S, E)$ is a regular tree grammar.

$$
\begin{array}{rcl}
N & = & \{\ \texttt{Book, Title, Editor, Author, Name}\ \} \\
T & = & \{\ \texttt{book, title, editor, author, name}, \mathit{string}\} \\
S & = & \{\ \texttt{Book}\ \} \\
E & = & \{\ \texttt{Book} \rightarrow \texttt{book}[\texttt{Title}, \texttt{Author}, \texttt{Editor}], \\
 & & \ \ \texttt{Title} \rightarrow \mathit{string}[\varepsilon],\ \texttt{Author} \rightarrow \texttt{author}[\texttt{Name}], \\
 & & \ \ \texttt{Editor} \rightarrow \texttt{editor}[\texttt{Name}],\ \texttt{Name} \rightarrow \mathit{string}[\varepsilon]\ \}
\end{array}
$$

we represent every text value by the node $\mathit{string}[\varepsilon]$ for convenience.

Without loss of generality we can assume that no two production rules have the same non-terminal in the left-hand side and the same terminal in the right-hand side at the same time. If a regular grammar contains such production rules, we only have to merge them into a single production rule. We also assume that every non-terminal is either a start symbol or occurs in the content model of some production rule (in other words, no non-terminal are useless).

Now we define how a regular tree grammar generates a set of trees over terminals. Classical approaches [16], [1] and [3] defines a function from a grammar to a tree called interpretation, that maps every node to a non-terminal. We no need to use this approach since in our enriched data model a node $\mathbf{n} = (\mathrm{id}, X)$ is coupled both with its identifier and its type.

Definition 2.8 (Interpretability) *Given a tree* t *and a regular tree grammar* $G = (N, T, S, P)$*, we say that* t *is interpretable wrt* G *if let* t' *a subtree of* t *the following conditions hold.*

1. *if* $t' = t$ *and* $\mathbf{n} = \text{rootNode}(t)$ *then* $\mathbf{n}_X \in S$.
2. *if* $t' = s_{\mathbf{n}}$ *then there exists a production rule* $Y \rightarrow \texttt{string}[\epsilon] \in P$ *and* $\mathbf{n}_X = Y$.
3. *if* $t' = l_{\mathbf{n}}[t_1, \ldots, t_p]$ *then there exists a production rule* $Y \rightarrow l[r] \in P$ *such that* $l \in T$, $\mathbf{n}_X = Y$ *and let* $\mathbf{n}^i = \text{rootNode}(t_i)$ *then* $\mathbf{n}^1_X, \ldots, \mathbf{n}^p_X$ *is generated by* r.

Example 2.9 *Given the grammar* G_1 *and the tree* t *as instance of the XML data model in* Figure (2.2)*, it's easy to see that for example*

- $\text{rootNode}(t) = \mathbf{n}_1$, $\mathbf{n}_{1X} = \texttt{Book}$ and $\texttt{Book} \in S$.
- $t = \texttt{book}_{\mathbf{n}_1}[t_1, t_2, t_3]$ and exists a production rule $\texttt{Book} \rightarrow \texttt{book}[r] \in P$ that generates the string t_1, t_2, t_3 since $r = (\texttt{Title}, \texttt{Author}, \texttt{Editor})$.
- $t@\mathbf{n}_3 = s_{\mathbf{n}_3}$ and there exists a production rule $\texttt{Title} \rightarrow \mathit{string}[\varepsilon] \in P$.

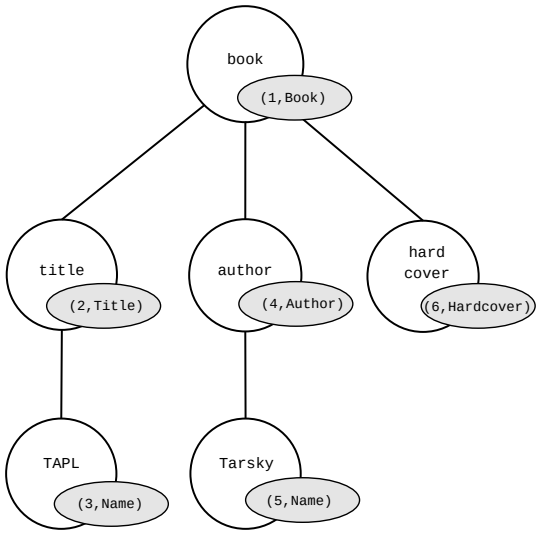

Figure 2.2: XML Data Model instance

Definition 2.10 (Generation) *Given a tree grammar* $G = (N, T, S, E)$*, a tree* t *is generated from* G *if it is* interpretable *wrt to* G.

Definition 2.11 (Regular Tree Language) *A regular language is the set of tree generated by a regular tree grammar.*

A classic result in the theory of this grammars is that for each regular tree grammars, a finite state automaton, usually called a *regular tree automaton*, can be defined so that it accepts a tree if and only if the tree is generated by the regular tree grammar, as stated in [3]. Other results says that algorithms for tree validity checking can also be defined without exploiting tree automata and just looking to regular tree grammar's productions. In particular, local and single-type tree grammars, that we're going to introduce, are two examples of classes of grammars for which recognizers are deterministic and work in a time which is polynomial in the number of subtrees of the input tree, and in the number of productions in the grammar. Local and single-type regular tree grammars are defined by imposing some restrictions on productions, so to allow deterministic choice of productions to apply when a tree is matched against a grammar during the recognition process. These restrictions are based on the notion of *competition between non terminals*.

Local Tree Grammars and Languages

We first define competition of non-terminals. Then, we have to introduce a restricted class called *local* by prohibiting competition of non-terminals. This class corresponds to DTD.

Definition 2.12 (Competing Non-Terminals) *Two different non-terminals A and B are said competing with each other if*

- *one production rule has A in the left-hand side*
- *one production rule has B in the left-hand side, and*
- *these two productions rules share the same terminal in the right hand side.*

Example 2.13 The following grammar $G_2 = (N, T, S, E)$ is a regular tree grammar.

$$
\begin{array}{lcl}
N & = & \{\ \texttt{Book, Author1, Author2, Name}\ \} \\
T & = & \{\ \texttt{book, author, name}, \mathit{string}\} \\
S & = & \{\ \texttt{Book}\} \\
E & = & \{\ \texttt{Book} \rightarrow \texttt{book}[\texttt{Title}, \texttt{Author1}, \texttt{Author2}], \\
 & & \ \ \texttt{Author1} \rightarrow \texttt{author}[\texttt{Name}], \\
 & & \ \ \texttt{Author2} \rightarrow \texttt{author}[\texttt{Name}], \\
 & & \ \ \texttt{Name} \rightarrow \mathit{string}[\varepsilon]\ \}
\end{array}
$$

`Author1` and `Author2` compete with each other, since the production rule for they share the same terminal `author` in the right-hand side. There are no other competing non-terminal pairs in this grammar.

Definition 2.14 (Local Tree Grammar and Language) *A local tree grammar is a regular tree grammar without competing non-terminals. A tree language is a local tree language if it is generated by a local tree grammar.*

Example 2.15 The following grammar $G_3 = (N, T, S, E)$ is a local tree grammar.

$$\begin{aligned} N &= \{\, \texttt{Book, Author1, Name, String} \,\} \\ T &= \{\, \texttt{book, author, name, pcdata} \,\} \\ S &= \{\, \texttt{Book} \,\} \\ E &= \{\, \texttt{Book} \rightarrow \texttt{book}[\texttt{Title}, \texttt{Author1}], \\ &\quad\ \texttt{Author1} \rightarrow \texttt{author}[\texttt{Name}], \\ &\quad\ \texttt{Name} \rightarrow \mathit{string}[\varepsilon] \,\} \end{aligned}$$

Single-Type Tree Grammars and Languages

To be complete, we introduce a less restricted class called *single type*, by prohibiting competition of non-terminals within a single content model. This class corresponds to XML Schema.

Definition 2.16 (Single-Type Tree Grammar and Language) *A single-type tree grammar is a regular tree grammar such that*

- *for each production rule, non-terminals in its content model do not compete with each other, and*
- *start symbols do not compete with each other.*

A tree language is a single-type language if it is generated by a single-type tree grammar.
The grammars G_1 and G_2 are not single type since `Author` and `Editor` compete and they are both in the content model of `Book` in the first and since `Author1` and `Author2` compete and they are both in the content model of `Book` in the second. The grammar G_3 is a single-type tree grammar since it's a local tree grammar.

Example 2.17 The following grammar $G_4 = (N, T, S, E)$ is a single-type tree grammar.

$$
\begin{array}{rcl}
N & = & \{\ \texttt{Book, Author, AuthorName, EditorName, Editor, Name}\ \} \\
T & = & \{\ \texttt{book, author, , editor, name, } \ \mathit{string}\} \\
S & = & \{\ \texttt{Book}\ \} \\
E & = & \{\ \texttt{Book} \rightarrow \texttt{book}[\texttt{Author}, \texttt{Editor}], \\
 & & \ \ \texttt{Author} \rightarrow \texttt{author}[\texttt{AuthorName}], \\
 & & \ \ \texttt{AuthorName} \rightarrow \texttt{name}[\texttt{Name}], \\
 & & \ \ \texttt{Editor} \rightarrow \texttt{editor}[\texttt{EditorName}], \\
 & & \ \ \texttt{EditorName} \rightarrow \texttt{name}[\texttt{Name}], \\
 & & \ \ \texttt{Name} \rightarrow \mathit{string}[\varepsilon]\ \}
\end{array}
$$

In this case `AuthorName` and `EditorName` compete, but they belongs to different content models.

As shown in Figure (2.3) single-type tree grammars are strictly more expressive than local tree grammars. That is, some single-type tree grammars cannot be written as local tree grammars. And this is consistent with the fact that XML Schema are more expressive than DTDs. Moreover regular tree grammars are more expressive than single-type tree grammars.

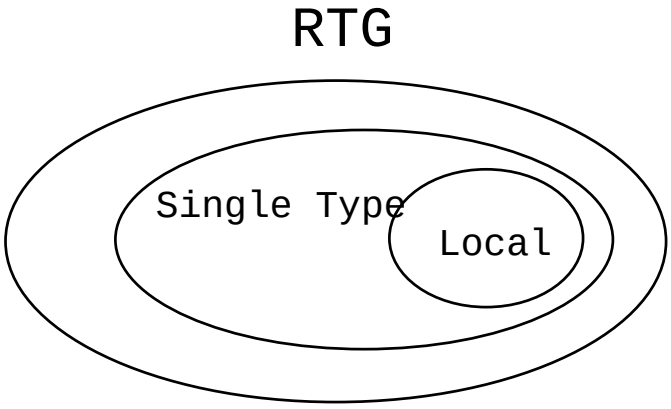

Figure 2.3: Geography of RTG Grammars

The choose of Regular Tree Grammars

An interesting question suggested in [16] may be why for XML documents are not used *context free (string) grammars*. Context free (string) grammars

represent sets of *strings*. Successful parsing of strings against such grammar provides derivation trees. This scenario is appropriate for programming languages and natural language, where programs and natural language text are strings rather than trees. On the other hand, start tags and end tags in an XML document collectively represent a *tree*. Since traditional context free (string) grammars are originally designed to describe permissible strings, they are inappropriate to describe permissible trees.

2.2.2 DTD and XML Schema

A schema is a formal definition of the syntax of an XML-based language, that is, it defines a family of XML documents. This is why typing is important to reason and optimize over the same family of documents. A *schema language* is a formal language for expressing schemas.

A schema language must satisfy three criteria to be useful. First it must provide sufficient expressiveness such that the most syntactic requirements can be formalized in the language. Second it must be possible to implement efficient schema processors, which works ideally in linear requirements both in time and space. Third the language must be understandable by non-experts to be used by a user different from the creator of the schema.

In this section we introduce Document Type Declaration (DTD) showing how it is related with regular tree grammars. DTD is a good compromise between simplicity and completeness of details to better understand the problem of document validation.

DTD

XML has since the first working draft contained a built-in schema language: Document Type Definition (DTD). Just as the XML notation is defined as a subset of SGML, the DTD part of XML is designed as the DTD part of SGML. We will refer always to the XML variant. Also, to avoid confusion, we will use the term *DTD schema* for referring to a particular schema written in the DTD language. From a language point of view we have to note that DTD is not itself written in XML notation.

A DTD schema consist of declaration of elements, attributes, and various other constructs. An *element declaration* is in the form

<!`ELEMENT` *element–name content–model*>

where *element–name* is an element name, such as `Book` in Example (2.7) and *content–model*s a regular expression over tags and text-symbols types

which defines the validity requirements of the contents of all elements of the given name. Every element name that occurs in the instance document must correspond to one element declaration in order for the document to be valid. Moreover, the contents of the element must match the associate content model. Content models come in four different flavours:

empty: the content model of an element is `EMPTY`, then the element must be empty. As previous explained, being empty means that it has no contents, but this says nothing about attributes.

```
<!ELEMENT hardcover EMPTY>
```

any: the content model can consist of any sequence of element and characters data. The keyword `ANY` is often used to model open contents, where element are not still defined. It's rarely used in final schemas.

```
<!ELEMENT undefinedField ANY>
```

mixed content: a content model of the form

$$(\#\texttt{PCDATA} \mid \texttt{title} \mid \texttt{author})*$$

where `title` and `author` are element names, means that the contents may contain arbitrary *parsable character data*, interspersed with any number of elements of the specified names.

element content: to specify constraints on the order and number of occurrences of child elements, a content model can be written using the grammar defined in 2.1.

For the sake of simplicity we miss the treatment of the other characteristic such as attribute declaration, and we refer to W3C literature since they do not concern our work.

Three questions about DTD language

As suggested in [13] there are interesting questions about the design of the DTD language. First either arbitrary character data is permitted in the contents or no character data is permitted. Why we are not allowed to impose constraints on character data, for example, such that only whitespace and digits are allowed in the contents of certain elements? Second, if character data is to be permitted in the contents, then we have no choice but using

the mixed content model, which cannot constrain the order and number of occurrences of the child elements. Why is not possible to use character data together with the element content model? This question is related to a tradition of roughly classifying SGML and XML languages as either *document oriented* or *data oriented* depending on whether they use the mixed content model or not, but such not an artificial classification not be necessary. Third why do we need the restriction to deterministic content models? After all it has been know for more than thirty years how to perform efficient pattern matching on general regular expressions. Clearly, these limitations become practical problems in the real world. The simple answer to these questions is that a design goal was that XML (and hence also DTD) should be a subset of the SGML language, which has these unfortunate properties. From another point of view the identification of these limitations has motivated the design of alternative schema languages.

From DTD to Local Regular Tree Grammars and back

Grammars generated as DTD schemas corresponds to local regular tree grammars. This is, because content model declarations are tag coupled, and this forbids the definition of different element types with same label. In terms of regular tree grammars, this correspond to not competing non-terminal symbols. We can build our local tree grammar $G = (N, T, S, P)$ starting from DTD and viceversa. We just consider the set of terminal symbols T as equal to the set of tags in the DTD. Under this assumption, we can see each declaration in the DTD having the form

$$<!\texttt{ELEMENT}\ l\ (\ r\ )>$$

where $l \in T$ and r is a regular expression over T. Then, we can define non-terminals N and build productions P as follows: consider a symbol $Y \in N$, for each $l \in T$ and for each DTD declaration $<!\texttt{ELEMENT}\ l\ (\ r\ )>$ provide a production $Y \rightarrow l[r]$ where R correspond to the content model r where (i) each type occurrence is replaced with its correspondent non-terminal in N and (ii) `EMPTY` is replaced with the regular expression ϵ.

Example 2.18 *Correspondence between DTD schema and Local RTG for the DTD.*

```
<!DOCTYPE book<[
 <!ELEMENT book(title, (author+|editor+)+)>
 <!ELEMENT title(#PCDATA)>
 <!ELEMENT author(name)>
 <!ELEMENT editor(name)>
 <!ELEMENT name(#PCDATA)>
 ]>
```

$$
\begin{array}{lcl}
N & = & \{\texttt{Book, Title, Editor, Author, Name}\} \\
T & = & \{\texttt{book, title, editor, author, name}, \textit{string}\} \\
S & = & \{\texttt{Book }\} \\
E & = & \{\ \texttt{Book} \rightarrow \texttt{book}[\texttt{Title}, (\texttt{Author}^+|\texttt{Editor}^+)^+], \\
 & & \quad \texttt{Title} \rightarrow \textit{string}[\varepsilon],\ \texttt{Author} \rightarrow \texttt{author}[\texttt{Name}], \\
 & & \quad \texttt{Editor} \rightarrow \texttt{editor}[\texttt{Name}],\ \texttt{Name} \rightarrow \textit{string}[\varepsilon]\ \}
\end{array}
$$

As synthesis of this correspondence between the two DTD schemas and Local RTG, we can abstract and give the formal definition of DTD. The definition recall the ones of the Local RTG but is simplified inferring the sets N and T from E.

Definition 2.19 (DTD schema) *A DTD schema is a pair (W, E) where W is the distinguished type of the root and E is the set of productions of the form $\{Y_1 \rightarrow R_1, \ldots, Y_n \rightarrow R_n\}$ such that*

1. the Y_i's are pairwise distinct
2. each R_i is of the form $a_i[r_i]$ or *String* where a_i is an element tag, and each r_i is a regular expression over names $\{Y_1, \ldots, Y_n\}$
3. for each pair $Y_i \rightarrow a_i[r_i]$ and $Y_j \rightarrow a_j[r_j]$, $i = j$ if and only if $a_i = a_j$
4. $W \in \{Y_1, \ldots, Y_n\}$

In particular, condition 3 states that two type-names cannot share the same non-terminal in the right-hand side of the production (this is non competition). In the following we write Names(r) for the set of all names used in r and $DN(E)$ for the set of all names defined in E (that is, $\{Y_1, .., Y_n\}$ which corresponds to the set N into a RTG). We also say that r is a regular expression over (W, E), if r is a regular expression over names in $DN(E)$ and is defined as in (2.1).

2.2.3 Validation with DTD

The process of document validation take in input an XML document d, and a schema s, which is written in that particular schema language, and then checks whether d is or not syntactically correct according to s. If it is the case then we say that d is *valid* wrt to s. If d is invalid, then most schema processors produce useful diagnostic error messages.

Based on what we see before for interpretability of tree wrt to Regular Tree Grammars we can give the definition on validity for XML Documents. If an XML Document is valid wrt a DTD-instance then it has a tree representation which is interpretable wrt a Local RTG.

Definition 2.20 (Valid Trees) *A tree t is valid wrt a DTD (W, E) if the following conditions hold.*

1. if $\mathbf{n} = \text{rootNode}(t)$ then $\mathbf{n}_X = W$
2. for each $\mathbf{n} \in \text{Nodes}(t)$ if $t@\mathbf{n} = s_{\mathbf{n}}$ and $\mathbf{n}_X = Y$ then $Y \to String \in E$
3. for each $\mathbf{n} \in \text{Nodes}(t)$, if $t@\mathbf{n} = l_{\mathbf{n}}[t_1, \ldots, t_m]$ then $\exists\ Z \to l[r]$ where $Z = \mathbf{n}_X$ and let $\mathbf{n}_i = \text{rootNode}(t_i)$ then $(\mathbf{n}_{1X}, \ldots, \mathbf{n}_{mX})$ is generated by r.

In this case we say that t is valid wrt the DTD and we write $t \in_X (W, E)$ to indicate it.

2.2.4 Validation with XML Schema

While DTD comes from the document community as a language to constrain the format of SGML documents, XML Schema is closer to the spirit of DBMS and general programming language type systems.
Non competition between non-terminal in RTG grammars induces locality, that is, the fact that for each document element almost one production can generate it. This allows also a direct and efficient document validation. For what concerns expressivity of data, locality is a limit of DTD since it disallows modularization in type definitions: types are bounded to tag and not to names, hence type cannot be referred in other declarations. Moreover locality implies that element with the same tag but different content model cannot be described. This is a problem in mapping relational data into XML data, for instance, thinking to the fact that two distinct relations can have the same field-name with different type. Single Type RTG allows

competition between non-terminal metavariable that does not belong to the same content model. It's easy to see the correspondence between Single Type RTG and grammars generated by XML Schema. A further limitation of DTD is the lack of a notion of type and range specifications for simple value such as integers, real and so on. Despite all those limitations, to exploit the problem of validation, DTDs seems a good compromise between simplicity and completeness of details. This is, the most of the readers know DTD rather than XML Schema. So, we do not present in detail XML Schema and refer the reader to the suggested literature [9], [16]. Moreover, all the results of this work holds for the entire class of RTG, and then also for grammars generated either from DTD or XML Schema since it's known that they are generated by two subclasses.

Chapter 3

Querying XML

This chapter introduces FLWR-XQuery the subset of XQuery that we consider as query language, and upon which we base our work. The chapter enucleates the constructs that compose FLWR-XQuery. We provide both syntax and semantics of the language.

3.1 XQuery and FLWR-XQuery

XQuery is a query language defined by the XML Query Working Group in [7]. The design of XQuery has been subject to a number of influences: from the compatibility with existing W3C standards XML Schema, XSLT, XPath and XML [cite], to the experiences in developing other query languages such as Quilt, XQL, XML-QL, SQL and OQL [3].

From a language theory point of view is interesting the fact that XQuery is a Turing complete typed functional language. This means that the evaluation of each expression return a value with no side effects, that a queried data may be typed, and then a type for the query result can be inferred.

The W3C working group has published a list of requirements, a description of the datamodels that underlies the language, a formal semantics description, a list of functions and operators, and a collection of use cases that illustrate applications of the language in [7],[8], [10] and [11] . Despite this rich documentation there's a lack in formality: the semantics of the language, although well defined, is not proposed in a concise manner, and it's unwieldy for proving certain properties. This is due to the large amount of details to consider in a rich language such as XQuery. It follows, as we see in literature [1], [3], [4], [5], [6], [20], [28] and [29] that, in order to work on a specific topic, it's often needed first a subset of the language that capture

the most interesting constructs which the problem is related to, and second a set of side assumptions to simplify the formal treatment of the problem.

That said, we propose a sublanguage of XQuery, that we dub FLWR-XQuery, that has almost the same expressive power as XQuery, has a readable syntax and semantics and is flexible enough to formal proof. For what concerns side assumptions, we choose to maintain some important characteristics of the language that often are missed in other works such as *ordered result* and *duplicate removals*. We specify FLWR-XQuery so that all syntactically valid expressions also satisfy XQuery syntax, and the result of a query evaluated using the proposed semantics will be exactly that evaluated by XQuery.

3.1.1 FLWR-XQuery Data model

The query data model provides an abstract representation of one or more XML documents or document fragments. The data model is based on the notion of *sequence* of nodes.

Definition 3.1 (Sequence) *A sequence of nodes is denoted by* $\vec{\mathbf{n}}$ *and is a vector of nodes* $(\mathbf{n}_1, \ldots, \mathbf{n}_p)$ *of length* p*.*

Given two sequences we denote $\vec{\mathbf{n}}, \vec{\mathbf{m}}$ for the concatenated sequence and we define

$$\vec{\mathbf{n}}, \vec{\mathbf{m}} = (\mathbf{n}_1, \ldots, \mathbf{n}_p, \mathbf{m}_1, \ldots, \mathbf{m}_q) \tag{3.1}$$

We denote with $|\mathbf{n}|$ the cardinality of the sequence, so in this case $|\mathbf{n}| = p$. The empty sequence is neutral wrt concatenations: $\vec{\mathbf{n}}, () = \vec{\mathbf{n}}$.

The equality between sequences we define is approximated. We need an approximated definition because evaluations of a query over both a document and the same pruned document can yield two sequences of nodes that actually represents the same result but differs in `id`. This is, because a pruned document can use a different enumeration of the ids of the nodes wrt the original document. In order to define what stated before, we need an approximated notion of equality between nodes. This notion is based on the fact that equal nodes induces the same subtrees of the forest they belong, if we do not consider ids. To do not consider the information related to the `id` of a node, given a tree t we dub $\overline{t}$ for the tree t where for each node $\mathbf{n} \in \text{Nodes}(t)$ we erase the node identifier information imposing $\mathbf{n}_{\texttt{id}} = \perp$. Then, given two nodes $\mathbf{n}$, $\mathbf{m} \in f$ we have that

$$\mathbf{n} \cong \mathbf{m} \quad \text{iff} \quad \overline{f@\mathbf{n}} = \overline{f@\mathbf{m}} \tag{3.2}$$

At the matter of facts, we consider as equal nodes with same type that are roots of the same subtree.
Now it's easy to define equality between sequences of nodes:

$$(\mathbf{n}_1, \ldots, \mathbf{n}_p) \cong (\mathbf{m}_1, \ldots, \mathbf{m}_p) \quad \text{iff} \quad \forall\, i = 1..p \;\; \mathbf{n}_i \cong \mathbf{m}_i \tag{3.3}$$

Finally, to assign at sequence $\vec{\mathbf{m}}$ as value of the sequence $\vec{\mathbf{n}}$ we write $\vec{\mathbf{n}} \leftarrow \vec{\mathbf{m}}$.

A sequence as is an ordered collection of zero or more nodes used as persistent roots of subtrees of the data. Nodes in a data model instance have a unique identity, established when the node is created, and may have other nodes as children, thus forming a node hierarchy reflecting XML tree structure. Nodes belonging to the same hierarchy are ordered accounting to document order. Ordering among attributes, and among nodes belonging to different documents is leaved as implementation defined. The only constraint is that has to be respected is that all the nodes of a tree must be ordered either before or after the nodes of another tree.

Sequences never appear as an item in another sequence. All operations that create sequences are defined to flatten their operands so that the result of the operation is a single level sequence. There's no distinction between a node and a sequence of length one, in other words, a node is considered identical to a sequence of length one. Empty sequences are allowed and are sometimes used to represent missing on unknown information, in much the same way that null values are used in relational systems. When a sequence of nodes is the evaluation of a query, the contained nodes are persistent roots of subtrees of the document, that are converted into an XML document by a process called *serialization*.

Example 3.2 *To illustrate the query data model, and provide basis for later examples, we consider the following DTD and XML database that contains a bibliography document based on the Use Cases proposed by W3C in [11].*

```
<?xmlversion = "1.0"?>
<!DOCTYPE bib[
 <!ELEMENT bib (book*)>
 <!ELEMENT book (title, (author + |editor+), publisher, price)>
 <!ELEMENT author (last, first)>
 <!ELEMENT editor (last, first, affiliation)>
 <!ELEMENT title (#PCDATA)>
 <!ELEMENT last (#PCDATA)>
 <!ELEMENT first (#PCDATA)>
 <!ELEMENT affiliation (#PCDATA)>
 <!ELEMENT publisher (#PCDATA)>
 <!ELEMENT price (#PCDATA)>
  ]>

<bib>
  <book>
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison − Wesley</publisher>
    <price>65.95</price>
  </book>

  <book>
    <title>Advanced Programming in the Unix environment</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison − Wesley</publisher>
    <price>65.95</price>
  </book>

  <book>
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
  </book>

</bib>
```

3.1.2 Expressions

FLWR-XQuery as XQuery has several kinds of expressions, most of which are composed by lower-level expressions, combined by operators or keywords. All the expressions are fully composable, that is, where an expression is expected any kind of expression may be used.

3.1.3 Path Expressions

Path expressions in the language we considered are based on the syntax of XPath. A path expression consists of a series of steps, separated by the slash character ("/"). Each Step is in the form Axis::Test, where Axis navigates the tree to select some nodes and Test is a filtering condition on the selected nodes. The result of each step is a set of nodes. The value of the path expression is the node set that returns from the last step in the path.

$$\mathsf{Path} = \mathsf{Step}_1/\ldots/\mathsf{Step}_n$$

Each step is evaluated wrt a particular node called *context node*. During path evaluation, the nodes selected by each step serve in turn as context nodes for the following step. If a step has several context nodes, it is evaluated *for each* of the context nodes in turn, and the resulting node sequences are concatenated to form the result of the step. The result of a path is always a set of nodes that must be organized as a sequence in document order.

From a syntax point of vier XPath expressions may be written in either unabbreviated syntax or in abbreviate syntax. The unabbreviated syntax for an axis step consists of an axis and a selection criterion, separated by two colons as in Query (3.1).

Query 3.1 *List of titles of books titles*

```
document("bib.XML")/child :: bib/
/child :: book/child :: title
```

The first step invokes the built-in `document` function, which returns the document node for the document named `bib.XML` of Example (3.2). The second step is an axis step that finds all the children of the document node (`child :: bib` and selects a single element node named `bib`, that is the root-element). The third step follows the `child` axis again to find all the child elements at the next level that are named `book` and that in turn have nested the element `title`. The final result of the path expressions is the result of

the third step: a sequence of `title` element nodes in document order, that after serialization appears as:

```
<title>TCP/IP Illustrated</title>
<title>Advanced Programming in the Unix environment</title>
<title>Data on the Web</title>
```

In practice, path expressions are usually written in abbreviated syntax. Several kinds of abbreviations are provided. Perhaps the most important of these is that the axis specifier may be omitted when the `child` axis is used. Since child is the most commonly used axis, this abbreviation is helpful in reducing the length of many path expressions. For example, Query (3.1) may be abbreviated as:

```
/bib/book/title
```

As in the above example, and from now on, we leave as implicit the builtin `document` function and assume that the root-element is the only one in context of the root "/". When two steps are separated by a double slash ("//") rather than by a single slash, it means that the second step may traverse multiple levels of the hierarchy, using the descendants axis rather than the single-level child axis (a more precise semantics will be described later), for example:

Query 3.2 *List all* `title` *elements found in the document* `bib.XML`.

```
//title
```

Query (3.2) which is equivalent to /`descendant–or–self` :: `title` searches for description elements that are descendants of the root node of a given document. Query (3.2) evaluates to a sequence of element nodes that could, in principle, have been found at various levels of the node hierarchy (though, in our sample document, all the elements are at the same level). The final evaluation is the same as in Query (3.1).

Axes

An axis is a sequence of nodes located relative to the context node. It is a first approximation to the sequence that we wish to obtain as the result of a location step because often is completed by a filtering on the selected nodes. FLWR-XQuery supports all the navigational axis of XPath, and especially, one things that give originality to our work is the treatment of all

the *horizontal* axes: `following–sibling`, `following`, `preceding–sibling` and `preceding` not directly supported in [1][2]. The remaining axes are, intuitively, called *vertical* axes. Follows a brief description of each kind of axis.

- `self`, the context node itself.
- `child`, the children of the context node.
- `descendant`, the descendants of the context node.
- `parent`, the unique parent node of the context node, but empty sequence if the content node is the root node.
- `ancestor`, all the ancestors of the context node, from the parent to the root.
- `following–sibling`, the right-hand siblings of the context node.
- `preceding–sibling`, the left-hand siblings of the context node.
- `following`, all nodes appearing strictly after in the document that the context node, but excluding descendants.
- `preceding`, all nodes appearing strictly before the context node, but excluding ancestors.
- `descendant–or–self`, the descendants of the context node and the context node.
- `ancestor–or–self`, all the ancestors of the context node, from the parent to the root, and the context node.

The main axes are illustrated in Figure(3.1). The `attribute` axis is obviously missed as explained before. Each axis has a direction, which determines the order in which the nodes are assigned positions in the sequence. Another important distinction is made between *forward* axes and *backward* axes. The forward axis are `self`, `child`, `descendant`, `following–sibling`, `following`, and the backward axes are `parent`, `ancestor`, `preceding–sibling` and `preceding`. For the sake of simplicity, during the dissertation, we omit

[2] As said in the introduction this axes are approximated.

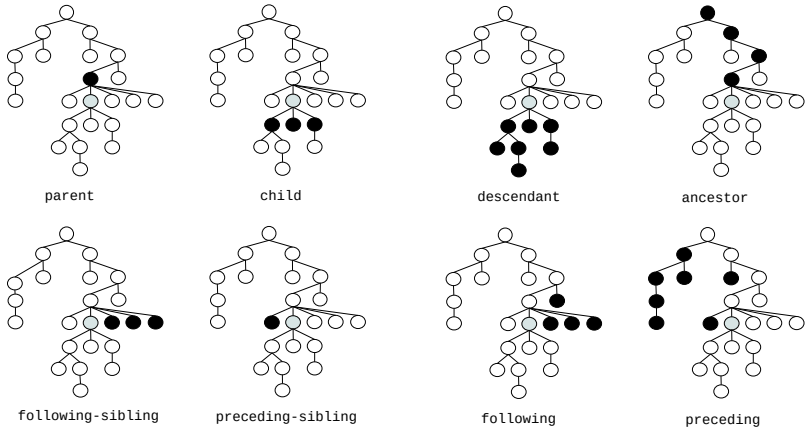

Figure 3.1: Navigational Axes

the treatment of `descendant–or–self` and `ancestor–or–self` axis since they are the composition of two described axis:

$$\texttt{descendant–or–self} :: \texttt{node} \approx \texttt{descendant} :: \texttt{node}\,,\ \texttt{self} :: \texttt{node} \quad (3.4)$$

and

$$\texttt{ancestor–or–self} :: \texttt{node} \approx \texttt{ancestor} :: \texttt{node}\,,\ \texttt{self} :: \texttt{node} \quad (3.5)$$

We are interested in cover formally all XPath. Horizontal axes are powerful since they allow to navigate the entire tree. In [12] is possible to find a series of examples where their use is more intuitive that the use of other axes. This is, especially if data is know as ordered with some criteria, for example, considering the DTD in Example (3.2) where we assume exists an alphabetical order over books title, as in the following two examples.

Query 3.3 *We use axis* `following–sibling` *to select all* `book` *element that follows the first element with* `title` *equal to* "`TCP/IPillustrated`".

$$/\texttt{bib}/\texttt{book}[\texttt{title} = \text{"}\texttt{TCP/IPillustrated}\text{"}]/\texttt{following–sibling} :: \texttt{node}$$

Which evaluates to:

```
<bookyear = "1992">
  <title>Advanced Programming in the Unix environment</title>
  <author><last>Stevens</last><first>W.</first></author>
  <publisher>Addison − Wesley</publisher>
  <price>65.95</price>
</book>

<bookyear = "2000">
  <title>Data on the Web</title>
  <author><last>Abiteboul</last><first>Serge</first></author>
  <author><last>Buneman</last><first>Peter</first></author>
  <author><last>Suciu</last><first>Dan</first></author>
  <publisher>Morgan Kaufmann Publishers</publisher>
  <price>39.95</price>
</book>
```

Query 3.4 *We use axis* `preceding` *to select all the books before those which author* `last` *name is* "`Suciu`"*:*

$$//\texttt{self} :: \texttt{text} = \text{"Suciu"}/\texttt{preceding} :: \texttt{node}[\texttt{book}]$$

Which evaluates to:

```
<bookyear = "1994">
  <title>TCP/IP Illustrated</title>
  <author><last>Stevens</last><first>W.</first></author>
  <publisher>Addison − Wesley</publisher>
  <price>65.95</price>
</book>

<bookyear = "1992">
  <title>Advanced Programming in the Unix environment</title>
  <author><last>Stevens</last><first>W.</first></author>
  <publisher>Addison − Wesley</publisher>
  <price>65.95</price>
</book>
```

Predicates

A predicate is an expression, enclosed in square brackets with the form

$$\mathsf{Step}[\mathsf{Cond}]$$

that use the condition Cond to filter a sequence of values, for example:

$$//\texttt{book}[\texttt{child} :: \texttt{editor}]/\texttt{title}$$

In the step `book`[`child` :: `author`], the phrase `child` :: `author` is a predicate that is used to select certain `book` nodes and discard others (notice that the semantics is another time the list of elements). If the predicate expression evaluates to an empty sequence, the candidate node is discarded, but if the predicate expression evaluates to a sequence containing at least one node (i.e. the book contains an `editor` element), the candidate element is selected. This form of predicate can be used to test for the existence of a child node that satisfies some condition.

We impose some restrictions on conditions. Predicates that use equality such as `book`[`author`/`last` = "`Stevens`"] are approximated by eliminating value-based equality. Moreover, we consider only *disjunction* of single conditions in the form $\mathsf{Cond}_1 \vee \ldots \vee \mathsf{Cond}_n$. So, every query that use *conjunctions* of single conditions or equality on predicates such as the follows

$$[(//\texttt{last} = \text{"}\texttt{Stevens}\text{"}\ \texttt{and}\ /\texttt{publisher} = \text{"}\texttt{A.W.}\text{"})\ \texttt{or}\ /\texttt{price} < \text{"}\texttt{50}\text{"}]$$

is approximated in our system with

$$[//\texttt{last}\ \texttt{or}\ /\texttt{publisher}\ \texttt{or}\ /\texttt{price}]$$

It's easy to see that all those approximations are sound. Another important class of predicates are those who use the negation of a condition that is a non pure structural condition. The main problem is that when we negate a condition is difficult to achieve completeness. Then to inject a negation of a a condition expression in our language we approximate it as follows:

$$\neg\mathsf{Cond} \quad = \quad (\,\mathsf{Cond}\ \text{or}\ \texttt{self}::\texttt{node}\,)$$

so we give enough structural information to perform a sound inference since `self`::`node` takes all the nodes at the first sublevel, but nothing can stated about completeness.

Projection based on structural condition is enough to evaluate soundly the semantic of a query with optimization as showed by experimental results in [1]. The set of conditions, which are ranged over by Cond, considers: the constants `true` and `false`, the use of a sub-query `q` (that doesn't constructs new elements) to verify the existence test for the condition, the disjunction of two conditions $\mathsf{Cond}_1 \vee \mathsf{Cond}_2$, the negation of a condition and the test $\ominus(\mathtt{x})$ that checks if a variable is binder to an empty sequence.

FLWR expressions

Iteration is an important part of a query language. XQuery provides a way to iterate over a sequence of values, binding a variable to each of the values in turn and evaluating an expression for each binding of the variable. The simplest form of iteration in XQuery consists of a `for` clause generated by the following production of the grammar of the language

$$\mathtt{for\ x\ in\ q_1\ return\ q_2}$$

that names a variable and provides a sequence of values to iterate over, followed by a `return` clause that contains the expression to be evaluated for each variable binding. The following example illustrates iteration:

$$\mathtt{for\ x\ in}\ (2,3)\ \mathtt{return\ x} + 1$$

The result of this simple iterative expression is the sequence $(3, 4)$.

A for clause may specify more than one variable, with an iteration sequence for each variable. Such a for clause produces tuples of variable bindings that form the Cartesian product of the iteration sequences. Unless otherwise specified, the binding tuples are generated in an order that preserves the order of the iteration sequences, using the rightmost variable as the *outer loop* and the rightmost variable as the *inner loop*. The following example illustrates a for clause that contains two variables and two iteration sequences:

```
for x in (2,3)
return
  for y in (5,10)
  return
    <fact> x times y  is  x * y </fact>
```

The result of this expression is the following sequence of four elements:

```
<fact> 2 times 5 is 10</fact>
<fact> 2 times 10 is 20</fact>
<fact> 3 times 5 is 15</fact>
<fact> 3 times 10 is 30</fact>
```

Those examples has been useful to clarify the semantics of the `for` construct, but they do not involve queries belonging to FLWR-XQuery because *literals* (i.e. atomic values such as integers) are not treated in the language we consider.

A variable may be bound to a value and used in an expression to represent that value. One way to bind a variable is by means of a `let` expression, which binds one or more variables and then evaluates an inner expression.

```
let x = q return q
```

The value of the `let` expression is the result of evaluating the inner expression with the variables bound. The following example illustrates the case.

```
let $x := 1, 2, 3
return 0, $x, 4
```

returns the sequence $0, 1, 2, 3, 4$.

As already seen, the purpose of the `for` clause and `let` clause is to bind variables. Each of these clauses contains one variable and an expression associated with each variable. The binding tuples produced by the `for` clauses and `let` clauses in XQuery can be filtered by the optional `where` clause. The where clause contains an expression that is evaluated for each binding tuple. If the value of the where expression is the Boolean value true or a sequence containing at least one node (an *existence test*), the binding tuple is retained; otherwise the binding tuple is discarded. The return clauses then executed once for each binding tuple retained by the where clause, in order. The results of these executions are concatenated into a sequence that serves as result.

```
for x in (2, 3)
return
  for y in (5, 10)
  where x = 3
  return
    <fact> x times y  is  x * y </fact>
```

The result of this expression is the following sequence of two elements:

```
<fact> 3 times 5 is 15</fact>
<fact> 3 times 10 is 30</fact>
```

In our work we decided to substitute the `where` with the `if` construct as we will see later. In what follows we give an example of how the above query with a `where` clause is rewrite in a query with an `if` clause.

```
for x in (2, 3)
return
  for y in (5, 10)
  return
  if x = 3 then
    <fact> x times y  is  x * y </fact>
```

Any resulting sequence can be reordered by a `order by` clause that contains one or more ordering expressions but this is not considered in FLWR-XQuery. The `for`, `let`, `where`, `return` and `order by` clause are both special cases of a more general expression called FLOWR expression, that gives the name to the language we consider. Since we do not semantically consider only `order by` it's meaningful to give the name FLWR-XQuery to the language.

Conditional expressions

A conditional expression in FLWR-XQuery provides a way of executing one expressions or less depending on the value of a second expression. It is written in the following form:

`if` Cond `then` q

the construct is simplified because is missed the classical `else` branch. The main construct `if` Cond `then` q_1 `else` q_2 can be soundly approximated via $\neg$Cond. The result of a conditional expression depends on the value of the expression in the `if` clause, called the *test expression*. We do not deal with primitive types such as Boolean values, then the rules are as follows: if the value of the test expression is a sequence containing at least one node (serving as an "existence test"), then the clause is executed. If the value of the test expression is an empty sequence, then nothing is executed. In particular, later defining the semantics of the language we will bind to the keyword `true` the value of the current context and with the keyword `false` the empty sequence.

Query 3.5 *List of all title of books printed with hardcover.*

```
for x in document("bib.XML")//book
return
  if x/hardcover
  then
    x/title
```

The evaluation of the Query (3.5) iterates over all the elements `book` and checks whether the navigation along the `child` axis for the `hardcover` attribute returns an existence node. If this is true then another navigation is performed to select the `title` of the node and add it to the result. We notice that the semantics of conditional expression is the same as the semantics of predicates.

Element constructors

Path expressions are powerful, but they have an important limitation: they can only select existing nodes. A full query language needs a facility to construct new elements and to specify their contents and relationships. This facility is provided in XQuery by a kind of expression called *element constructor* which is presents also in FLWR-XQuery in the following form

<a>q

in FLWR-XQuery element construction is allowed only in the rightmost part of the query, as a container for the forest selected by the rightmost expressions as in the following example:

Query 3.6 *List of all books with only the title element*

```
for x in //book
return
  <book>  x/title  </book>
```

The query evaluates to:

```
<book year = "1994">
  <title>TCP/IP Illustrated</title>
</book>

<book year = "1992">
  <title>Advanced Programming in the Unix environment</title>
</book>

<book year = "2000">
  <title>Data on the Web</title>
</book>
```

The content of an element constructor is a subexpression that may generates a sequence of nodes. The element nodes produced by a single evaluation of element constructor is a new node with its own node identity, which is the persistent root of the computed subexpression. If one of the newly constructed element has child nodes that are derived from existing nodes, as in the above example, the new child nodes and attributes are copies of the source nodes, in particular, with new identities, the same type and label or text. Since is difficult the typing of this structure, as solution, and without loss of generality, we always type nodes of element construction with the null type $\perp$. In the above evaluation are created three new distinguished instances of the datamodel, that are trees such that each root node has *label* `book` and *type* $\perp$. We give in Figure (3.2) the datamodel instances created as result of the query. A query that presents element construction in a

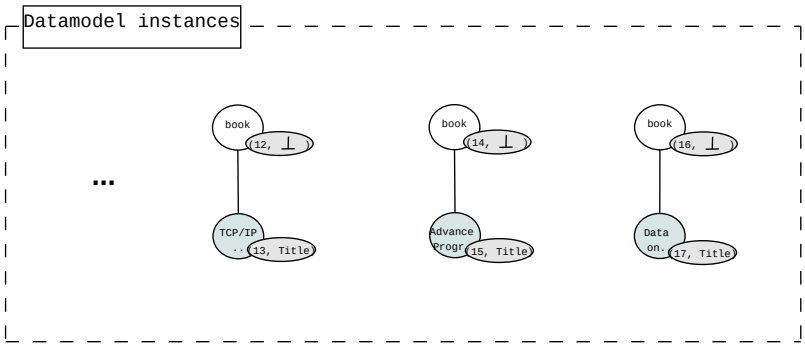

Figure 3.2: New builded data tree as result of Query (3.6)

non-rightmost position is called *ill-formed*.

Query 3.7 *Create a list of all author's names from a list of authors in the db.*

```
<namelist>
  for x in <authorslist>//author</authorslist>
  return
      <name> x/author/last/self :: text </name>
</namelist>
```

The query is ill-formed because use element construction in two *non-rightmost* query positions: at the top of the query and in the computation of the iteration sequence of the `for` construct.
This query is not allowed in FLWR-XQuery, but obviously can be evaluated in XQuery. During the evaluation of the query first of all is created a new instance of the data model that if would be serialized then it induces the following list of authors:

```
<authorslist>
  <author><last>Stevens</last><first>W.</first></author>
  <author><last>Stevens</last><first>W.</first></author>
  <author><last>Abiteboul</last><first>Serge</first></author>
  <author><last>Buneman</last><first>Peter</first></author>
  <author><last>Suciu</last><first>Dan</first></author>
</authorslist>
```

In the `return` clause the created list is navigated to compute the result as follows.

```
<namelist>
  <name>Stevens</name>
  <name>Stevens</name>
  <name>Abiteboul</name>
  <name>Buneman</name>
  <name>Suciu</name>
</namelist>
```

In this case, element construction build a new instance of the data model and navigates over it. No type-based static optimization (and no pruning) can or must be done in this case, because new elements are created with production not in the scope of the DTD. Then we forbid every operation, such as navigation, inside instances of the data-model builded at run time.

Then we need a weak notion of validity to deal with trees that created a run time and that can have nodes with empty type.

Definition 3.3 (Weakly valid tree) *A tree t is weakly valid with respect to a DTD (W, E) if given two node $\mathbf{n}$, $\mathbf{m}$ such that $\mathbf{n}$ is the parent of $\mathbf{m}$, if $\mathbf{n}_X, \mathbf{m}_X \in DN(E)$ then exists a production $Y \to l[r]$ such that $\mathbf{n}_X = Y$ and $\mathbf{m}_X \in \text{Names}(r)$*

A weakly valid tree is a particular data-structure that generalize the kinds of trees processed *during the query execution*. A weakly valid tree is either the input XML document that is supposed a valid tree, or a new tree builded during the computation by element-creation construct. The latter do not have to be pruned, because they are generated at run time. It's easy to see that the trees in Figure (3.2) are weakly valid trees, since the root types does not belongs to $DN(E)$.

Proposition 3.4 *If t is a valid tree then t is also a weakly valid tree.*

3.2 Syntax

We resume the syntax of FLWR-XQuery in table (3.1) as an abstract syntax, it assumes that extra brackets and precedence rules are added for disambiguation. The language handle XPath navigation, FLOWR expressions and element construction. We will range over query expressions with q. We use $\ominus(\mathtt{x})$ to check if the variable $\mathtt{x}$ is empty.

3.3 Semantic

We now proceed with the formal semantics of the language. All the expressions to be queried are evaluated against an *XML Store* η, which contains all the XML Data Model instances, and a *dynamical environment* ρ which contains variable bindings.

3.3.1 Preliminary Definitions

Definition 3.5 (Store) *We define a store η as a collection of trees.*

$$\eta = \{t_1, \ldots, t_n\}$$

The purpose of the store is to maintain the data trees that we are dealing with during query evaluation. In particular, at the top level of the store we have the input tree t that has to be pruned later and other trees that are created during the evaluation by element construction. All these trees are disjoints: if $\text{Nodes}(t_i) \cap \text{Nodes}(t_j) \neq \varnothing$ then $i = j$.

Table 3.1: *FLWR – XQuery Syntax*

q ::=	()	Path ::=	Step
	q, q		Step/Path
	<a>q		Step[Cond]
	x		Step[Cond]/Path
	x/Path		
	for x in q return q	Step ::=	Axis :: Test
	let x = q return q		
	if Cond then q	Axis ::=	self | child
Cond ::=	true		parent
	false		descendant
	q		ancestor
	⊖(x)		following
	Cond ∨ Cond		following–sibling
			preceding
			preceding–sibling
		Test ::=	tag
			node()
			text()

Definition 3.6 (Valid store) *We say that a store $\eta = \{t_1, \ldots, t_n\}$ is valid wrt a dtd (W, E) if the following conditions hold.*

1. *if $t \in \eta$ is the input tree, then t is a valid tree wrt the DTD (W, E).*
2. *if $t \in \eta$ is created during the computation, then t is a weakly valid tree wrt the DTD (W, E).*

The input tree is unique under our assumptions.

XQuery supports variables which can be bound using `let` or `for` expressions. Once a variable is bounded, it can be used in a subexpression. For example, consider the query:

```
for x in /bib/book
return x/title
```

During the evaluation, we use a dynamical environment to remember that the variable x has been bound to nodes resulting from the evaluation of the path `/bib/book` in order to apply further navigation steps.

Definition 3.7 (Dynamical Environment) *A dynamical environment is a set of bindings between variables and a sequences of nodes.*

$$\rho = \{\mathtt{x_1} \mapsto \vec{\mathbf{n}}_1, \ldots, \mathtt{x_n} \mapsto \vec{\mathbf{n}}_n\}$$

Every variable is bounded with a sequence of nodes. Given a variable $\mathtt{x_i}$ we need to access to the dynamical environment to take (if any) the sequence bounded to a variable, and we write $\rho(\mathtt{x_i}) = \vec{\mathbf{n}}_i$. Moreover we can either define a new variable or overwrite an existing one with $\rho[\mathtt{x_j} \mapsto \vec{\mathbf{n}}_j]$.

3.3.2 Navigational Axis

Navigational axis allows to visit the data tree. In order to define the semantics of all the axes we need to define a set of relation between nodes that belongs to the same tree. We start with the basic binary edge relation parent-child between nodes, then we define a relation for sibling nodes and finally a relation for following nodes.

Example 3.8 *The relation $\varepsilon^{\uparrow}_t$ over the tree in Figure (3.3) results as follows:*

$$\begin{aligned}\varepsilon^{\uparrow}_{t_1} \;=\; \{\;& (\mathbf{n}_1, \mathbf{n}_2),\\ & (\mathbf{n}_2, \mathbf{n}_3),\ (\mathbf{n}_2, \mathbf{n}_4),\ (\mathbf{n}_2, \mathbf{n}_7),\ (\mathbf{n}_2, \mathbf{n}_{10}),\ (\mathbf{n}_2, \mathbf{n}_{11}),\\ & (\mathbf{n}_4, \mathbf{n}_5),\ (\mathbf{n}_4, \mathbf{n}_6),\\ & (\mathbf{n}_7, \mathbf{n}_8),\ (\mathbf{n}_7, \mathbf{n}_9)\ \}\end{aligned}$$

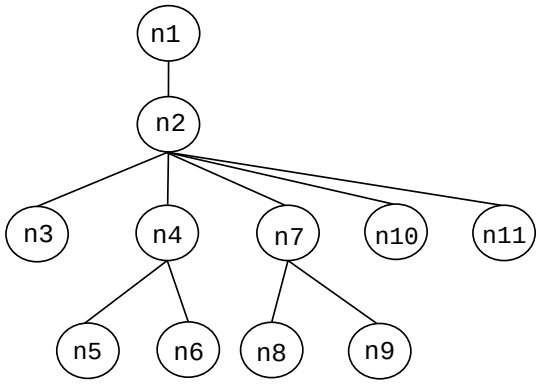

Figure 3.3: Data Tree t_1

Since the evaluation are made against a store that contains a family of trees we need to extend the relation to ensure a correct evaluation.

Definition 3.9 (Parent Edge relation) *Given a valid store η, the edge parent relation between two nodes $\mathbf{n}, \mathbf{m} \in \text{Nodes}(\eta)$ is defined as follows.*

$$\varepsilon_t^{\uparrow} = \{\, (\mathbf{n}, \mathbf{m}) \mid\ t@\mathbf{n} = \mathtt{tag}_{\mathbf{n}}[f\ t\ f'] \quad \text{and} \quad \text{rootNode}(t) = \mathbf{m} \}$$

We use $\varepsilon_t^{\uparrow +}$, $\varepsilon_t^{\uparrow *}$ to denote the transitive and the transitive and reflexive closure of the relation $\varepsilon_t^{\uparrow}$.

Closure of relations

Given a binary relation R, most of the time we need to use the *transitive*, and the *reflexive* and *transitive* closure of a R, that we dub R^+ and R^* respectively. In what follows we give the schema of the inductive definitions.

Definition 3.10 (Transitive closure) *Given a binary relation R, the transitive closure, denoted as R^+, is the minimum relation such that*

1. if $(x, y) \in R$ then $x \in R^+$
2. if $(x, z) \in R$ and $(z, y) \in R^+$ then $(x, y) \in R^+$

Definition 3.11 (reflexive and transitive closure) *Given the relation R, the transitive closure, denoted as R^*, is the minimum relation such that*

1. if $(x, y) \in R$ then $x \in R^+$
2. if $(x, z) \in R$ and $(z, y) \in R^+$ then $(x, y) \in R^+$
3. if $(x, x) \in R^*$

The semantics of an axis navigation is denoted by $[\![\mathsf{Axis}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle}$, where $\vec{\mathbf{n}}$ is a sequence of nodes that belongs to some trees into the store η, and is used as current context. The evaluation defined by using $\varepsilon_t^{\uparrow}$,$\varepsilon_t^{\uparrow+}$ and $\varepsilon_t^{\uparrow*}$ always yield a set.

$$
\begin{aligned}
[\![\texttt{self}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} &= \textstyle\bigcup_{\mathbf{n}_i \in \vec{\mathbf{n}}} \{\mathbf{n}_i\} \\
[\![\texttt{child}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} &= \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m} \mid (\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\uparrow},\ t \in \eta\} \\
[\![\texttt{parent}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} &= \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m} \mid (\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\uparrow},\ t \in \eta\} \\
[\![\texttt{descendant}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} &= \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m} \mid (\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\uparrow+},\ t \in \eta\} \\
[\![\texttt{ancestor}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} &= \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m} \mid (\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\uparrow+},\ t \in \eta\}
\end{aligned}
$$

Example 3.12 *As an example, for* $\eta = \{t_2\}$ *in* Figure (3.3) *we have that*

$$
\begin{aligned}
[\![\texttt{self}]\!]_{\langle \eta;\,(\mathbf{n}_1,\mathbf{n}_5)\rangle} &= (\mathbf{n}_1, \mathbf{n}_5) \\
[\![\texttt{child}]\!]_{\langle \eta;\,(\mathbf{n}_2)\rangle} &= (\mathbf{n}_3, \mathbf{n}_4, \mathbf{n}_7, \mathbf{n}_{10}, \mathbf{n}_{11}) \\
[\![\texttt{parent}]\!]_{\langle \eta;\,(\mathbf{n}_1,\mathbf{n}_9)\rangle} &= (\mathbf{n}_7) \\
[\![\texttt{descendant}]\!]_{\langle \eta;\,(\mathbf{n}_4,\mathbf{n}_7)\rangle} &= (\mathbf{n}_5, \mathbf{n}_6, \mathbf{n}_8, \mathbf{n}_9) \\
[\![\texttt{ancestor}]\!]_{\langle \eta;\,(\mathbf{n}_8)\rangle} &= (\mathbf{n}_7, \mathbf{n}_2, \mathbf{n}_1)
\end{aligned}
$$

As already stated, there is no difference between evaluating a Path in FLWR-XQuery and XQuery. In particular, we ensure two properties that are often relaxed in literature: in the evaluations all the duplicated nodes are removed and on path navigation the result of an axis selection is ordered wrt the document order, we dub $<_t$. XQuery resolves both the problem using the auxiliary functions `distinct-doc-order` on path navigation [7] that remove duplicates and order nodes to form the correct sequence. For what concerns the removal of duplicates they arises in expressions, as in the following query evaluation over the XML tree in Figure (3.4).

$$\texttt{//child :: node//author}$$

Once fixed the persistent root of the document all nodes are selected with

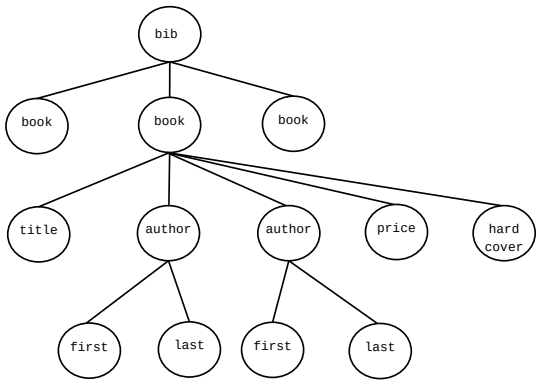

Figure 3.4: Data Tree t_2

//`child` :: `node` and then the descendant axis path navigation selects the `author` nodes. Since both the node `bib` and `book` satisfies the axis navigation we have that the nodes with `author` are taken twice. The problem is solved by using a *set semantic* for what concerns path navigation. If the evaluation yields to a set then duplicated nodes are implicitly removed. Further an external function, called docOrder() establish the document order between the elements of the set. This technique is that of the specifications of XPath 1.0.

Extension to siblings, preceding and followings

To perform node selection on axis like `following`, `following–sibling`, `preceding–sibling`, `preceding` we need to define some new relations between the nodes of a tree. We obtain the parent of a fixed node $\mathbf{n}$ using Definition (3.9) in order to define a relation between contiguous siblings.

Definition 3.13 (Right-Contiguous Edge Relation) *Given a valid store η the right contiguous siblings edge relation between two nodes $\mathbf{n}, \mathbf{m} \in \mathrm{Nodes}(\eta)$ is defined as follows.*

$$\varepsilon_t^{\rightarrow} = \{\, (\mathbf{n}, \mathbf{m}) \mid \varepsilon_t^{\uparrow}(\mathbf{n}) = \varepsilon_t^{\uparrow}(\mathbf{m}) = \mathbf{p} \;\; \wedge \;\; t@\mathbf{p} = l_{\mathbf{p}}[f'\, t'\, t''\, f''] \;\; \wedge$$

$$\wedge \;\; \mathrm{rootNode}(t') = \mathbf{n} \;\; \wedge \;\; \mathrm{rootNode}(t'') = \mathbf{m}\}$$

$\mathbf{p}$ is the parent of both $\mathbf{n}$ and $\mathbf{m}$. The nodes $\mathbf{n}$ and $\mathbf{m}$ are contiguous children and $\mathbf{n}$ precede $\mathbf{m}$. We will write $\varepsilon_t^{\rightarrow +}$ for the transitive closure of the relation that, at the matter of facts, models the relation between node siblings.

Example 3.14 *Given the tree* t_1 *in* Figure (3.3) *we have that:*

$$\varepsilon_t^{\rightarrow} = \{(\mathbf{n}_3, \mathbf{n}_4),\ (\mathbf{n}_4, \mathbf{n}_7),\ (\mathbf{n}_7, \mathbf{n}_{10}),\ (\mathbf{n}_{10}, \mathbf{n}_{11}),\ (\mathbf{n}_5, \mathbf{n}_6),\ (\mathbf{n}_8, \mathbf{n}_9)\}$$

And it's easy to see how the relation $\varepsilon_t^{\rightarrow}$ *works, for example*

$$\varepsilon_t^{\uparrow}(\mathbf{n}_3) = \mathbf{n}_2 = \varepsilon_t^{\uparrow}(\mathbf{n}_4)$$

and

$$t_1@\mathbf{n}_2 = l_{\mathbf{n}_2}[t', t'', f]$$

where

$$\text{rootNode}(t') = \mathbf{n}_3 \text{ and } \text{rootNode}(t'') = \mathbf{n}_4$$

Finally, we can define the relation between node followings in a store.

Definition 3.15 (Following Edge Relation) *Given a valid store* η *the followings edge relation between two tree nodes* $\mathbf{n}, \mathbf{m} \in \text{Nodes}(t)$ *is defined as follows.*

$$\begin{aligned}\varepsilon_t^{\twoheadrightarrow} = \{(\mathbf{n}, \mathbf{m}) |\ \ & \exists\, \mathbf{o}, \mathbf{p} \in \text{Nodes}(t)\ .\ \ (\mathbf{p}, \mathbf{n}) \in \varepsilon_t^{\uparrow *} \ \text{ and} \\ & (\mathbf{p}, \mathbf{o}) \in \varepsilon_t^{\rightarrow +} \ \text{ and } \ (\mathbf{o}, \mathbf{m}) \in \varepsilon_t^{\uparrow *}\}\end{aligned}$$

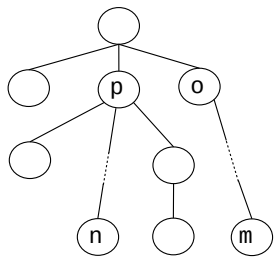

Figure 3.5: Followings edge relation between nodes

Example 3.16

To clarify the last definition we use as example the set of all the nodes in following relation with the node $\mathbf{n}_4$ belonging to the tree t_1 of Figure (3.3).

$$\varepsilon_t^{\twoheadrightarrow}(\mathbf{n}_4) = \{\mathbf{n}_7,\ \mathbf{n}_8,\ \mathbf{n}_9,\ \mathbf{n}_{10},\ \mathbf{n}_{11}\}$$

In particular

- $(\mathbf{n}_4, \mathbf{n}_8) \in \varepsilon_t^{\rightarrow\!\!\!\!\rightarrow}$ since $\varepsilon_t^{\uparrow *}(\mathbf{n}_4, \mathbf{n}_4)$, $\varepsilon_t^{\rightarrow +}(\mathbf{n}_4, \mathbf{n}_7)$ and $\varepsilon_t^{\uparrow *}(\mathbf{n}_7, \mathbf{n}_8)$
- $(\mathbf{n}_4, \mathbf{n}_{11}) \in \varepsilon_t^{\rightarrow\!\!\!\!\rightarrow}$ since $\varepsilon_t^{\uparrow *}(\mathbf{n}_4, \mathbf{n}_4)$, $\varepsilon_t^{\rightarrow +}(\mathbf{n}_4, \mathbf{n}_{11})$ and $\varepsilon_t^{\uparrow *}(\mathbf{n}_{11}, \mathbf{n}_{11})$

Now we can define the evaluation of the remaining axis navigation. For a light notations we use `fs`, `ps`, `f`, `p` to denote respectively `following–sibling`, `preceding–sibling`, `following`, `preceding`.

$$
\begin{aligned}
[\![\texttt{fs}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} &= \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\, \mathbf{m} \mid (\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\rightarrow +}\,,\ t \in \eta\} \\
[\![\texttt{ps}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} &= \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\, \mathbf{m} \mid (\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\rightarrow +}\,,\ t \in \eta\} \\
[\![\texttt{f}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} &= \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\, \mathbf{m} \mid (\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\rightarrow\!\!\!\!\rightarrow}\,,\ t \in \eta\} \\
[\![\texttt{p}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} &= \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\, \mathbf{m} \mid (\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\rightarrow\!\!\!\!\rightarrow}\,,\ t \in \eta\}
\end{aligned}
$$

Example 3.17 *For the tree t_1 in* Figure (3.3) *we give some evaluations.*

$$
\begin{aligned}
[\![\texttt{fs}]\!]_{\langle \eta;\, (\mathbf{n}_1,\mathbf{n}_8)\rangle} &= (\mathbf{n}_6, \mathbf{n}_9) \\
[\![\texttt{ps}]\!]_{\langle \eta;\, (\mathbf{n}_2,\mathbf{n}_6)\rangle} &= (\mathbf{n}_5) \\
[\![\texttt{f}]\!]_{\langle \eta;\, (\mathbf{n}_1,\mathbf{n}_9)\rangle} &= (\mathbf{n}_{10}, \mathbf{n}_{11}) \\
[\![\texttt{ps}]\!]_{\langle \eta;\, (\mathbf{n}_3,\mathbf{n}_7)\rangle} &= (\mathbf{n}_6, \mathbf{n}_5, \mathbf{n}_4, \mathbf{n}_3)
\end{aligned}
$$

In the end, we define the semantics for test filtering conditions.

$$
\begin{aligned}
[\![\texttt{node}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} &= \textstyle\bigcup_{i=1..|\vec{\mathbf{n}}|} \{\, \mathbf{n}_i \,\} \\
[\![\mathit{tag}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} &= \{\, \mathbf{n} \,|\, \mathbf{n} \in \vec{\mathbf{n}} \quad \mathit{and} \quad t@\mathbf{n} = \mathit{tag}_{\mathbf{n}}[f]\,,\ t \in \eta\} \\
[\![\texttt{text}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} &= \{\, \mathbf{n} \,|\, \mathbf{n} \in \vec{\mathbf{n}} \quad \mathit{and} \quad t@\mathbf{n} = s_{\mathbf{n}}\,,\ t \in \eta\}
\end{aligned}
$$

3.3.3 Operational Semantic

In what follows we give the deduction rules that are used to define the semantics of the language. Each rule consists of a set of premises and conclusions composed by semantics judgments specifying that the evaluation of an query `q` with respect to a store η and to a dynamical environment ρ (that

associate a sequence to each variable free in $\mathtt{q}$) produces a sequence of nodes $\vec{\mathbf{n}}$ and an updated store η'.

$$\rho \Vdash_{\eta} \mathtt{q} \Rightarrow \vec{\mathbf{n}}; \eta'$$

In particular the sequence $\vec{\mathbf{n}}$ is a sequence of *persistent roots* of subtrees belonging to the store η'. Since only element construction creates new trees in the store, in the cases where no updates are made, we omit to write η' in the right part of the query and leave it as implicit. We use a variable of the dynamical environment $c_d \in \{\mathtt{x_1}, \ldots, \mathtt{x_n}\}$ to bind the evaluation of the current context.

3.3.4 Description of rules

We describe here in detail how evaluates the single expressions.

Path Navigation

(S-Var) The evaluation of a variable bounded into the dynamical environment is simply its bounded sequence of nodes into the dynamical environment.

(S-VarPath) A path navigation, starting from the sequence bounded to a variable, substitutes the current context with the set of nodes bounded by the variable and then evaluates path navigation.

(S-StepPath) A path navigation starting from the sequence evaluated as a step, substitutes the current context with the set of nodes resulting from the step.

(S-Cond) The evaluation of a condition is performed iterating over each single element of the sequence bounded by the current context. As output, it gives either a node or an empty sequence depending on whether the node satisfies or not the condition. The results are concatenated in order to form a sequence.

(S-Axis) The axis proposed semantics follows XPath 1.0 semantics, is a *set semantic*. The set semantics implies that we loose order between elements, but we gain the assurance that no elements are repeated. To have document

$$\frac{c_d \mapsto \vec{\mathbf{m}} \in \rho \qquad [\![\mathsf{Axis}]\!]_{\langle \eta;\, \vec{\mathbf{m}}\rangle} = S \qquad \vec{\mathbf{n}} = \mathrm{docOrder}_\eta(S)}{\rho \Vdash_\eta \mathsf{Axis}\texttt{::node} \Rrightarrow \vec{\mathbf{n}}} \text{ (S-Axis)}$$

$$\frac{c_d \mapsto \vec{\mathbf{m}} \in \rho \qquad [\![\mathsf{Test}]\!]_{\langle \eta;\, \vec{\mathbf{m}}\rangle} = S \qquad \vec{\mathbf{n}} = \mathrm{docOrder}_\eta(S)}{\rho \Vdash_\eta \texttt{self::}\mathsf{Test} \Rrightarrow \vec{\mathbf{n}}} \text{ (S-Test)}$$

$$\frac{\mathbf{x} \mapsto \vec{\mathbf{n}} \in \rho}{\rho \Vdash_\eta \mathbf{x} \Rrightarrow \vec{\mathbf{n}}} \text{ (S-Var)}$$

$$\frac{\mathbf{x} \mapsto \vec{\mathbf{m}} \;\in\; \rho \qquad \rho[c_d \mapsto \vec{\mathbf{m}}] \;\Vdash_\eta\; \mathsf{Path} \;\Rrightarrow\; \vec{\mathbf{n}}}{\rho \Vdash_\eta \mathbf{x}/\mathsf{Path} \Rrightarrow \vec{\mathbf{n}}} \text{ (S-VarPath)}$$

$$\frac{\rho \;\Vdash_\eta\; \mathsf{Step} \;\Rrightarrow\; \vec{\mathbf{m}} \qquad \rho[c_d \mapsto \vec{\mathbf{m}}] \;\Vdash_\eta\; \mathsf{Path} \;\Rrightarrow\; \vec{\mathbf{n}}}{\rho \Vdash_\eta \mathsf{Step}/\mathsf{Path} \Rrightarrow \vec{\mathbf{n}}} \text{ (S-StepPath)}$$

$$\frac{\rho \;\Vdash_\eta\; \mathsf{Step}[\mathsf{Cond}] \;\Rrightarrow\; \vec{\mathbf{m}} \qquad \rho[c_d \mapsto \vec{\mathbf{m}}] \;\Vdash_\eta\; \mathsf{Path} \;\Rrightarrow\; \vec{\mathbf{n}}}{\rho \Vdash_\eta \mathsf{Step}[\mathsf{Cond}]/\mathsf{Path} \Rrightarrow \vec{\mathbf{n}}} \text{ (S-StepCondPath)}$$

$$\frac{\begin{array}{c}\rho[c_d \mapsto \vec{\mathbf{m}}_1] \;\Vdash_\eta\; \mathsf{Step} \;\Rrightarrow\; \vec{\mathbf{n}}_1 \\ \vdots \\ c_d \mapsto \vec{\mathbf{m}} \;\in\; \rho \qquad \rho[c_d \mapsto \vec{\mathbf{m}}_m] \;\Vdash_\eta\; \mathsf{Step} \;\Rrightarrow\; \vec{\mathbf{n}}_{|\vec{\mathbf{m}}|}\end{array}}{\rho \Vdash_\eta \mathsf{Step} \Rrightarrow (\vec{\mathbf{n}}_1, \ldots, \vec{\mathbf{n}}_{|\vec{\mathbf{m}}|})} \text{ (S-UnfoldedPath)}$$

$$\frac{\rho \Vdash_\eta \mathsf{Axis}\texttt{::node/self::}\mathsf{Test} \Rrightarrow \vec{\mathbf{n}}}{\rho \Vdash_\eta \mathsf{Axis}\texttt{::}\mathsf{Test} \Rrightarrow \vec{\mathbf{n}}} \text{ (S-AxisTest)}$$

$$\frac{\begin{array}{c}\rho[c_d \mapsto \vec{\mathbf{m}}_1] \;\Vdash_\eta\; \mathsf{Cond} \;\Rrightarrow\; \vec{\mathbf{n}}_1 \\ \vdots \\ c_d \mapsto \vec{\mathbf{m}} \;\in\; \rho \qquad \rho[c_d \mapsto \vec{\mathbf{m}}_m] \;\Vdash_\eta\; \mathsf{Cond} \;\Rrightarrow\; \vec{\mathbf{n}}_{|\vec{\mathbf{m}}|}\end{array}}{\rho \Vdash_\eta \texttt{self::node}[\mathsf{Cond}] \Rrightarrow (\vec{\mathbf{n}}_1, \ldots, \vec{\mathbf{n}}_{|\vec{\mathbf{m}}|})} \text{ (S-Cond)}$$

$$\frac{\rho \Vdash_\eta \mathsf{Step}/\texttt{self::node}[\mathsf{Cond}] \Rrightarrow \vec{\mathbf{n}}}{\rho \Vdash_\eta \mathsf{Step}[\mathsf{Cond}] \Rrightarrow \vec{\mathbf{n}}} \text{ (S-StepCond)}$$

Table 3.2: semantics Rules for Path Navigation

$$\frac{}{\rho \Vdash_\eta \ () \Rightarrow ()} \ \text{(S-Empty)}$$

$$\frac{\rho \Vdash_\eta q_1 \Rightarrow \vec{\mathbf{n}}_1; \eta_1 \qquad \rho \Vdash_{\eta_1} q_1 \Rightarrow \vec{\mathbf{n}}_2; \eta_2}{\rho \Vdash_\eta q_1, q_2 \Rightarrow \vec{\mathbf{n}}_1, \vec{\mathbf{n}}_2; \eta_2} \ \text{(S-Concat)}$$

$$\frac{\begin{array}{ll}\mathbf{n}_a \leftarrow (\text{freshId}(\eta), \bot) & \\ \rho \Vdash_\eta \ \mathtt{q} \Rightarrow \vec{\mathbf{m}} & t_{|\eta|+1} \leftarrow \text{insertInto}(\mathbf{n}_a, f) \\ f \leftarrow \text{clone}(\vec{\mathbf{m}}, \eta) & \eta' \leftarrow \eta \cup t_{|\eta|+1}\end{array}}{\rho \Vdash_\eta \ \texttt{<a>q} \Rightarrow \mathbf{n}_a \, ; \eta'} \ \text{(S-EltConstr)}$$

$$\frac{\begin{array}{ll} & \Gamma[c_d \mapsto \mathbf{m}_1] \quad \rho \Vdash_\eta q_2 \Rightarrow \vec{\mathbf{n}}_1 \\ & \vdots \\ \rho \Vdash_\eta q_1 \Rightarrow \vec{\mathbf{m}} & \Gamma[c_d \mapsto \mathbf{m}_{|\vec{\mathbf{m}}|}] \quad \rho \Vdash_\eta q_2 \Rightarrow \vec{\mathbf{n}}_{|\vec{\mathbf{m}}|}\end{array}}{\rho \Vdash_\eta \ \texttt{for x in } q_1 \ \texttt{return } q_2 \Rightarrow \vec{\mathbf{n}}_1, \ldots, \vec{\mathbf{n}}_{|\vec{\mathbf{m}}|}} \ \text{(S-For)}$$

$$\frac{\rho \Vdash_\eta q_1 \Rightarrow \vec{\mathbf{m}} \qquad \rho[c_d \mapsto \vec{\mathbf{m}}] \Vdash_\eta q_2 \Rightarrow \vec{\mathbf{n}}}{\rho \Vdash_\eta \ \texttt{let x} = q_1 \ \texttt{return } q_2 \Rightarrow \vec{\mathbf{n}}} \ \text{(S-Let)}$$

$$\frac{\rho \Vdash_\eta \ \mathsf{Cond} \Rightarrow \vec{\mathbf{m}} \qquad \rho \Vdash_\eta \ \mathtt{q} \Rightarrow \vec{\mathbf{o}} \qquad \text{if } \vec{\mathbf{m}} \ncong () \text{ then } \vec{\mathbf{n}} \leftarrow \vec{\mathbf{o}} \text{ else } \vec{\mathbf{n}} \leftarrow ()}{\rho \Vdash_\eta \ \texttt{if Cond then q} \Rightarrow \vec{\mathbf{n}}} \ \text{(S-If)}$$

Table 3.3: semantics Rules for FLOWR Expressions and Element Construction

$$\frac{(c_d \mapsto \vec{\mathbf{n}}) \in \rho}{\rho \Vdash_\eta \texttt{true} \Rightarrow \vec{\mathbf{n}}} \text{ (S-CondTrue)}$$

$$\frac{}{\rho \Vdash_\eta \texttt{false} \Rightarrow ()} \text{ (S-CondFalse)}$$

$$\frac{\rho \Vdash_\eta \mathsf{q} \Rightarrow \vec{\mathbf{m}} \qquad \text{if } \vec{\mathbf{m}} \cong () \text{ then } \vec{\mathbf{n}} \leftarrow () \text{ else } \vec{\mathbf{n}} \leftarrow \vec{\mathbf{m}}}{\rho \Vdash_\eta \mathsf{q} \Rightarrow \vec{\mathbf{n}}} \text{ (S-CondQry)}$$

$$\frac{\rho \Vdash_\eta \mathsf{Cond}_1 \Rightarrow \vec{\mathbf{n}}_1 \qquad \rho \Vdash_\eta \mathsf{Cond}_2 \Rightarrow \vec{\mathbf{n}}_2}{\rho \Vdash_\eta \mathsf{Cond}_1 \vee \mathsf{Cond}_2 \Rightarrow \vec{\mathbf{n}}_1, \vec{\mathbf{n}}_2} \text{ (S-CondDisj)}$$

$$\frac{(\mathtt{x} \mapsto \vec{\mathbf{m}}) \in \rho \qquad \text{if } \vec{\mathbf{m}} \cong () \text{ then } \vec{\mathbf{n}} \leftarrow \vec{\mathbf{m}} \text{ else } \vec{\mathbf{n}} \leftarrow ()}{\rho \Vdash_\eta \ominus(\mathtt{x}) \Rightarrow \vec{\mathbf{n}}} \text{ (S-CondIsEmptyVar)}$$

Table 3.4: semantics Rules for Conditions

order between elements we use the function docOrder(). We are also sure that every time we have to establish the document order between two elements, then both belong to the same tree. This is because path navigation are done always before element construction.

(S-Test) As in the preceding rules the semantics of a test filtering is a set that after is ordered to get a sequence.

(S-AxisTest) The evaluation of a simple step is defined unfolding it in two steps. First the axis selection evaluates in a sequence which is filtered by the test.

FLWR Expressions

(S-Empty) The evaluation of the empty query is the null sequence.

(S-Concat) The evaluation of a query concatenation is in turn the concatenation of result sequence of the subqueries. The subexpressions can build new elements because concatenation of element construction expressions is a possible case of query. This said, we must consider the evaluation of the second

query with the updated store in output from the evaluation of the first query.

(S-For) The semantics of the `for` construct is similar to a path navigation. The left query is evaluated and then, once a time and in order, element in the resulting sequence are bounded to the `x` variable before evaluating the right query. The final sequence is the concatenation of the resulting partial evaluations.

(S-Let) The semantics of the let construct first evaluate the left query and after binds the result sequence to `x` before executing the right query.

(S-If) If the condition expression is not the empty set then the query is evaluated, else the result is the null sequence.

Conditions

A condition is an expression that does not modify both the store and the dynamical environment.
(S-CondTrue) The evaluation of the constant true returns the current dynamical context.

(S-CondFalse) The evaluation of the constant false returns the empty sequence.

(S-CondQry) When a query is used as condition, by hypothesis does not modify both the store and the dynamical environment. The condition yields to true if the query evaluation is not an empty sequence.

(S-CondDisj) The disjunction of two conditions is the concatenation of both the result vectors.

(S-CondEmpty) The semantics of the *empty variable* evaluator is true when the variable is bounded with the null sequence into the store, else is false.

Element Construction

The semantics of element construction described in rule S-EltConstr is the only one that modify the store in FLWR-XQuery.

$$\frac{\begin{array}{ll}\mathbf{n}_a \leftarrow (\text{freshId}(\eta), \bot) & \\ \rho \Vdash_\eta \texttt{q} \Mapsto \vec{\mathbf{m}} & t_{|\eta|+1} \leftarrow \text{insertInto}(\mathbf{n}_a, f) \\ f \leftarrow \text{clone}(\vec{\mathbf{m}}, \eta) & \eta' \leftarrow \eta \cup t_{|\eta|+1}\end{array}}{\rho \Vdash_\eta \texttt{<a>q} \Mapsto \mathbf{n}_a \,;\, \eta'}$$

First of all a node $\mathbf{n}_a$ with *fresh* id wrt the nodes inside the store, with *null type* $\bot$ and tag specified between the brackets (in this case `tag`) is created. A fresh type (not already in $DN(E)$) is mandatory to do not infect the type inference which is based only on types already defined in the DTD. Then the subquery `q` is evaluated yielding the sequence $\vec{\mathbf{m}}$. Since every $\mathbf{m} \in \vec{\mathbf{m}}$ belongs to a tree into the store is possible to extract the forest induced by the nodes in $\vec{\mathbf{m}}$ called f. The function clone$(\vec{\mathbf{m}}, \eta)$ make a copy of the forest f induced by nodes in $\vec{\mathbf{m}}$, ascribing each node with a fresh id wrt the store η, but maintaining the type of the copied node. The nodes of the f are then ordered by the function serialize. In the end, a new tree with root the fresh node $\mathbf{n}_a$ and children the forest f is created, and addicted in the last position of the store.

Definition 3.18 (Document Order) *Given a tree t at the top level of the store η and set of nodes $U \subseteq$ Nodes(t) the* document order $<_t$ *is a total order relation defined as follows.*

$$\mathbf{n} <_t \mathbf{m} \quad \textit{iff} \quad (\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\uparrow +} \textit{ or } (\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\rightarrow}$$

We need a formulation to deal with a store rather than a single tree. Since every tree into the store is enumerated during creation as in S-EltConstr, it can be viewed as a sequence of trees $\{t_1, \ldots, t_p\}$ where t_1 is the input tree. It is possible then to order nodes that don't belong to the same tree in base of the order of the trees they belong into the store.

Definition 3.19 (Document Order for Store) *Given a store $\eta = \{t_1, \ldots, t_n\}$ and two nodes $\mathbf{n}, \mathbf{m} \in$ Nodes(η) the document order $<_\eta$ for the store is a total order relation defined as follows:*

$$\begin{array}{rll}\mathbf{n} <_\eta \mathbf{m} & iif & \mathbf{n} <_t \mathbf{m} \quad \textit{and} \quad \mathbf{n}, \mathbf{m} \in \text{Nodes}(t_i) \quad \textit{or} \\ & & \mathbf{n} \in \text{Nodes}(t_i),\ \mathbf{n} \in \text{Nodes}(t_j) \quad \textit{and} \quad i < j\end{array}$$

Definition 3.20 (Serialize Function) *Given a tree t and a subset of nodes of the tree $U \in \text{Nodes}(t)$ the serialize function orders the nodes following the document order.*

$$\begin{aligned} \text{docOrder}_t(U): \quad & Nodes \times Trees \to Sequences \\ & U, t \mapsto \vec{\mathbf{n}} \end{aligned}$$

and the following properties hold:

1. for each $i = 1 \dots |U| \quad \mathbf{n}_i \in \vec{\mathbf{n}} \quad iff \quad \mathbf{n}_i \in U$
2. for each $i = 1 \dots |U| - 1 \quad \mathbf{n}_i <_\eta \mathbf{n}_{i+1}$

Definition 3.21 (FreshId Function) *Given a store η the FreshId function gives back an identifier that is not used in the store.*

$$\begin{aligned} \text{freshId}(\eta): \quad & Stores \to Ids \\ & \eta \mapsto \text{j} \end{aligned}$$

and for each $\mathbf{n} \in \text{Nodes}(\eta)$ then $\mathbf{n}_{\text{id}} \neq \text{j}$.

Definition 3.22 (Clone Function) *Let $\vec{\mathbf{n}}$ a sequence of nodes belonging to a tree contained into a store η. The clone function makes a copy of the subforest composed only by nodes in the sequence, renaming all their ids but maintaining their types.*

$$\begin{aligned} \text{clone}(\eta, \vec{\mathbf{n}}): \quad & Stores \times Sequences \to Forests \\ & \eta, \vec{\mathbf{n}} \mapsto f \end{aligned}$$

Given a store η and a sequence of nodes $\vec{\mathbf{n}} \in \text{Nodes}(\eta)$ we define $[\vec{\mathbf{n}}]_\eta$ as the subforest induct by the nodes of $\vec{\mathbf{n}}$. This forest contains only the nodes belonging to the sequence and the edges between them.
Then for the clone function the following properties hold:

1. $\overline{[\vec{\mathbf{n}}]_\eta} = \overline{f}$
2. for each $k = 1 \dots |\vec{\mathbf{n}}|$ let $\mathbf{m}_k = \phi(\mathbf{n}_k)$ then $\mathbf{m}_X = \mathbf{n}_X$ and $\{\mathbf{n}_{\text{id}}\} \cap \text{Ids}(f) = \varnothing$

It's also interesting to see the following proposition.

Definition 3.23 (Insert-into Function) *Given a node* $\mathbf{n}$ *and a forest* f *the insert-into function creates a tree where the root is the node* $\mathbf{n}$ *with children* f*.*

$$\begin{aligned} \text{insertInto}(\mathbf{n}, f): \quad & Sequences \times Forests \rightarrow Trees \\ & \mathbf{n}, f \mapsto t \end{aligned}$$

and the following properties hold.

1. $\text{rootNode}(t) = \mathbf{n}$
2. $t@\mathbf{n} = l_{\mathbf{n}}[f]$

Chapter 4

Static Type Analysis

In this Chapter we define deduction rules to statically infer the type of a FLWR-XQuery expression that will be used in type projection. We define some relations between the types of a document valid wrt a schema to write basically the typing rules for the navigational axis, and then the rules to type the rest of the language. We show that our analysis is sound and also complete for a large class of documents. Soundness means that if an enriched type is inferred for a query and the evaluation yields to a sequence of nodes, then the enriched type contains all the type-names of the nodes in the result of the query. Completeness means that if we take an enriched type smaller (i.e. more selective) than the inferred one, then there exists an instance of XML document valid wrt the DTD such that, evaluating the query on that instance yields to elements whose types are not included in the set.

4.1 Type Inference

Definition 4.1 (Type set) *Let* $\vec{\mathbf{n}}$ *a sequence of nodes and* U *a set of nodes, then we define the set of the types of the nodes contained in* $\vec{\mathbf{n}}$ *as follows.*

$$\chi(\vec{\mathbf{n}}) = \bigcup_{i=1..|\vec{\mathbf{n}}|} \mathbf{n}^i_X$$

$$\chi(U) = \bigcup_{\mathbf{n}\in U} \mathbf{n}_X$$

where, in this case, $\mathbf{n}^i$ stands for the i-th element of the sequence.

The aim of type inference, given a query q and a DTD (W, E), is to find a

subset of names of $DN(E)$ that generates elements that can be found in the evaluation of $\mathtt{q}$ over a tree valid wrt the DTD.
Formally we want to infer a set $\tau \subseteq DN(E)$ such that for all trees t valid wrt the DTD (W, E),

$$\text{if } \rho \Vdash_\eta \mathtt{q} \Rightarrow \vec{\mathbf{n}} \text{ then } \chi(\vec{\mathbf{n}}) \subseteq \tau \tag{4.1}$$

Moreover we aim at an analysis which is precise enough to guarantee, on a large class of types and for a large class of queries, that whenever the query semantics is empty over all possible instances of the input DTD, then the inferred type τ is empty, as well:

$$\text{if } \rho \Vdash_\eta \mathtt{q} \Rightarrow () \text{ then } \tau = \varnothing \tag{4.2}$$

This property is crucial in order to have precise projectors. We start our type inference for the navigational axes.

4.2 Relations over Type-Names

A DTD type the nodes of the data-model instance. Every single production defines a type and the types in its content model. Moreover, between the types of a content model there's an order that must be respected by a document in order to be valid. The main idea is that we can use these characteristics of the schema to determine a dependency between types and statically infer the type of an axis selection.

Considering the portion of DTD below extract from Example (3.2), we have for example this definitions:

Example 4.2

<!ELEMENT bib (book*)>

<!ELEMENT book (title, (author + |editor+), publisher, price)>

which corresponds to the productions

Bib → bib[Book*]
Book → book[Title, (Author+ | Editor+), Publisher, Price]

it's easy to see that in a valid XML data-model instance all the nodes typed by Title, Author, Editor, Publisher and Price have as a parent a node

typed by `Book`. Moreover, it's easy to see that a node typed with `Publisher` must appear before a node typed by `Price`.

A possible valid data model instance wrt the DTD is the following:

Definition 4.3 (Forward) *Given a DTD (W, E) and Y, $Z \in DN(E)$, we write $Z \Rightarrow_E Y$ iff $Z \rightarrow l[r] \in E$ and $Y \in Names(r)$. We use $\Rightarrow_E^+$ and $\Rightarrow_E^*$ to denote respectively the transitive closure and the transitive and reflexive closure of $\Rightarrow_E$.*

For the portion of DTD in Example (4.2) we have that:

$$\texttt{Bib} \Rightarrow_E \texttt{Book}$$
$$\texttt{Bib} \not\Rightarrow_E \texttt{Author}$$
$$\texttt{Bib} \Rightarrow_E^+ \texttt{Author}$$
$$\texttt{Bib} \Rightarrow_E^* \texttt{Bib}$$

Then we can state that if two nodes are in the Parent edge relation wrt a data tree, then their type-names are in the Forward-Reachbility relation wrt the schema.

Lemma 4.4 *Given a tree t valid with respect to a DTD (W,E)and two nodes $\mathbf{n}, \mathbf{m} \in \text{Nodes}(t)$ such that $\mathbf{n}_X, \mathbf{m}_X \in DN(E)$. If $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\uparrow}$ then $\mathbf{n}_X \Rightarrow_E \mathbf{m}_X$.*

Proof.

Since $\mathbf{n}_X, \mathbf{m}_X \in DN(E)$ and following the definition of $\varepsilon_t^{\uparrow}$ we have that if $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\uparrow}$ then

$$t@\mathbf{n} = l[f'\ t\ f''] \quad \text{and} \quad \text{rootNode}(t) = \mathbf{m}$$

and since t is valid we have that

$$\exists\, Y \rightarrow l[\,r\,] \in E \quad \text{such that} \quad \mathbf{n}_X = Y \quad \text{and} \quad \mathbf{m}_X \in Names(r)$$

and by definition of $\Rightarrow_E$ we have that $\mathbf{n}_X \Rightarrow_E \mathbf{m}_X$ □

We can extend Lemma (4.4) at both the transitive and reflexive and transitive closures of the relations $\varepsilon_t^{\uparrow}$ and $\Rightarrow_E$. If two nodes are in the transitive closure of the Parent edge relation (that is, ancestor-descendant relation), then their type-names are in the transitive closure of the Forward relation.

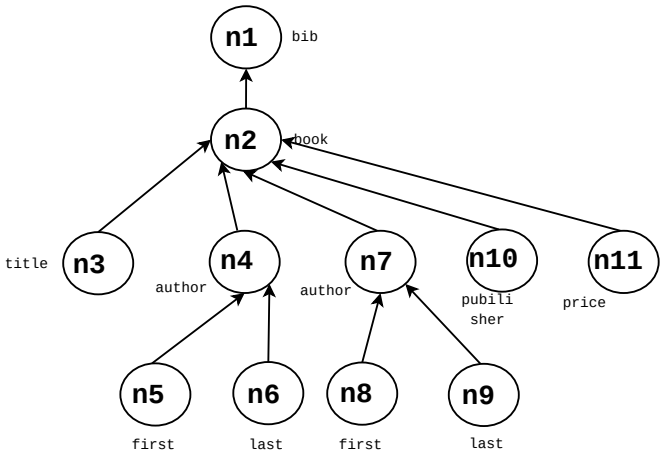

Figure 4.1: Graph of Nodes in $\varepsilon_t^{\uparrow}$ Relation for document in Example (3.2)

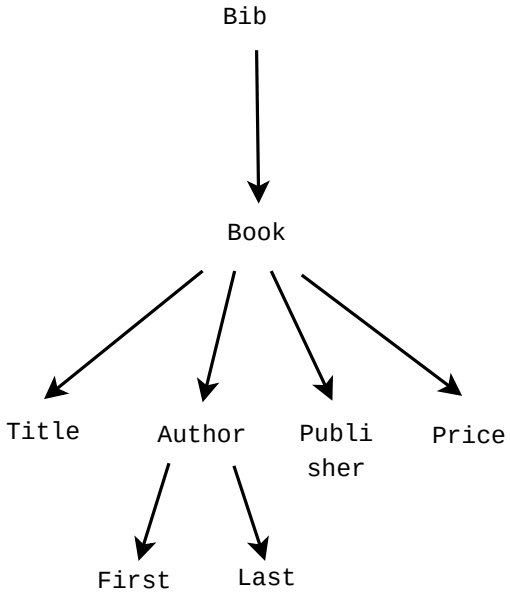

Figure 4.2: Graph of Type dependency in $\Rightarrow_E$ Relation for DTD in Example (3.2)

Lemma 4.5 *Given a tree t valid with respect to a DTD (W,E)and two nodes* $\mathbf{n}, \mathbf{m} \in \text{Nodes}(t)$ *such that* $\mathbf{n}_X, \mathbf{m}_X \in DN(E)$. *If* $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\uparrow +}$ *then* $\mathbf{n}_X \Rightarrow_E^+ \mathbf{m}_X$.

Proof.

Since $\mathbf{n}_X, \mathbf{m}_X \in DN(E)$ and following the inductive definition of $\varepsilon_t^{\uparrow +}$ there are two cases.

1. $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\uparrow}$ then by Lemma (4.4) holds Forward relation between node types $\mathbf{n}_X \Rightarrow_E \mathbf{m}_X$, and then by definition of transitive closure $\mathbf{n}_X \Rightarrow_E^+ \mathbf{m}_X$.

2. $\exists\, \mathbf{o}$ such that $(\mathbf{n}, \mathbf{o}) \in \varepsilon_t^{\uparrow +}$and $(\mathbf{o}, \mathbf{m}) \in \varepsilon_t^{\uparrow +}$. By inductive hypothesis $\mathbf{n}_X \Rightarrow_E^+ \mathbf{o}_X$ and $\mathbf{o}_X \Rightarrow_E^+ \mathbf{m}_X$. Finally, by definition of transitive closure $\mathbf{n}_X \Rightarrow_E^+ \mathbf{m}_X$ □

If two nodes are in the reflexive and transitive closure of the Parent edge relation (that is, ancestor-descendant relation), then their type-names are in the reflexive and transitive closure of the Forward relation.

Lemma 4.6 *Given a tree t valid with respect to a DTD (W,E)and two nodes* $\mathbf{n}, \mathbf{m} \in \text{Nodes}(t)$ *such that* $\mathbf{n}_X, \mathbf{m}_X \in DN(E)$. *If* $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\uparrow *}$ *then* $\mathbf{n}_X \Rightarrow_E^* \mathbf{m}_X$.

Proof.

Since $\mathbf{n}_X, \mathbf{m}_X \in DN(E)$ and following the inductive definition of $\varepsilon_t^{\uparrow +}$ there are three cases.

1. $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\uparrow}$ then by Lemma (4.4) holds Forward relation between node types $\mathbf{n}_X \Rightarrow_E \mathbf{m}_X$, and then by definition of transitive closure $\mathbf{n}_X \Rightarrow_E^* \mathbf{m}_X$.

2. $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\uparrow +}$ by Lemma (4.5) we have that $\mathbf{n}_X \Rightarrow_E^+ \mathbf{m}_X$ and then $\mathbf{n}_X \Rightarrow_E^* \mathbf{m}_X$.

3. $(\mathbf{n}, \mathbf{n}) \in \varepsilon_t^{\uparrow}$ and by definition $\mathbf{n}_X \Rightarrow_E^* \mathbf{n}_X$ □.

To define a relation between types that appear contiguously in the data-model of a production we need a more fine formalism. This because the irregularity of semistructured data allowed by "?", "∗", "+" and "|" implies

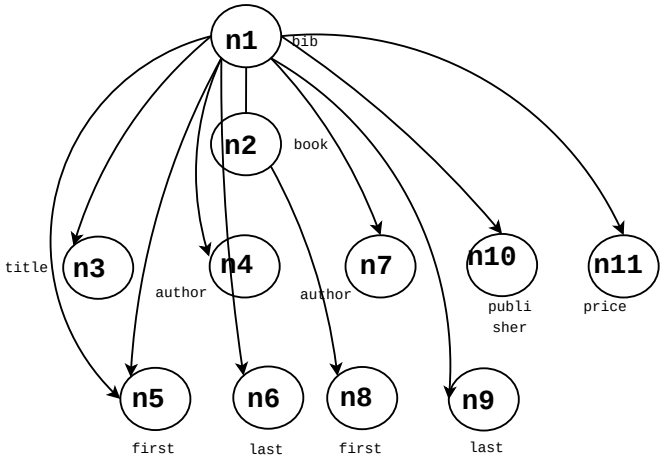

Figure 4.3: Graph of Nodes in $\varepsilon_t^{\uparrow+}$ Relation for the node $\mathbf{n}_1$ for document in Example (3.2)

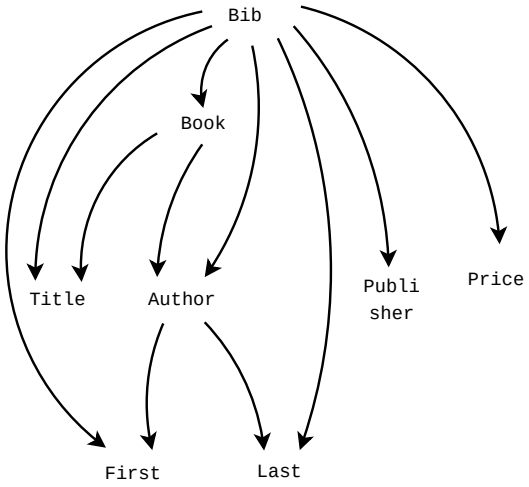

Figure 4.4: Graph of Type dependency in $\Rightarrow_E^+$ Relation for DTD in Example (3.2)

that a type can be followed by a *set* of types rather than a single one. We recall the previous example:

$$\texttt{Bib} \rightarrow \texttt{bib}[\texttt{Book}^*]$$
$$\texttt{Book} \rightarrow \texttt{book}[\texttt{Title}, (\texttt{Author}^+ \,|\, \texttt{Editor}^+), \texttt{Publisher}, \texttt{Price}]$$

in this case, for example, we have that either `Author` or `Editor` can be contiguous to `Title` or `Publisher`, respectively as preceding or following.

Then, given a type-name Y and a regular expression r, we define three functions that capture the set of all the types that can follow contiguously Y. These functions are: first(r), the set of types that match the first symbol of some word in the language generated by r; last(r) the dual set for last position and symbols, and follow(r, Y) that capture the set of all types S that can follow a specific type Y in the regular expression r.

For simplicity we modify the grammar that generates regular expression:

$$r ::= \quad r|r \quad | \quad (r,r) \quad | \quad r^+ \quad | \quad Y \quad | \quad \epsilon$$

of course in the following we will consider $r^+ = (r, r^*)$ and $r^? = r \mid \epsilon$.

Definition 4.7 (Language Generated by a Regular Expression) *A regular language $L(r)$ is the set of words generated by the regular expression r.*

Then SGML standard rules over the kind of regular expression that can be used to define content model of types. Since XML is defined on SGML the rules are the same. In particular, only regular expressions called *one-unambiguous* can be used to define a content model of a type. The peculiarity of those regular expression, as suggested in [19], [17] and [18], is that is possible to build an automaton to check if $w \in L(r)$ in a time which is quadratic respect to the size of the one-unambiguous regular expression. Validation does not concerns our study, so we refer the reader to the suggested literature for details on this topic.

Definition 4.8 (First and Last Functions) *We can define first and last inductively.*

$$\text{first}(\,\epsilon\,) = \text{last}(\,\epsilon\,) = \varnothing \qquad \text{first}(A) = \text{last}(A) = \{A\}$$

$$\text{first}(\,r_1|r_2\,) = \text{first}(r_1) \cup \text{first}(r_2)$$

$$\text{last}(\, r_1|r_2\,) = \text{last}(r_1) \cup \text{last}(r_2)$$

$$\text{first}(\, r_1\,,\, r_2\,) = \begin{cases} \text{first}(r_1) \cup \text{first}(r_2) & \textit{if } \epsilon \in r_1 \\ \text{first}(r_1) & \textit{otherwise} \end{cases}$$

$$\text{last}(\, r_1\,,\, r_2\,) = \begin{cases} \text{last}(r_1) \cup \text{last}(r_2) & \textit{if } \epsilon \in r_2 \\ \text{last}(r_1) & \textit{otherwise} \end{cases}$$

$$\text{first}(r_1^+) = \text{first}(r_1)$$

$$\text{last}(r_1^+) = \text{last}(r_1)$$

It is possible to check if ϵ belongs to a regular expression r in a time linear to the length of r by operating a postorder traversal of the syntax tree of r.

Definition 4.9 (Follow function) *The function follow(r,Y) maps a name in r to subset of names belonging to r. Each element in the resulting set can follow contiguously Y in a word generated by r.*

$$\text{follow}(\, \epsilon, Y) = \varnothing \qquad \text{follow}(Z, Y) = \varnothing$$

$$\text{follow}(\, r_1|r_2\,,\, Y) \;=\; \text{follow}(r_1, Y) \;\cup\; \text{follow}(r_2, Y)$$

$$\text{follow}(\, r_1, r_2\,,\, Y_1) \;=\; \begin{cases} \text{follow}(r_1, Y) \;\cup\; \text{follow}(r_2, Y) & \textit{if } Y \notin \text{last}(r_1) \\ \text{follow}(r_1, Y) \;\cup\; \text{follow}(r_2, Y) \\ \cup\, \text{first}(r_2) & \textit{if } Y \in \text{last}(r_1) \end{cases}$$

$$\text{follow}(r_1^+, Y_1) = \text{follow}((r_1, r_1)\,,\, Y_1)$$

The inductive definition suggests a computation of first, last, and follow that is cubic in the size of r. We discovered later that the same formalization we defined were used in the context of document validation in and originally formulate in [18].
Now we test our definition with some examples.

Example 4.10 *Let* $r = (\texttt{Title}, (\texttt{Author}|\texttt{Editor}))$ *we have that*

$$\begin{aligned}
\text{follow}(\texttt{Title}, r) &= \text{follow}(\texttt{Title},\ \texttt{Title}) \cup \text{first}(\texttt{Author}^+|\texttt{Editor}^+) \\
&= \varnothing \cup \text{first}(\texttt{Author}^+) \cup \text{first}(\texttt{Editor}^+) \\
&= \text{first}(\texttt{Author}, \texttt{Author}) \cup \text{first}(\texttt{Editor}, \texttt{Editor}) \\
&= \text{first}(\texttt{Author}) \cup \text{first}(\texttt{Editor}) \\
&= \{\texttt{Author}, \texttt{Editor}\}
\end{aligned}$$

Example 4.11 *Let* $r = (\texttt{Title}, \texttt{Hardcover}^?, \texttt{Price})$ *we want to compute the follows of* `Title`

$$\begin{aligned}
\text{follow}(\texttt{Title}, r) &= \text{follow}(\texttt{Title},\ \texttt{Title}) \cup \text{first}(\texttt{Hardcover}^?, \texttt{Price}) \\
&= \varnothing \cup \text{first}(\,(\texttt{Hardcover}|\epsilon)\,, \texttt{Price}) \\
&= \text{first}(\,\texttt{Hardcover}|\epsilon\,) \cup \text{first}(\texttt{Price}) \\
&= \text{first}(\texttt{Hardcover}) \cup \text{first}(\epsilon) \cup \text{first}(\texttt{Price}) \\
&= \{\texttt{Hardcover}, \texttt{Price}\}
\end{aligned}$$

Example 4.12 *Let* $r = (\texttt{Title}^*, (\texttt{Author}|\texttt{Editor})^*, \texttt{Hardcover}^?)$ *we have:*

$$\begin{aligned}
\text{last}(r) &= \text{last}(\,(\texttt{Title}^+|\epsilon\,),\ (\,(\texttt{Author}|\texttt{Editor})^+|\epsilon\,),\ (\,\texttt{Hardcover}|\epsilon\,)\,) \\
&= \text{last}(\,(\texttt{Title}|\epsilon), (\,(\texttt{Author}|\texttt{Editor})|\epsilon\,), (\,\texttt{Hardcover}|\epsilon\ ) \\
&= \text{last}(\,\texttt{Title}|\epsilon)\ \cup\ \text{first}(\,(\texttt{Author}|\texttt{Editor})\,|\epsilon), (\texttt{Hardcover}|\epsilon)\,) \\
&= \text{last}(\,\texttt{Title}) \cup \text{last}(\epsilon)\ \cup\ \text{last}(\,(\texttt{Author}|\texttt{Editor})\,|\epsilon\,)\,) \\
&\quad \cup \text{first}(\texttt{Hardcover}|\epsilon\,) \\
&= \text{last}(\,\texttt{Title})\ \cup\ \text{last}(\texttt{Author}|\texttt{Editor})\ \cup\ \text{last}(\epsilon) \\
&\quad \cup \text{first}(\texttt{Hardcover}|\epsilon\,) \\
&= \text{last}(\,\texttt{Title})\ \cup\ \text{last}(\texttt{Author})\ \cup\ \text{last}(\texttt{Editor}) \\
&\quad \cup \text{last}(\texttt{Hardcover}\,)\ \cup\ \text{last}(\epsilon) \\
&= \{\texttt{Title}, \texttt{Author}, \texttt{Editor}, \texttt{Hardcover}\}
\end{aligned}$$

Definition 4.13 (Right-Contiguous Reachbility) *Given a DTD* (X, E)*, and two names* Y_1,$Y_2 \in DN(E)$*, we say that* Y_2 *follow contiguously* Y_2*, and write* $Y_1 >_E Y_2$*, if and only if exist a production* $Z \to l[r] \in E$ *such that* $Y_2 \in \text{follow}(r, Y_1)$*. We denote with* $>_E^+$, $>_E^*$ *respectively the transitive and the reflexive and transitive closure of the relation.*

Considering the portion of DTD defined in Example (4.2) we have that:

$$\begin{aligned}
&\texttt{Title} >_E \texttt{Editor} \\
&\texttt{Editor} >_E \texttt{Publisher} \\
&\texttt{Editor} \not>_E \texttt{Author} \\
&\texttt{Title} >_E^+ \texttt{Price} \\
&\texttt{Author} >_E^* \texttt{Author}
\end{aligned}$$

Lemma 4.14 (Contiguity) *Given two words u, w generated by r and two positions Y, Z generated by r. If $uYZw$ is generated by r then $Z \in \text{follow}(r, Y)$.*

Lemma 4.15 *Given a tree t valid with respect to a DTD (W,E)and two nodes $\mathbf{n}, \mathbf{m} \in \text{Nodes}(t)$ such that $\mathbf{n}_X, \mathbf{m}_X \in DN(E)$. If $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\rightarrow}$ then $\mathbf{n}_X >_E \mathbf{m}_X$*

Proof.

Since $\mathbf{n}_X, \mathbf{m}_X \in DN(E)$ and following the definition we have that if $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\rightarrow}$ then

$$\varepsilon_t^{\uparrow}(\mathbf{n}) = \varepsilon_t^{\uparrow}(\mathbf{m}) = \mathbf{p} \ , \quad t@\mathbf{p} = l_{\mathbf{p}}[f'\, t\, t'\, f'']$$

$$\text{rootNode}(t) = \mathbf{n} \quad \text{and} \quad \text{rootNode}(t') = \mathbf{m}$$

Moreover by definition of $\varepsilon_t^{\uparrow}$ and since a DTD is a local tree grammar we have that the type of the nodes, with the same parent, must be in the same regular expression and then

$$\exists\ Z \rightarrow l[\,r\,] \in E \quad \text{and} \quad \mathbf{n}_X, \mathbf{m}_X \in r$$

Finally, since $l_{\mathbf{p}}[f'\, t\, t'\, f'']$ is a production then $a\, \mathbf{n}_X\, \mathbf{m}_X\, b$ (where a and b are words) is in the language generated by the regular expression r, by Lemma (4.14) then $\mathbf{m}_X \in \text{follow}(r, \mathbf{n}_X)$ and by definition $\mathbf{n}_X >_E \mathbf{m}_X$.

The lemma can be extended, stating that if two nodes are in the Right-Contiguous edge relation, they their type are in transitive closure of the Right-Contiguous reachbility relation.

Lemma 4.16 *Given a tree t valid with respect to a DTD (W,E)and two nodes $\mathbf{n}, \mathbf{m} \in \text{Nodes}(t)$ such that $\mathbf{n}_X, \mathbf{m}_X \in DN(E)$. If $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\rightarrow +}$ then $\mathbf{n}_X >_E^+ \mathbf{m}_X$*

Proof.

Since $\mathbf{n}_X, \mathbf{m}_X \in DN(E)$ and by the inductive definition of $\varepsilon_t^{\rightarrow +}$ there are two cases.

1. If $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\rightarrow}$ then holds right-contiguous reachbility between node types by Lemma (4.15) $\mathbf{n}_X >_E \mathbf{m}_X$, and then by definition of transitive closure $\mathbf{n}_X >_E^+ \mathbf{m}_X$.

2. By induction if $\exists$ **o** such that $(\mathbf{n}, \mathbf{o}) \in \varepsilon_t^{\rightarrow +}$and $(\mathbf{o}, \mathbf{m}) \in \varepsilon_t^{\rightarrow +}$ and $\mathbf{n}_X >_E^+ \mathbf{o}_X$ and $\mathbf{o}_X >_E^+ \mathbf{m}_X$ and then by definition of transitive closure $\mathbf{n}_X >_E^+ \mathbf{m}_X$.

Finally, we define a relation between two names that can appear one-right-wrt-the-other in the language of a regular expression.

Definition 4.17 (Right Reachbility) *Given a well formed DTD (X, E), and two names Y_1, $Y_2 \in DN(E)$, we say that Y_2 is right reachable from Y_1, and write $Y_1 \gg Y_2$, iff exists $Z, X, Y_1, Y_2 \in DN(E)$ such that $Z \Rightarrow_E^* Y_1$ and $Z >^+ X$ and $X \Rightarrow_E^* Y_2$.*

And, as before, a lemma states that if two nodes are in the Following edge relation, then their name-types are in the Right-Reachbility Relation.

Lemma 4.18 *Given a tree t valid wrt a dtd (W,E), and two nodes $\mathbf{n}, \mathbf{m} \in$ nodes(t), if $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\rightarrow\!\!\!\!\rightarrow}$* then $\mathbf{n}_X \gg \mathbf{m}_X$

Proof.

If $\mathbf{n}_X \notin DN(E)$ or $\mathbf{m}_X \notin DN(E)$ then the thesis follows.

Else $\mathbf{n}_X, \mathbf{m}_X \in DN(E)$ and if $(\mathbf{n}_X, \mathbf{m}_X) \in \varepsilon_t^{\rightarrow\!\!\!\!\rightarrow}$ then there exists $\mathbf{o}, \mathbf{p} \in$ Nodes(t) such that $(\mathbf{p}, \mathbf{n}) \in \varepsilon_t^{\uparrow *}$, $(\mathbf{p}, \mathbf{o}) \in \varepsilon_t^{\rightarrow +}$ and $(\mathbf{o}, \mathbf{m}) \in \varepsilon_t^{\uparrow *}$. By Lemmas (4.6) and (4.16) and Definition (4.17) of Right-Reachbility follows immediately that $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\rightarrow\!\!\!\!\rightarrow}$ □

At this point, we can define static type inference for a Step navigation. We recall that a Step is composed by axis navigation and test filtering and it's presented in the form Axis::Test.

Definition 4.19 (Axis Typing) *Let (W, E) be a DTD, and a type $\tau \subseteq DN(E)$.*

$$
\begin{aligned}
\mathbf{A}_E(\tau\,,\,\texttt{self}) &= \tau \\
\mathbf{A}_E(\tau\,,\,\texttt{child}) &= \textstyle\bigcup_{Y\in\tau}\{Z \mid Y \Rightarrow_E Z\} \\
\mathbf{A}_E(\tau\,,\,\texttt{descendant}) &= \textstyle\bigcup_{Y\in\tau}\{Z \mid Y \Rightarrow_E^+ Z\} \\
\mathbf{A}_E(\tau\,,\,\texttt{parent}) &= \textstyle\bigcup_{Y\in\tau}\{Z \mid Y \Rightarrow_E Z\} \\
\mathbf{A}_E(\tau\,,\,\texttt{ancestor}) &= \textstyle\bigcup_{Y\in\tau}\{Z \mid Y \Rightarrow_E^+ Z\} \\
\mathbf{A}_E(\tau\,,\,\texttt{following–sibling}) &= \textstyle\bigcup_{Y\in\tau}\{Z \mid Y >^+ Z\} \\
\mathbf{A}_E(\tau\,,\,\texttt{preceding–sibling}) &= \textstyle\bigcup_{Y\in\tau}\{Y \mid Z >^+ Y\} \\
\mathbf{A}_E(\tau\,,\,\texttt{following}) &= \textstyle\bigcup_{Y\in\tau}\{Z \mid Y \gg Z\} \\
\mathbf{A}_E(\tau\,,\,\texttt{preceding}) &= \textstyle\bigcup_{Y\in\tau}\{Y \mid Z \gg Y\}
\end{aligned}
$$

Definition 4.20 (Test Filtering Typing) *Let (W, E) be a DTD, and a type $\tau \subseteq DN(E)$.*

$$\begin{aligned} \mathbf{T}_E(\tau, \texttt{node}) &= \tau \\ \mathbf{T}_E(\tau, tag) &= \{\, Y \mid Y \in \tau\,,\ Y \rightarrow tag[r] \in E\} \\ \mathbf{T}_E(\tau, \texttt{text}) &= \{\, Y \mid Y \in \tau\,,\ Y \rightarrow \texttt{pcdata}[] \in E\} \end{aligned}$$

4.2.1 Type-contexts

The presence of upward axes such as `parent` and `ancestor` makes the typing of composed paths difficult because in DTDs an element can appear in the context of two different elements. The naive solution consisting of inferring a type for composed path by composing the typing we just defined for single steps, works only in absence of upward axes. For example, given the simplified DTD rooted at W

$$\begin{aligned} &\texttt{Book} \rightarrow \texttt{book}[\texttt{Title}(\texttt{Author}|\texttt{Editor})] \\ &\texttt{Title} \rightarrow \texttt{title}[\texttt{String}] \\ &\texttt{Author} \rightarrow \texttt{author}[\texttt{Name}] \\ &\texttt{Editor} \rightarrow \texttt{editor}[\texttt{Name}] \\ &\texttt{Name} \rightarrow \mathit{String} \end{aligned}$$

which generates a book with either the author or the editor information. We put the name-type `Name` in two different content models. If we consider the path

$$/\texttt{book}/\texttt{author}/\texttt{name}/\texttt{parent} :: \texttt{node}$$

then the precise type that we should have is $\{\texttt{Author}\}$, but repeatly using definition (4.19) and (4.20) we end up with $\{\texttt{Author}, \texttt{Editor}\}$. This is because the reachbility relation $\Rightarrow_E$ holds both for (`Author`, `Name`) and (`Editor`, `Name`).

To resolve this problem we need to introduce particular types called *context types*, to be updated at each step of the type process and containing names already encountered in previous steps. Then we use them to refine type inference for upward axes. In the previous example, when typing the second step we build a context type $\{\texttt{Book}, \texttt{Author}, \texttt{Name}\}$ indicating that for the moment the three names are the only ones visited by the traversal. Then we use definition (4.19) to type `parent` thus obtaining $\{\texttt{Author}, \texttt{Editor}\}$ but we *intersect* it with the type-context obtaining the correct type $\{\texttt{Author}\}$. So, using contexts and intersection we have the possibility to refine inferred types for backward axes.

We denote a type-context with κ. And for the soundness of our analysis we need that for all trees t valid wrt the DTD (W, E), if

$$\text{if } \rho \Vdash_\eta \mathtt{q} \Rrightarrow \vec{\mathbf{n}} \text{ then } \chi([\![\mathtt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) \subseteq \kappa \tag{4.3}$$

this means that all the type of the ancestor of the nodes in the evaluation of a query must be in κ.
The pair $\Sigma = (\tau, \kappa)$ composed by a type and an context type is called *enriched type*. An enriched type is well formed with respect to a DTD (W, E) when $\tau \subseteq DN(E)$ and $\kappa \subseteq \mathbf{A}_E(\tau, \mathtt{ancestor})$, that is, if the type-context contains only names that occurs in chains ending with names in τ.

4.3 Typing Rules

Typing rules infers an enriched type Σ for the query $\mathtt{q}$.

$$\Gamma \vdash_E \mathtt{q} : \Sigma$$

As for the semantics of the language we need an environment Γ which is a *static environment* that binds *variables* and the *current context* to the proper *enriched type*.

$$\Gamma = \{\mathtt{x_1} \mapsto \Sigma^1, \ldots, \mathtt{x_n} \mapsto \Sigma^n\}$$

We use distinguished a variable of the static environment $c_s \in \{\mathtt{x_1}, \ldots, \mathtt{x_n}\}$ to bind the enriched type of the *current context*. Given a variable $\mathtt{x_i}$ we need to access to the static environment to take (if any) the enriched type bounded to a variable, and we write $\Gamma(\mathtt{x_i}) = \Sigma^i$. Moreover we can both define a new one or overwrite an existing binding with $\Gamma[\mathtt{x_j} \mapsto \Sigma^j]$.

Definition 4.21 (Union of enriched types) *Given two enriched types* $\Sigma^1 = (\tau_1, \kappa_1)$ *and* $\Sigma^2 = (\tau_2, \kappa_2)$ *their union is a new enriched type defined as follows:*

$$\Sigma^1 \cup \Sigma^2 = (\tau_1 \cup \tau_2\,,\ \kappa_1 \cup \kappa_2)$$

4.3.1 Description of rules

(T-AxisTest) This rule unfold the axis and test selection typing in two separated steps.

$$\frac{\Gamma \vdash_E \mathsf{Axis}{::}\texttt{node}/\texttt{self}{::}\mathsf{Test} : \Sigma}{\Gamma \vdash_E \mathsf{Axis}{::}\mathsf{Test} : \Sigma}\ (\text{T-AxisTest}) \qquad \frac{\Gamma \vdash_E \mathsf{Step}/\texttt{self}{::}\texttt{node}[\mathsf{Cond}] : \Sigma}{\Gamma \vdash_E \mathsf{Step}[\mathsf{Cond}] : \Sigma}\ (\text{T-StepCond})$$

$$\frac{c_s{\mapsto}(\tau,\kappa) \in \Gamma \qquad \mathsf{Axis} \in \{\,\texttt{self}, \texttt{child}, \texttt{descendant}\,\}}{\Gamma \vdash_E \mathsf{Axis}{::}\texttt{node} : (\mathbf{A}_E(\tau,\mathsf{Axis})\,,\, \kappa \cup \mathbf{A}_E(\tau,\mathsf{Axis}))}\ (\text{T-ForwAx})$$

$$\frac{c_s{\mapsto}(\tau,\kappa) \in \Gamma \qquad \mathsf{Axis} \in \{\,\texttt{parent}, \texttt{ancestor}\,\}}{\Gamma \vdash_E \mathsf{Axis}{::}\texttt{node} : (\kappa \cap \mathbf{A}_E(\tau, \mathsf{Axis})\,,\, \kappa \cap \mathbf{A}_E(\tau\,, \mathsf{Axis}))}\ (\text{T-PareAnceAxis})$$

$$\frac{c_s{\mapsto}(\tau,\kappa) \in \Gamma \qquad \mathsf{Axis} \in \{\texttt{following-sibling}, \texttt{preceding-sibling}\,\}}{\Gamma \vdash_E \mathsf{Axis}{::}\texttt{node} : (\mathbf{A}_E(\tau,\mathsf{Axis})\,,\, (\kappa \backslash \tau) \cup \mathbf{A}_E(\tau,\mathsf{Axis}))}\ (\text{T-FolPreSib})$$

$$\frac{c_s{\mapsto}(\tau,\kappa) \in \Gamma \qquad \mathsf{Axis} \in \{\,\texttt{following}, \texttt{preceding}\,\}}{\Gamma \vdash_E \mathsf{Axis}{::}\texttt{node} : (\,\mathbf{A}_E(\tau,\mathsf{Axis}),\, \mathbf{A}_E(\tau,\mathsf{Axis}) \cup (\mathbf{A}_E(\mathbf{A}_E(\tau,\mathsf{Axis}),\texttt{ancestor}) \cap \kappa))}\ (\text{T-FolPreAxis})$$

$$\frac{c_s{\mapsto}(\tau,\kappa) \in \Gamma \qquad \mathsf{Test} \in \{\,\texttt{tag}, \texttt{node}, \texttt{text}\,\}}{\Gamma \vdash_E \texttt{self}{::}\mathsf{Test} : (\mathbf{T}_E(\tau,\mathsf{Test})\,,\, (\,\kappa \cap \mathbf{A}_E(\mathbf{A}_E(\tau',\mathsf{Test}),\texttt{ancestor})) \cup \mathbf{T}_E(\tau,\mathsf{Test}))}\ (\text{T-Test})$$

$$\frac{(x{:}\Sigma) \in \Gamma}{\Gamma \vdash_E x : \Sigma}\ (\text{T-Var}) \qquad \frac{\Gamma \vdash_E x : \Sigma' \qquad \Gamma[c_s{\mapsto}\Sigma'] \vdash_E \mathsf{Path} : \Sigma}{\Gamma \vdash_E x/\mathsf{Path} : \Sigma}\ (\text{T-VarPath})$$

$$\frac{\Gamma \vdash_E \mathsf{Step} : \Sigma' \qquad \Gamma[c_s{\mapsto}\Sigma'] \vdash_E \mathsf{Path} : \Sigma}{\Gamma \vdash_E \mathsf{Step}/\mathsf{Path} : \Sigma}\ (\text{T-StepPath})$$

$$\frac{\Gamma \vdash_E \mathsf{Step}[\mathsf{Cond}] : \Sigma' \qquad \Gamma[c_s{\mapsto}\Sigma'] \vdash_E \mathsf{Path} : \Sigma}{\Gamma \vdash_E \mathsf{Step}[\mathsf{Cond}]/\mathsf{Path} : \Sigma}\ (\text{T-StepCondPath})$$

$$\frac{\begin{array}{l} c_s \mapsto (\{Y_1,\ldots,Y_n\},\kappa) \in \Gamma \\ \kappa'_i = \kappa \cap \mathbf{A}_E(Y_i, \texttt{ance-or-self}) \end{array} \qquad \begin{array}{c} \Gamma[c_s \mapsto (\{Y_1\},\kappa'_1)] \quad \Gamma \vdash_E \mathsf{Path} : \Sigma^1 \\ \vdots \\ \Gamma[c_s \mapsto (\{Y_n\},\kappa'_n)] \quad \Gamma \vdash_E \mathsf{Path} : \Sigma^n \end{array}}{\Gamma \vdash_E \mathsf{Path} : \bigcup_{\Sigma^i_\tau \neq \varnothing} \Sigma^i}\ (\text{T-UnfoldedPath})$$

$$\frac{\begin{array}{l} c_s \mapsto (\{Y_1,\ldots,Y_n\},\kappa) \in \Gamma \\ \kappa'_i = \kappa \cap \mathbf{A}_E(Y_i, \texttt{ance-or-self}) \end{array} \qquad \begin{array}{c} \Gamma[c_s \mapsto (\{Y_1\},\kappa'_1)] \quad \Gamma \vdash_E \mathsf{Cond} : \Sigma^1 \\ \vdots \\ \Gamma[c_s \mapsto (\{Y_n\},\kappa'_n)] \quad \Gamma \vdash_E \mathsf{Cond} : \Sigma^n \end{array}}{\Gamma \vdash_E \texttt{self}{::}\texttt{node}[\mathsf{Cond}] : \bigcup_{\Sigma^i_\tau \neq \varnothing} \Sigma^i}\ (\text{T-Cond})$$

Table 4.1: Typing Rules for Path Navigation

$$\frac{}{\Gamma \vdash_E () : (\varnothing,\varnothing)} \text{(T-Empty)}$$

$$\frac{\Gamma \vdash_E q_1 : \Sigma_1 \qquad \Gamma \vdash_E q_2 : \Sigma_2}{\Gamma \vdash_E q_1,q_2 : \Sigma_1 \cup \Sigma_2} \text{(T-Concat)}$$

$$\frac{\Gamma \vdash_E \mathtt{q} : \Sigma}{\Gamma \vdash_E \texttt{<a>q} : \Sigma} \text{(T-EltConstr)}$$

$$\frac{\begin{array}{l}\Gamma \vdash_E q_1 : (\{Y_1,\ldots,Y_n\},\kappa) \\ \kappa'_i = \kappa \cap \mathbf{A}_E(Y_i,\texttt{ance-or-self}) \end{array} \qquad \begin{array}{c}\Gamma[x \mapsto (\{Y_1\},\kappa'_1)] \quad \Gamma \vdash_E q_2 : \Sigma^1 \\ \vdots \\ \Gamma[x \mapsto (\{Y_n\},\kappa'_n)] \quad \Gamma \vdash_E q_2 : \Sigma^n \end{array}}{\Gamma \vdash_E \texttt{for } x \texttt{ in } q_1 \texttt{ return } q_2 : \bigcup_{\Sigma^i_\tau \neq \varnothing} \Sigma^i} \text{(T-For)}$$

$$\frac{\Gamma \vdash_E q_1 : \Sigma' \qquad \Gamma[x \mapsto \Sigma'] \vdash_E q_2 : \Sigma}{\Gamma \vdash_E \texttt{let } x = q_1 \texttt{ return } q_2 : \Sigma} \text{(T-Let)}$$

$$\frac{\begin{array}{l} c_s \mapsto (\{Y_1,\ldots,Y_n\},\kappa) \in \Gamma \\ \kappa'_i = \kappa \cap \mathbf{A}_E(Y_i,\texttt{ance-or-self}) \\ \tau' = \{Y_i \mid \Sigma^i_\tau \neq \varnothing\} \end{array} \qquad \begin{array}{c}\Gamma[c_s \mapsto (\{Y_1\},\kappa'_1)] \vdash_E \mathsf{Cond} : \Sigma^1 \\ \vdots \\ \Gamma[c_s \mapsto (\{Y_n\},\kappa'_n)] \vdash_E \mathsf{Cond} : \Sigma^n \\ \text{if } \tau' \neq \varnothing \text{ then } \Gamma \vdash_E \mathtt{q} : \Sigma \text{ else } \Sigma = (\varnothing,\varnothing) \end{array}}{\Gamma \vdash_E \texttt{if Cond then } q_2 : \Sigma} \text{(T-If)}$$

Table 4.2: Typing for FLWR Expressions

$$\frac{(c_s{:}\Sigma) \in \Gamma}{\Gamma \vdash_E \texttt{true} : \Sigma} \text{(T-CondTrue)}$$

$$\frac{}{\Gamma \vdash_E \texttt{false} : (\varnothing,\varnothing)} \text{(T-CondFalse)}$$

$$\frac{\Gamma \vdash_E \mathsf{Cond}_1 : \Sigma^1 \qquad \Gamma \vdash_E \mathsf{Cond}_2 : \Sigma^2}{\Gamma \vdash_E \mathsf{Cond}_1 \vee \mathsf{Cond}_2 : \Sigma^1 \cup \Sigma^2} \text{(T-CondDisj)}$$

$$\frac{\Gamma \vdash_E \mathtt{q} : (\tau,\kappa) \qquad \text{if } \tau \neq \varnothing \text{ then } \Sigma = (\tau,\kappa) \text{ else } \Sigma = \texttt{false}}{\Gamma \vdash_E \mathtt{q} : \Sigma} \text{(T-CondQry)}$$

$$\frac{\Gamma(\mathtt{x}) = (\tau,\kappa) \qquad \text{if } \tau = \varnothing \text{ then } \Sigma = \texttt{true} \text{ else } \Sigma = (\varnothing,\varnothing)}{\Gamma \vdash_E \ominus(\mathtt{x}) : \Sigma} \text{(T-CondIsEmptyVar)}$$

Table 4.3: Typing Rules for Conditions

(T-StepCond) This rule unfold a step typing with filtering predicate in two separated steps.

Below follows the rules for the axis selection that are the core of the type system and are dived between forward and backward axes.

(T-ForwAx) considers $\mathsf{Axis} \in \{\texttt{self}, \texttt{child}, \texttt{descendant}, \texttt{following–sibling}, \texttt{preceding–sibling}\}$. The type of the selected nodes is given by the axis selection names function $\mathbf{A}_E(,)$. The type-context has to contain also the ancestor type-names.

(T-PareAnceAx) with $\mathsf{Axis} \in \{\texttt{parent}, \texttt{ancestor}\}$ then the type of the select nodes is given by the axis selection names function. The type-context has to maintain the type-names ancestors only of the nodes that satisfies the selection, this is ensured by: $\kappa \cap \mathbf{A}_E(\tau, \mathsf{Axis})$

(T-FolPre) with $\mathsf{Axis} = \{\texttt{following}, \texttt{preceding}\}$ the context type is computed considering that the following and preceding selection selects type-name and all the name-types of their ancestor except for the root name of the document because it's an ancestor of any name in the current static-context.

(T-FolPreSib) with $\mathsf{Axis} = \{\texttt{following–sibling}, \texttt{preceding–sibling}\}$ the context type is computed considering that the following and preceding selection selects type-name and all the name-types of their ancestor except for the root name of the document because it's an ancestor of any name in the current static-context.

(T-Test) with $\mathsf{Test} = \{\texttt{tag}, \texttt{node}, \texttt{text}\}$ are discarded from the type-context the ancestors of the nodes that do not satisfies the Test.

(T-Var) The enriched type of a variable bounded states into the static environment.

(T-VarPath) The type of a path navigation starting from a variable bounded is inferred replacing the enriched type of the current static-context with the enriched type of the variable, and the typing the path navigation.

(T-StepPath) The type of a path navigation starting from a step is inferred replacing the enriched type of the current static-context with the enriched

type of the step, and the typing the path navigation.

(T-StepCondPath) The type of a path navigation starting from a step is inferred replacing the enriched type of the current static-context with the enriched type of the step filtered by the condition, and the typing the path navigation.

(T-UnfoldedPath) This rule allows to work on a single type at time.

$$\frac{\begin{array}{ll} & \Gamma[c_s \mapsto (\{Y_1\}, \kappa'_1)] \quad \Gamma \vdash_E \mathsf{Path} : \Sigma^1 \\ c_s \mapsto (\{Y_1, \ldots, Y_n\}, \kappa) \in \Gamma & \vdots \\ \kappa'_i = \kappa \cap \mathbf{A}_E(Y_i, \texttt{ance–or–self}) & \Gamma[c_s \mapsto (\{Y_n\}, \kappa'_n)] \quad \Gamma \vdash_E \mathsf{Path} : \Sigma^n \end{array}}{\Gamma \vdash_E \mathsf{Path} : \bigcup_{\Sigma^i_\tau \neq \varnothing} \Sigma^i}$$

To better understand the rule we study a case based on the DTD in Example (3.2). We want to type the result of the query `child :: Bib` with respect to the following static-context:

$$c_s \mapsto (\{\texttt{Bib}, \texttt{Book}\}, \{\texttt{Bib}, \texttt{Book}\}) \in \Gamma$$

The first thing to note is that we need to restrict the type-context of the static-context in each partial typing with $\kappa'_i = \kappa \cap \mathbf{A}_E(\{Y_i\}, \texttt{ance–or–self})$ because otherwise, giving directly κ in input and using rule T-ForwAx we would have undesired types in the resulting type-context as follows:

$$\Gamma[c_s \mapsto (\{\texttt{Bib}\}, \{\texttt{Bib}, \texttt{Book}\})] \vdash_E \texttt{child} :: \texttt{bib} : (\{\texttt{Bib}\}, \{\texttt{Bib}, \underline{\texttt{Book}}\})$$

Then using the restricted type-context κ'_i the typing is precise:

$$\Gamma[c_s \mapsto (\{\texttt{Bib}\}, \{\texttt{Bib}\})] \vdash_E \texttt{child} :: \texttt{bib} : (\{\texttt{Bib}\}, \{\texttt{Bib}\})$$

Moreover, computing the final type of the query we need to discard the type-context associated with empty types in the partial inferences. For example, let us consider the second partial typing where we apply rule T-ForwAx:

$$\Gamma[c_s \mapsto (\{\texttt{Book}\}, \{\texttt{Bib}, \texttt{Book}\})] \vdash_E \texttt{child} :: \texttt{bib} : (\varnothing, \{\texttt{Bib}, \texttt{Book}\})$$

The type-context $\{\texttt{Bib}, \texttt{Book}\}$ must be discarded. In order to do that in the conclusion of the rule we have a filtered union of enriched type, that ensure precision:

$$\bigcup_{\Sigma^i_\tau \neq \varnothing} \Sigma^i = (\{\texttt{Bib}\}, \{\texttt{Bib}\})$$

Remark 4.22 (A mistake in the original work) *We want to underline that in the original article, while writing with rules that allow to work on a single type at time, the authors didn't consider the problem due to the non restriction of the type-context in each partial evaluation. This means that for the system they proposed does not hold completeness.*

(T-Cond) The rule allows to type a condition working on a single type at a time. The structure of the rule is similar to (T-UnfoldedPath).

Example 4.23 *Considering the DTD proposed in Example (3.2), we want to type the result of the expression* $\texttt{following} :: \texttt{node}[\texttt{book}]$ *with hypothesis* $c_s \mapsto (\{\texttt{Author}\}, \{\texttt{Bib}, \texttt{Book}, \texttt{Author}\}) \in \Gamma$.

$$\dfrac{\dfrac{\dfrac{c_s \mapsto (\{\texttt{Author}\},\{\texttt{Bib},\texttt{Book},\texttt{Author}\}) \in \Gamma}{\Gamma \vdash_E \texttt{following::node} : (\tau',\kappa')} \qquad \dfrac{:}{\Gamma[c_s \mapsto (\tau',\kappa')] \vdash_E \texttt{self::node}[\texttt{book}] : (\tau,\kappa)}}{\Gamma \vdash_E \texttt{following::node/self::node}[\texttt{book}] : (\tau,\kappa)}}{\Gamma \vdash_E \texttt{following::node}[\texttt{book}] : (\tau,\kappa)}$$

We unfolded the typing derivation, and now we focus on the axis selection with predicate typing.

$$\begin{aligned}
\tau' &= \mathbf{A}_E(\texttt{Author}, \texttt{following}) \\
\kappa' &= \mathbf{A}_E(\texttt{Author}, \texttt{following}) \cup \\
&\quad (\mathbf{A}_E(\mathbf{A}_E(\texttt{Author}, \texttt{following}), \texttt{ancestor}) \cap \kappa) \\
&= \{\texttt{Publisher}, \texttt{Price}, \texttt{Bib}, \texttt{Book}, \texttt{Author}, \texttt{Title}, \texttt{Edithor}, \texttt{Last}, \\
&\quad \texttt{First}, \texttt{Affiliation}\} \cup \\
&\quad (\{\texttt{Bib}, \texttt{Book}, \texttt{Author}, \texttt{Edithor}\} \cap \{\texttt{Bib}, \texttt{Book}, \texttt{Author}\}) \\
&= \{\texttt{Publisher}, \texttt{Price}, \texttt{Bib}, \texttt{Book}, \texttt{Author}, \texttt{Title}, \texttt{Edithor}, \texttt{Last}, \\
&\quad \texttt{First}, \texttt{Affiliation}\} \cup \{\texttt{Bib}, \texttt{Book}, \texttt{Author}\} \\
&= \{\texttt{Publisher}, \texttt{Price}, \texttt{Bib}, \texttt{Book}, \texttt{Author}, \texttt{Title}, \texttt{Edithor}, \texttt{Last}, \\
&\quad \texttt{First}, \texttt{Affiliation}\}
\end{aligned}$$

At this point, for each name $Y \in \tau'$ the restricted type-context κ'_i is computed (as in the example for the rule T-UnfoldedPath) and the existential test on tag `book` is performed. Finally we have that the type and the type-context of the result of the query are:

$$\begin{aligned}
\tau &= \{\texttt{Book}\} \\
\kappa &= \{\texttt{Bib}, \texttt{Book}\}
\end{aligned}$$

(T-Empty) The empty query is typed to empty type and empty type-context.

(T-Concat) The typing of a query concatenation is obtained as the union of the enriched types of the subqueries .

(T-EltCreation) The element creation does not add new names to the type of a query because element created are typed with $\perp$.

(T-For) The rule allows to type iteration working on a single type at a time. The structure of the rule is similar to (T-UnfoldedPath).

(T-Let) The type projector of a let clause its the type projector of the second query inferred with the fresh variable x bounded to the type projector of the first query.

(T-If) The rule allows to type a condition working on a single type at a time. The structure of the rule is similar to (T-UnfoldedPath), with the difference that the types of the conditions are used to perform an existence test.

4.4 Soundness

A set of typing rules is sound if preserve all the types of the nodes in the query evaluation.

Definition 4.24 (Consistent static and dynamical environments) *Given a dynamic environment* $\rho = \{x_1 \mapsto \vec{\mathbf{n}}_1, \ldots, \mathtt{x}_n \mapsto \vec{\mathbf{n}}_n\}$ *and a static environment* $\Gamma = \{x_1 \mapsto \Sigma^1, \ldots, \mathtt{x}_n \mapsto \Sigma^n\}$ *we say that* ρ *is consistent with respect to* Γ *if for each variable* $\mathtt{x}_i \in \rho$ *let* $\Sigma^i = \Gamma(\mathtt{x}_i)$ *and* $\vec{\mathbf{n}}_i = \rho(\mathtt{x}_i)$ *then*

1. $\chi(\vec{\mathbf{n}}_i) \subseteq \Sigma^i_\tau$
2. $\chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}}_i \rangle}) \subseteq \Sigma^i_\tau$.

Definition 4.25 (Consistent store and dynamical environment) *A dynamical environment* ρ *is consistent wrt a store* η *iff for each bind* $\mathtt{x}_j \mapsto \vec{\mathbf{n}}_j \in \rho$ *there exists a tree* $t \in \eta$ *such that* $\mathrm{Nodes}(\vec{\mathbf{n}}_j) \subseteq \mathrm{Nodes}(t)$

Definition 4.26 (Evaluation) *An evaluation wrt a DTD (W, E) is a triple $\mathcal{A} = (\eta, \rho, \Gamma)$ composed by a store η valid wrt the DTD (W,E), a dynamical environment ρ consistent wrt η, and a static environment Γ consistent wrt the dynamical environment.*

Lemma 4.27 (Axis selection soundness) *Let $\mathcal{A} = (\eta, \rho, \Gamma)$ an evaluation wrt a DTD (W, E) where $c_d \mapsto \vec{\mathbf{n}} \in \rho$, $c_s \mapsto (\tau, \kappa) \in \Gamma$ and $\tau \supseteq \chi(\vec{\mathbf{n}})$ then*

$$\chi([\![\mathsf{Axis}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \mathbf{A}_E(\tau, \mathsf{Axis})$$

Lemma 4.28 (Test filtering soundness) *Let $\mathcal{A} = (\eta, \rho, \Gamma)$ an evaluation wrt a DTD (W, E) where $c_d \mapsto \vec{\mathbf{n}} \in \rho$, $c_s \mapsto (\tau, \kappa) \in \Gamma$ and $\tau \supseteq \chi(\vec{\mathbf{n}})$ then*

$$\chi([\![\mathsf{Test}]\!]_{\langle \eta, c_d \rangle}) \subseteq \mathbf{T}_E(\tau, \mathsf{Test})$$

Theorem 4.29 (Soundness of type system) *Let $\mathcal{A} = (\eta, \rho, \Gamma)$ an evaluation wrt a DTD (W, E) where $c_d \mapsto \vec{\mathbf{m}} \in \rho$, $c_s \mapsto (\tau', \kappa') \in \Gamma$ and $\tau' \supseteq \chi(\vec{\mathbf{m}})$. If $\Gamma \vdash_E \mathsf{q} : (\tau, \kappa)$ and $\rho \Vdash_\eta \mathsf{q} \Rightarrow \vec{\mathbf{n}}$ then*

1. $\chi(\vec{\mathbf{n}}) \subseteq \tau$
2. $\chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \kappa$

Corollary 4.30 (Empty type implies empty sequence) *Let $\mathcal{A} = (\eta, \rho, \Gamma)$ an evaluation wrt a DTD (W, E), if $\Gamma \vdash_E \mathsf{q} : (\varnothing, \kappa)$ and q doesn't construct new elements then $\rho \Vdash_\eta \mathsf{q} \Rightarrow ()$*

Proof.
If $\tau = \varnothing$ then by Theorem (4.29) of Soundness of Type System we have that if $\chi(\vec{\mathbf{n}}) \subseteq \varnothing$ then $\vec{\mathbf{n}} = ()$ and $\chi(()) \subseteq \varnothing$ □

4.5 Completeness

It turns out that the type system is complete for DTDs that are *–guarded, non-recursive, and parent-unambiguous. Intuitively, a DTD is *–guarded when every union occurring in its productions is guarded by * (or by +), it is non recursive if the maximal depth of all documents validating it is upper bounded, while it is parent-unambiguous if no name types both the parent and a strict ancestor of the parent of another name. Formally, we have the following definition

Definition 4.31 *Let (W, E) be a DTD*

1. *E is *–guarded if for each* $Y \rightarrow l[r] \in E$*, whenever the regular expression contains an union then the union is in the scope of a* $*$ *or a* $+$*.*

2. *E is non-recursive if it is never the case that* $Y \Rightarrow_E^+ Y$*, for any name* $Y \in DN(E)$

3. *E is parent-unambiguous if is never the case that there exists two distinct type-names* Y, Z *such that* $Y \Rightarrow_E X$ *and* $Z \Rightarrow_E X$*.*

Non-recursively and *–guardedness are properties enjoyed by a large number of commonly used DTD s. As stated in [1] they involves a lot of XML Query Use Cases [11]: among the ten DTDs defined in the Use Cases, seven are both non-recursive and *–guarded, one is only *–guarded, one is only non-recursive, and just one does not satisfy either property. Concerning the parent-unambiguous property, although DTDs satisfying this property are less frequent (five on the ten DTDs in [11]), its absence is in practice not very problematic since, as we will see, only the presence of the parent axis may hinder completeness.

To see why completeness does not hold in general consider the following DTD rooted at W and which is recursive and not *–guarded

$$\begin{aligned}
&\texttt{Book} \rightarrow \texttt{book}[\texttt{Name}(\texttt{Author}|\texttt{Editor}), \texttt{Dummy}]\\
&\texttt{Author} \rightarrow \texttt{author}[\texttt{Name}]\\
&\texttt{Editor} \rightarrow \texttt{editor}[\texttt{Name}]\\
&\texttt{Name} \rightarrow \texttt{name}[\texttt{pcdata}]\\
&\texttt{Dummy} \rightarrow \texttt{dummy}[\texttt{Dummy}, \texttt{String}]
\end{aligned}$$

and the following two queries

$$\begin{aligned}
&\mathtt{q}_1 = \texttt{book}[\texttt{author}]/\texttt{editor}\\
&\mathtt{q}_2 = \texttt{book}/\texttt{author}/\texttt{name}/\texttt{parent} :: \texttt{node}\\
&\mathtt{q}_3 = \texttt{book}/\texttt{dummy}/\texttt{parent} :: \texttt{node}
\end{aligned}$$

The type inferred for the first query contains both `Author` and `Editor`. These are useless since the query is always empty. This is due to the non *–guarded union (`Author`|`Editor`): if we had (`Author`|`Author`)$*$ instead, then the query might yield a non-empty result, therefore `Author` and `Editor` must correctly (and completely) be in the query type.

The second query shows the reason why completeness does not hold in presence of parent unambiguous DTD. The precise type of this query should

be `Author`. However, the inferred type is {`Book`, `Author`}. This is because the last step `parent` :: `node` is typed with the context `Book`, `Author`, `Name` and this contains $\mathbf{A}_E(\texttt{Name}, \texttt{parent}) = \{\texttt{Book}, \texttt{Author}\}$. Here {`Name`} is the type for the {`author`} node selected by `child` :: `author` and the $\mathbf{A}_E(,)$ operator assigns it {`Book`, `Author`} as parent type, even if the real parent type for {`Name`} in this case should be {`Author`} . Hence, the intersections operated by the type rule for parent are not powerful enough to guarantee precision for cases like this one.

The third query shows the reason why completeness does not hold in presence of recursion and backward axes (recursion with only forward axes does not pose any problem for completeness). The type of the third query should be `Book`, but instead the type `Book`, `Dummy` is inferred. This is due to the recursion `Dummy` $\rightarrow$ `dummy`[`Dummy`, `String`] since `Dummy` $\Rightarrow_E$ `Dummy`, once `Dummy` is reached it is kept in the inferred type for every backward step.

As stated in [1] the second and third cases suggests that to be extremely precise, instead of sets of names, contexts should rather be sets of chains of names, computed and opportunely managed by the type analysis. However (*i*) managing sets of chains instead of simple sets of names dramatically complicates the treatment, due to recursive axes like descendant, (*ii*) the problem may arise only for queries that use parent axis and the concomitance of parent ambiguity make the event rare in practice, and (*iii*) the loss of precision looks in most cases negligible. Therefore we considered that such a small gain did not justify the increase in complexity needed to handle this case.

4.5.1 Completeness of Type System

Theorem 4.32 (Axis Selection Completeness) *Let $\mathcal{A} = (\eta, \rho, \Gamma)$ an evaluation wrt a DTD (W, E) (a) *–guarded, (b) non-recursive, and (c) parent-unambiguous where $c_d \mapsto \vec{\mathbf{n}} \in \rho$, $c_s \mapsto (\tau, \kappa) \in \Gamma$ and $\tau = \chi(\vec{\mathbf{n}})$ then*

$$\chi(\llbracket \mathsf{Axis} \rrbracket_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \mathbf{A}_E(\tau, \mathsf{Axis})$$

An improvement in the state of the art is that now holds *completeness* on type system also for horizontal axes. In the original work [1] horizontal axes work were approximated by a non complete formulation.

Theorem 4.33 (Test Filtering Completeness) *Let $\mathcal{A} = (\eta, \rho, \Gamma)$ an evaluation wrt a DTD (W, E) (a) *–guarded, (b) non-recursive, and (c) parent-unambiguous where $c_d \mapsto \vec{\mathbf{n}} \in \rho$, $c_s \mapsto (\tau, \kappa) \in \Gamma$ and $\tau = \chi(\vec{\mathbf{n}})$ then*

$$\chi(\llbracket \mathsf{Test} \rrbracket_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \mathbf{T}_E(\tau, \mathsf{Test})$$

Theorem 4.34 (Completeness of Type System) *Let* $\mathcal{A} = (\eta, \rho, \Gamma)$ *an evaluation wrt a DTD* (W, E) *(a) *–guarded, (b) non-recursive, and (c) parent-unambiguous, if* $\Gamma \vdash_E \mathtt{q} : (\tau, \kappa)$ *and* $\rho \Vdash_\eta \mathtt{q} \Mapsto \vec{\mathbf{n}}$*, If* $\Gamma \vdash_E \mathtt{q} : (\tau, \kappa)$ *and* $\rho \Vdash_\eta \mathtt{q} \Mapsto \vec{\mathbf{n}}$ *then*

1. $\chi(\vec{\mathbf{n}}) = \tau$
2. $\chi([\![\mathtt{ance\text{–}or\text{–}self}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \kappa$

A relevant improvement in the state of the art is that now exists a proof of *completeness* of the type system for a language that considers XQuery constructs. We recall another time that the use of the path extraction function implied a lack in formal proofs in the original work [1].

Chapter 5

Type based XML Projection

In this section we use enriched type obtained by the static type system of the previous section to infer type-projectors.
We give an example of projection using Type Projectors. We want to query the XML data tree valid wrt the DTD proposed in Example (3.2) and represented in Figure (5.1a) selecting all the `title` elements with the expression

$$/\texttt{descendant-or-self} :: \texttt{title}$$

The semantics of the query yields to all the nodes of type `Title` that stands in the input document. Despite the fact that the query explicit only the `title` tag, thanks to the availability of types, we are able to recognize the chain of names from the root of the document to the `Title` type and then prune the document in the parsing phase. We want to stress that the pruning based on projection paths [2] does not consider types. In this case pruning would be performed traversing all the document and buffering each path of the tree which would be discarded once arrived on a leaf without find a `title` element.

5.1 Type projectors

As in [1], in order to formalize Type Projectors we need to define particular strings of names called *chains* and ranged over by c.

Definition 5.1 (Chains of Names) *Given a DTD (W, E) and a name Y we define the chains of names rooted in Y as follows*

$$Chains_{(W,E)}(Y) = \{Y\ Z_1 \dots Z_n \mid Y \Rightarrow_E Z_1 \Rightarrow_E \dots \Rightarrow_E Z_n,\ n \geq 0\}$$

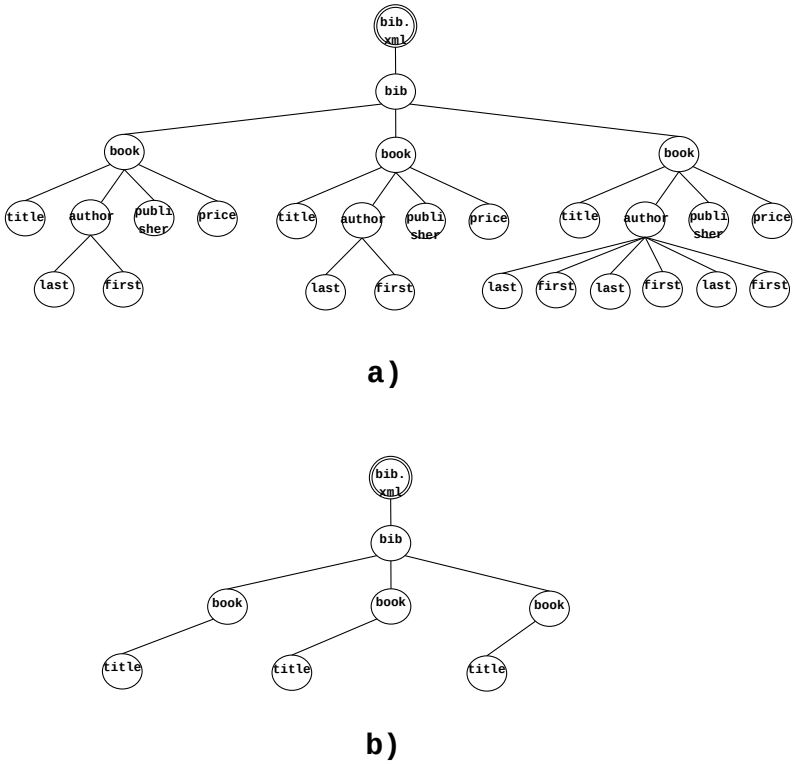

Figure 5.1: a) original document, b) pruned document

And we use Names(c) to denote the set of all names occurring in a chain c.

Definition 5.2 (Type Projectors) *Given a DTD (W, E), a (possibly empty) set of names $\pi \subseteq DN(E)$ is a type projector for (W, E) if and only if there exists $C \subseteq Chains_{(W,E)}(W)$ such that*

$$\pi = \bigcup_{c \in C} \text{Names}(c)$$

A type projector is thus a set of names generated (i.e. reached) by a suite of productions starting from the root of the DTD. For example, it turns out that the type projector for the query /`descendant–or–self` :: `title` over the DTD proposed in Example (3.2) is

$$\pi = \{\texttt{Bib}, \texttt{Book}, \texttt{Title}\}$$

A type projector can be used to pruned a valid tree as follows:

Definition 5.3 (Type Driven Projectors) *Let π be a type projector for (W, E) and t valid wrt the DTD (W, E). The $\pi-$projection of t, noted as $t\backslash_\pi$ is defined as follows:*

$$\begin{array}{lcll}
l_{\mathbf{n}}[f] \setminus \pi & = & l_{\mathbf{n}}[f \setminus \pi] & \mathbf{n}_X \in \pi \\
l_{\mathbf{n}}[f] \setminus \pi & = & () & \mathbf{n}_X \notin \pi \\
s_{\mathbf{n}} \setminus \pi & = & s_{\mathbf{n}} & \mathbf{n}_X \in \pi \\
s_{\mathbf{n}} \setminus \pi & = & () & \mathbf{n}_X \notin \pi \\
(f, f') \setminus \pi & = & (f \setminus \pi,\ f' \setminus \pi) &
\end{array}$$

Given a tree t valid with respect to a DTD (W,E), we use subsets of DN(E) to project that tree. Only nodes that are associated with names in π are kept in the projection. Of course not every subset of names can be used to project a tree, since we want to delete whole subtrees (not nodes in the middle of a tree), and this is the reason why projection is defined in terms of chains as in Definition (5.2).

Definition 5.4 (Projection($\preceq$)) *Given two forests f and f' we say that f' is a projection of f, noted as $f' \preceq f$, if f' is obtained by replacing some subforests of f by the empty forest.*

Proposition 5.5 *Given a tree t valid wrt the DTD (W, E), if π is a type projector for t then $t \setminus \pi$ is a projection of t.*

5.2 Type Projection Inference

Once more naive solution does not work. For instance, for simple paths $\mathsf{Step}_1/\ldots/\mathsf{Step}_n$, we may consider as type projector with respect to (W, E) the set $\bigcup_{i=1\ldots n} \tau_i \cup \{W\}$ where for $i = 1...n$:

$$(\{W\}, \{W\}) \vdash_E \mathsf{Step}_1/\ldots/\mathsf{Step}_n \ : \ (\tau_i, -)$$

(we use "-" as a placeholder for uninteresting parameters). This definition is sound but not precise at all, as can be seen by considering the expression `descendant` :: `node`/`Path` the use of the above union yields a set containing τ_1 defined as

$$(\{W\}, \{W\}) \vdash_E \texttt{descendant} :: \texttt{node} \ : \ (\tau_1, -)$$

that is, all descendants of the root W (no pruning is performed). Instead, we would like to discard, at least, all names that are descendants of W but that are not ancestors of a node matching Path. These are the names $Y \in \mathbf{T}_E(\mathbf{A}_E(W, \texttt{descendant}), \texttt{node})$ such that

$$(\{W\}, \kappa) \vdash_E \texttt{descendant} :: \texttt{node}/\mathsf{Path} \ : \ (\varnothing, -)$$

for some appropriate context κ. A similar reasoning applies to `ancestor`.

Furthermore we have to consider if a subexpression is useful or not for final evaluation of the query. This is important in computing type projectors, because implies to consider or not descendant of inferred types in type-projectors. As in [1] we use a flag m that indicates whether `q` is a query that serves to materialize a partial or final result (and we set $m = 1$), or that just selects a set of nodes whose descendants are not needed ($m = 0$). As example let us consider a bounded variable `x` and the following query:

```
for y in x//author
  return <authorelement> y </authorelement>
```

In one hand the expression `x//author` is used only to iterate, it does not contribute to the final result of the query and then the types of the descendant of nodes typed as `Author` are not useful. In the other hand, the rightmost expression `<authorelement> y </authorelement>` is used as result of the query and then types of the descendant of nodes typed as `Author` are needed to do not prune the subtrees they are contained. Moreover we notice that since element construction is allowed only in the leftmost part of

the query then its content is always useful for the final evaluation, in other words, it's always marked with 1.

Finally, wherever is possible in the type projection rules, we optimize applying Corollary (4.30). We recall that it states that if the type of an expression is empty, and the expression does not construct new elements, than the expression evaluate to the empty sequence. This allow us to discard names from the type projector in soundness.

5.2.1 Description of Rules

In what follows we give a brief description of the type-projection inference rules.

(P-AxTe) The rule unfold the type projection inference for an Axis::Test step in two consecutive steps. Whatever the expression is useful for the final result of the query is decided by the parameter m that is share between the premise and conclusion judgments.

(P-AxTeCond) The rule unfold the type projection inference for Axis::Test[Cond], a step with predicate as filtering condition, in two consecutive steps. Whatever the expression is useful for the final result of the query is decided by the parameter m that is share between the premise and conclusion judgments.

(P-AxNo1) The rule infers type projector for an axis selection in the case that the expression is not useful for the final result of the query. Indeed, as explained before, no descendants of types in τ are taken into the final type projector.

(P-AxNo2) The rule infers type projector for an axis selection in the case that the expression is not useful for the final result of the query. Indeed, as explained before, descendants of types in τ are taken into the final type projector.

(P-SeTe1) The rule infers type projector for a step filtering in the case that the expression is not useful for the final result of the query. Indeed, as explained before, no descendants of types in τ are taken into the final type projector.

(P-Sete2) The rule infers type projector for a step filtering in the case that the

$$\frac{\Gamma \Vdash_E^{[m]} \mathsf{Axis}::\mathtt{node}/\mathtt{self}::\mathsf{Test} : \pi}{\Gamma \Vdash_E^{[m]} \mathsf{Axis}::\mathsf{Test} : \pi}(\text{P-AxTe}) \qquad \frac{\Gamma \Vdash_E^{[m]} \mathsf{Axis}::\mathsf{Test}/\mathtt{self}::\mathtt{node}[\mathsf{Cond}] : \pi}{\Gamma \Vdash_E^{[m]} \mathsf{Axis}::\mathsf{Test}[\mathsf{Cond}] : \pi}(\text{P-AxTeCond})$$

$$\frac{\Gamma \vdash_E \mathsf{Axis}::\mathtt{node} : (\tau,\kappa) \qquad \mathsf{Axis} \in \{\mathtt{child}, \mathtt{des}, \mathtt{parent}, \mathtt{ancestor}\}}{\Gamma \Vdash_E^{[0]} \mathsf{Axis}::\mathtt{node} : \tau \cup \kappa}(\text{P-VAx1})$$

$$\frac{\Gamma \vdash_E \mathsf{Axis}::\mathtt{node} : (\tau,\kappa) \qquad \mathsf{Axis} \in \{\mathtt{child}, \mathtt{des}, \mathtt{parent}, \mathtt{ancestor}\}}{\Gamma \Vdash_E^{[1]} \mathsf{Axis}::\mathtt{node} : \mathbf{A}_E(\tau,\mathtt{descendant\text{–}or\text{–}self}) \cup \kappa}(\text{P-VAx2})$$

$$\frac{c_s \mapsto (\tau',\kappa') \in \Gamma \qquad \Gamma \vdash_E \mathsf{Axis}::\mathtt{node} : (\tau,\kappa) \qquad \mathsf{Axis} \in \{\mathtt{fs}, \mathtt{ps}, \mathtt{following}, \mathtt{preceding}\}}{\Gamma \Vdash_E^{[0]} \mathsf{Axis}::\mathtt{node} : \tau' \cup \kappa' \cup \tau \cup \kappa}(\text{P-HAx1})$$

$$\frac{c_s \mapsto (\tau',\kappa') \in \Gamma \qquad \Gamma \vdash_E \mathsf{Axis}::\mathtt{node} : (\tau,\kappa) \qquad \mathsf{Axis} \in \{\mathtt{fs}, \mathtt{ps}, \mathtt{following}, \mathtt{preceding}\}}{\Gamma \Vdash_E^{[1]} \mathsf{Axis}::\mathtt{node} : \tau' \cup \kappa' \cup \mathbf{A}_E(\tau,\mathtt{descendant\text{–}or\text{–}self}) \cup \kappa}(\text{P-HAx2})$$

$$\frac{\Gamma \vdash_E \mathtt{self}::\mathsf{Test} : (\tau,\kappa)}{\Gamma \Vdash_E^{[0]} \mathtt{self}::\mathsf{Test} : \tau \cup \kappa}(\text{P-SeTe1})$$

$$\frac{\Gamma \vdash_E \mathtt{self}::\mathsf{Test} : (\tau,\kappa)}{\Gamma \Vdash_E^{[1]} \mathtt{self}::\mathsf{Test} : \mathbf{A}_E(\tau,\mathtt{descendant\text{–}or\text{–}self}) \cup \kappa}(\text{P-SeTe2})$$

$$\frac{\Gamma \vdash_E \mathtt{self}::\mathtt{node}[\mathsf{Cond}] : (\tau,\kappa)}{\Gamma \Vdash_E^{[0]} \mathtt{self}::\mathtt{node}[\mathsf{Cond}] : \tau \cup \kappa}(\text{P-SeNoCond1})$$

$$\frac{\Gamma \vdash_E \mathtt{self}::\mathtt{node}[\mathsf{Cond}] : (\tau,\kappa)}{\Gamma \Vdash_E^{[1]} \mathtt{self}::\mathtt{node}[\mathsf{Cond}] : \mathbf{A}_E(\tau,\mathtt{descendant\text{–}or\text{–}self}) \cup \kappa}(\text{P-SeNoCond2})$$

$$\frac{\begin{array}{ll}\Gamma \vdash_E \mathtt{x} : (\{Y_1,\ldots,Y_n\},\kappa) & \tau = \{Y_i | \Sigma_\tau^{/} = \varnothing\} \\ \kappa_i' = \kappa \cap \mathbf{A}_E(\{Y_i\},\mathtt{ance\text{–}or\text{–}self}) & \kappa' = \kappa \cap \mathbf{A}_E(\tau,\mathtt{ance\text{–}or\text{–}self}) \\ \Gamma[c_s \mapsto (\{Y_i\},\kappa_i')] \vdash_E \mathsf{Path} : \Sigma^i & \Gamma[c_s \mapsto (\tau,\kappa')] \Vdash_E^{[m]} \mathsf{Path} : \pi\end{array}}{\Gamma \Vdash_E^{[m]} \mathtt{x}/\mathsf{Path} : \pi}(\text{P-VarPath})$$

$$\frac{\begin{array}{ll}\Gamma \vdash_E \mathsf{Step} : (\{Y_1,\ldots,Y_n\},\kappa) & \tau = \{Y_i | \Sigma_\tau^i \neq \varnothing\} \\ \kappa_i' = \kappa \cap \mathbf{A}_E(\{Y_i\},\mathtt{ance\text{–}or\text{–}self}) & \kappa' = \kappa \cap \mathbf{A}_E(\tau,\mathtt{ance\text{–}or\text{–}self}) \\ \Gamma[c_s \mapsto (\{Y_i\},\kappa_i')] \vdash_E \mathsf{Path} : \Sigma^i & \Gamma[c_s \mapsto (\tau,\kappa')] \Vdash_E^{[m]} \mathsf{Path} : \pi\end{array}}{\Gamma \Vdash_E^{[m]} \mathsf{Step}/\mathsf{Path} : \pi}(\text{P-StepPath})$$

$$\frac{\begin{array}{ll}\Gamma \vdash_E \mathsf{Step}[\mathsf{Cond}] : (\{Y_1,\ldots,Y_n\},\kappa) & \tau = \{Y_i | \Sigma_\tau^i \neq \varnothing\} \\ \kappa_i' = \kappa \cap \mathbf{A}_E(\{Y_i\},\mathtt{ance\text{–}or\text{–}self}) & \kappa' = \kappa \cap \mathbf{A}_E(\tau,\mathtt{ance\text{–}or\text{–}self}) \\ \Gamma[c_s \mapsto (\{Y_i\},\kappa)] \vdash_E \mathsf{Path} : \Sigma^i & \Gamma[c_s \mapsto (\tau,\kappa')] \Vdash_E^{[m]} \mathsf{Path} : \pi\end{array}}{\Gamma \Vdash_E^{[m]} \mathsf{Step}[\mathsf{Cond}]/\mathsf{Path} : \pi}(\text{P-StepCond})$$

$$\frac{\begin{array}{ll} & \Gamma[c_s \mapsto (\{Y_1\},\kappa_1')] \Vdash_E^{[m]} \mathsf{Path} : \pi_1 \\ c_s \mapsto (\{Y_1,\ldots,Y_n\},\kappa) \in \Gamma & \vdots \\ \kappa_i' = \kappa \cap \mathbf{A}_E(\{Y_i\},\mathtt{ance\text{–}or\text{–}self}) & \Gamma[c_s \mapsto (\{Y_n\},\kappa_n')] \Vdash_E^{[m]} \mathsf{Path} : \pi_n\end{array}}{\Gamma \Vdash_E^{[m]} \mathsf{Path} : \bigcup_{i=1..n} \pi_i}(\text{P-Path})$$

Table 5.1: Type Projection Rules for Path Navigation

$$\frac{}{\Gamma \Vdash_E^{[m]} () : \varnothing} \text{(P-Empty)}$$

$$\frac{\Gamma \vdash_E \mathtt{x} : (\tau,\kappa)}{\Gamma \Vdash_E^{[0]} \mathtt{x} : \tau \cup \kappa} \text{(P-Var1)} \qquad \frac{\Gamma \vdash_E \mathtt{x} : (\tau,\kappa)}{\Gamma \Vdash_E^{[1]} \mathtt{x} : \mathbf{A}_E(\tau,\mathtt{descendant\text{–}or\text{–}self}) \cup \kappa} \text{(P-Var2)}$$

$$\frac{\Gamma \Vdash_E^{[m]} \mathtt{q}_1 : \pi_1 \quad \Gamma \Vdash_E^{[m]} \mathtt{q}_2 : \pi_2}{\Gamma \Vdash_E^{[m]} \mathtt{q}_1,\mathtt{q}_2 : \pi_1 \cup \pi_2} \text{(P-Concat)} \qquad \frac{\Gamma \Vdash_E^{[1]} \mathtt{q} : \pi}{\Gamma \Vdash_E^{[1]} \texttt{<a>}\mathtt{q}\texttt{} : \pi} \text{(P-EltConstr)}$$

$$\frac{\begin{array}{ll} \Gamma \Vdash_E^{[0]} \mathtt{q}_1 : \pi' & \\ \Gamma \vdash_E \mathtt{q}_1 : (\{Y_1,\ldots,Y_n\},\kappa) & \tau = \{Y_i | \Sigma_\tau \neq \varnothing\} \\ \kappa'_i = \kappa \cap \mathbf{A}_E(\{Y_i\},\mathtt{ance\text{–}or\text{–}self}) & \kappa' = \kappa \cap \mathbf{A}_E(\tau,\mathtt{ance\text{–}or\text{–}self}) \\ \Gamma[\mathtt{x} : (Y_i,\kappa'_i)] \vdash_E \mathtt{q}_2 : \Sigma^i & \Gamma[\mathtt{x} : (\tau,\kappa')] \Vdash_E^{[m]} \mathtt{q}_2 : \pi \end{array}}{\Gamma \Vdash_E^{[m]} \mathtt{for\ x\ in\ q}_1\ \mathtt{return\ q}_2 : \pi' \cup \pi} \text{(P-For)}$$

$$\frac{\Gamma \Vdash_E^{[0]} \mathtt{q}_1 : \pi' \qquad \Gamma \vdash_E \mathtt{q}_1 : \Sigma \quad \Gamma[\mathtt{x}{:}\Sigma] \Vdash_E^{[m]} \mathtt{q}_2 : \pi}{\Gamma \Vdash_E^{[m]} \mathtt{let\ x} = \mathtt{q}_1\ \mathtt{return\ q}_2 : \pi' \cup \pi} \text{(P-Let)}$$

$$\frac{\Gamma \Vdash_E^{[0]} \mathtt{self::node[Cond]} : \pi' \qquad \text{if } \pi' \neq \varnothing \text{ then } \Gamma \Vdash_E^{[m]} \mathtt{q} : \pi \text{ else } \pi = \varnothing}{\Gamma \Vdash_E^{[m]} \mathtt{if\ Cond\ then\ q} : \pi \cup \pi'} \text{(P-If)}$$

Table 5.2: Type Projection Rules for FLOWR Expressions

expression is not useful for the final result of the query. Indeed, as explained before, descendants of types in τ are taken into the final type projector.

(P-SeNoCond1) The rule infers type projector for a predicate filtering in the case that the expression is not useful for the final result of the query. Indeed, as explained before, no descendants of types in τ are taken into the final type projector.

(P-SeNoCond2) The rule infers type projector for a predicate filtering in the case that the expression is not useful for the final result of the query. Indeed, as explained before, descendants of types in τ are taken into the final type projector.

(P-VarPath) The rule infers type projector for a path navigation starting from a variable bounded. The main idea is to use the enriched type of the bounded variable as enriched type for the static context. The rule use Corollary (4.30) to discard types that does not satisfies the Path navigation. Once discarded all unuseful types the enriched type of the static context is fixed and the type projector is inferenced for the path. Whether the expression is useful for the final result of the query is decided by the parameter m that is share between the premise and conclusion judgments on type projector.

(P-StepPath) The rule infers type projector for a path navigation starting from a step. The main idea is to use the enriched type of the result of the step as enriched type for the static context. The rule use Corollary (4.30) to discard types that does not satisfies the Path navigation. Once discarded all unuseful types the enriched type of the static context is fixed and the type projector is inferenced for the path. Whether the expression is useful for the final result of the query is decided by the parameter m that is share between the premise and conclusion judgments on type projector.

(P-StepCond) The rule infers type projector for a path navigation starting from a step with predicate filtering. The main idea is to use the enriched type of the result of the step as enriched type for the static context. The rule use Corollary (4.30) to discard types that does not satisfies the Path navigation. Once discarded all unuseful types the enriched type of the static context is fixed and the type projector is inferenced for the path. Whether the expression is useful for the final result of the query is decided by the parameter m that is share between the premise and conclusion judgments on type projector.

(P-Path) The rule allows to work with a single type in the current context at a time. We notice that the restriction made on each type context κ'_j is in agree with the type inference rules of Chapter 5, and as explained are needed to achieve precision and do not hinder completeness.

(P-Empty) The type projector for the empty query is always an empty set.

(P-Var1) The type projector for a bounded variable is contained into the store. The rule infers type projector for an axis selection in the case that the expression is not useful for the final result of the query. Indeed, as explained before, no descendants of types in τ are taken into the final type projector.

(P-Var2) The type projector for a bounded variable is contained into the store. The rule infers type projector for an axis selection in the case that the expression is not useful for the final result of the query. Indeed, as explained before, descendants of types in τ are taken into the final type projector.

(P-Concat) The rule infers the type projector for a concatenation of queries as union of the type projectors inferred for each subexpression inductively. Whether the expression is useful for the final result of the query is decided by the parameter m that is share between the premise and conclusion judgments on type projector.

(P-EltConstr) The type projection for an element construction is the type projector of the query nested into the construction. This because no types are associated with constructed elements. Moreover, standing on the fact that element construction is allowed only in the rightmost part of the query then it is always useful to the final result of the query and then $m = 1$.

(P-For) The rule infers type projector for iteration construct. The main idea is to use the enriched type of the evaluation of the first query bounded to variable. The rule works on a single type at time and use Corollary (4.30) to discard all types that raise an empty typing when bounded to the variable in the second query. Once discarded all unuseful types the enriched type of the static context is fixed and the type projector is inferenced for the second query. Whether the expression is useful for the final result of the query is decided by the parameter m that is share between the premise and

conclusion judgments on type projector.

(P-Let) The type projector for the let expression is inferred as the union of the type projector for the first and the second query. The second query is typed updating the static environment with the enriched type of the first query bounded to a variable. We notice that the first query is always not useful to the final result of the let construct while the second query may be depending on the value of m.

(P-If) The rule infers type projector for the if construct. The final type projector is obtained as union of the type projector of both the expressions at the top level of the construct. Since the typing of a condition discards all the type context associated with empty types, and the fact that type projectors are obtained as union of type and relative typecontext, we have that if the type of the query is empty then projector of the query is empty. Then we can use type projector for the existence test and then to decide to type the second query or to return the empty set.

5.3 Soundness

Theorem 5.6 (Soundness of type projection inference) *Given an evaluation $\mathcal{A} = (\eta, \rho, \Gamma)$ wrt a DTD (W,E). If $\Gamma \Vdash_E^{[m]} \mathsf{q} : \pi$ then the following propositions hold.*

1. *π is a type projector for (W, E)*
2. *If $\rho \Vdash_\eta q \Mapsto \vec{\mathbf{n}}; \eta'$ then $\rho \Vdash_{\eta \setminus \pi} q \Mapsto \vec{\mathbf{m}}; \eta''$ and $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$*

The above theorem states that executing the query q on a tree t returns the same set of nodes as executing it on $t \setminus \pi$ the tree t pruned by the inferred projector. From a practical perspective it is important to notice that according to standard XQuery semantics, the semantics of a query contains only the nodes of the result of the query not their sub-trees. The latter may thus be pruned by the inferred projector. Therefore, if we want to materialize the result of a query we must not cut these nodes, and rather use a projector that takes all the descendants of the selected names.

5.4 Precision of the type projection inference

The precision of the type projection inference depends directly from the precision of the type system. We proved that the type system as complete, but

despite this there are some cases where is really difficult to ensure precision of the type projection inference. Following [1], we know that completeness requires also the following conditions on queries:

Definition 5.7 *A query* q *is strongly-specified if*

i) its predicates do not use backward axes,

ii) along q *and along each path in the predicates of* q *there are no two consecutive (possibly conditional) steps whose* Test *part is* node*, and*

iii) each predicate in q *contains at most one path and this does not terminate by a step whose* Test *is* node

As suggested in the original work, these specification of query reflect a very common class of queries. Indeed almost all the path present in XMark benchmarks are strong specified.

Example 5.8 *Considering the following portion of DTD:*

<!ELEMENT bib (book*)>
<!ELEMENT book (title, (author$^+$ | editor$^+$), publisher, price)>

we give some example of both strongly-specified

(1) /descendant :: author/child :: first/ancestor :: node
(2) /descendant :: node/author[child :: first]/
/self :: author/ancestor :: node

and not strongly-specified

(3) /bib[descendant :: node/ancestor :: bib]
(4) /descendant :: author[child :: node]
(5) /descendant :: author[child :: first or child :: last]

Conjecture 5.1 (Completeness of type projection inference) *Given an evaluation* $\mathcal{A} = (\eta, \rho, \Gamma)$ *with respect to a DTD* (W, E) **–guarded, non-recursive and parent unambiguous, and a query* q *strong specified. If* $\Gamma \Vdash_E^{[m]}$ q : π *then there exists a tree valid wrt* (W, E) *such that for each* $\{Y\} \in \pi$ *let* $\pi' = \pi \setminus \mathbf{A}_E(\{Y\}, \mathtt{dos})$ *we have that*

$$\textit{if} \quad \Gamma \Vdash_\eta \ \mathtt{q} \ \Rrightarrow \ \vec{\mathbf{n}}; \eta' \quad \textit{then} \quad \Gamma \Vdash_{\eta_{\setminus \pi'}} \ \mathtt{q} \ \Rrightarrow \ \vec{\mathbf{m}}; \eta' \quad \textit{and} \quad \vec{\mathbf{n}}\neg \cong \vec{\mathbf{m}}$$

Because of lack of time we didn't report the proof of the theorem. Despite this, let us see why completeness does not hold for non strong-specified query considering the DTD in Example (3.2). For example, query (4) does not respect condition *ii)* since it use `node` test twice consecutive. The semantic of the query yields to the set of nodes of type `Author` which has at least one child node. Then the minimal type projector must take `Author`, all the ancestors of nodes typed as `Author` and the types of the children of nodes typed as `Author`. Looking at the DTD, if the document is valid then every node of this kind has at least two nodes that are typed as `First` and `Last`. It's not needed to take both the names, we need only one to be sure that the predicate is satisfied. Despite this the proposed rules of type inference includes both `First` and `Last` thus breaking completeness. Query (3) does not satisfies condition *i)*, it's easy to see that another time the minimal type projector for the query is $\pi = \{\texttt{Bib}, \texttt{Book}\}$ since we need only one node to satisfy the existence test into the predicate. The projection inference conclude with all the types in $DN(E)$ because all of they have `bib` as ancestor. Finally in query (5), that does not satisfies condition *iii)*, it's easy to see that we presence of a disjunction of conditions takes another time type inference to a non minimal set.

Chapter 6

Conclusion and Future works

The main aspect of our work is the formal foundation it provides wrt the state of the art. In the original article the authors didn't treat directly XQuery language because of its complexity, and decided to reduce the problem to XPath optimization using the path extraction function as already explained. With this choose authors loose every chance to have formal proofs of Soundness and Completeness over XQuery of their type system, and in the end they provide only proofs for XPath. In this dissertation we gave a formal foundation of a relevant subset of XQuery with a easy understandable semantics and flexibility to formal proofs. Further, a set of rules for static type inference and type projection inference are provided and formally proved. The theorems of Soundness and Completeness we stated are now a new starting point to develop a series of application of type projectors on the language FLWR-XQuery. In what follows we discuss three possible applications.

- Subquery execution optimization. When dealing with bunches of queries over the same document type projection inference could yield to huge type projectors that do not perform a relevant pruning while loading the document in main memory. Moreover, in this fashion during the execution of a query, and especially in presence of expression that use `descendant–or–self` axis entire not-useful subtrees are visited. To overcome this problem it is possible to develop a static type projection inference system that associate every subexpression with all the types that are touched by the query execution: its proper type projector. This in the query execution phase avoid unuseful reads and also save memory because no unuseful data is loaded in each partial computation. Once computed, type-projectors for subquery can perform

the suggested optimization in two ways. One approach consider the redefinition of the XQuery access plan strategies. Otherwise, it is possible to rewrite the query and explicitly impose a controlled navigation of the document.

- Security access control. An access-control mechanism for XML documents aim to support efficient and secure query access, without revealing sensitive information to unauthorised users. Specifically, for an XML document there may be multiple user groups who want to query the same document. For these use groups different access policies may be imposed, specifying what elements of the document the users are granted to access to. When data are typed by a DTD it is possible to use type projectors to enforce these access policies. Given a query submitted by an user we have, in one hand, thanks to type projectors, the types useful for the query execution, and in the other hand, the rights of the user for access certain data. At this point become easily to ensure that the evaluation of the query overt the document returns only information in that the user is allowed to access. This also a various level of granularity (e.g., restricting access to entire subtrees or specific elements in the document tree based on their content or location).

- Update optimization. Another open issue in this branch of research is the optimization of XML update operations expressed by XQuery. The XQuery Update Facility specification states what kind of updates can be applied to XML documents by taking into account only the effects on the data present in main memory. Issues related to the problem of making updates persistent and efficiently executed are not dealt with, and left to the implementation. Addressing these issues are important especially when the size of XML document to update can become quite large, and update operations can be quite complex. When dealing with typed data, it is possible to use type projectors in order to optimise memory management for XQuery update operations in order to minimise the amount of data to be processed during updates. To this end it is possible to reuse all the static analysis techniques developed in this dissertation to determine, before update processing, what are the piece of data strictly necessary for the update. In this way we avoid managing regions of the input XML documents that are not affected/touched by the updates.

Some works remains open for this document. They are almost related to the approximation of expression that hinder completeness in typing or in type projection inference. We did not prove formally that the approximation of predicates we give in Chapter 3 is sound, despite the fact it seems trivially sound. The most relevant open issue is the proof of completeness of the type projection inference. Another point that we did not treat is the one related to the use of external function that we aspect to work basing on the approximation given in the original article, but the relevance is secondary. Moreover the static analysis proposed must be implemented and tested in order to obtain an important feedback of the work done.

Chapter 7

Proofs

7.1 Soundness of axis selection

Lemma(4.27) (Axis Selection Soundness)
Let $\mathcal{A} = (\eta, \rho, \Gamma)$ an evaluation wrt a DTD (W, E) where $c_d \mapsto \vec{\mathbf{n}} \in \rho$, $c_s \mapsto (\tau, \kappa) \in \Gamma$ and $\tau \supseteq \chi(\vec{\mathbf{n}})$ then

$$\chi([\![\mathsf{Axis}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \mathbf{A}_E(\tau, \mathsf{Axis})$$

Proof by case analysis of Axis.

[Axis = `self`]
By of axis selection we have that

$$\chi([\![\texttt{self}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \chi(\vec{\mathbf{n}}) = \tau = \mathbf{A}_E(\tau, \texttt{self})$$

[Axis = `child`]
By definition of axis selection $[\![\texttt{child}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle} = \bigcup_{\mathbf{n} \in \vec{\mathbf{n}}} \{\mathbf{m} \mid (\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\uparrow}\}$ and since if $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\uparrow}$ implies $\mathbf{n}_X \Rightarrow_E \mathbf{m}_X$ by Lemma (4.4) we have that

$$\begin{array}{rcl} \chi([\![\texttt{child}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) & \subseteq & \bigcup_{\mathbf{n}_X \in \chi(\vec{\mathbf{n}})} \{\mathbf{m}_X \mid \mathbf{n}_X \Rightarrow_E \mathbf{m}_X\} \\ \text{(and by hyp)} & \subseteq & \bigcup_{\mathbf{n}_X \in \tau} \{\, \mathbf{m}_X \mid \mathbf{n}_X \Rightarrow_E \mathbf{m}_X \,\} \\ & = & \mathbf{A}_E(\tau, \texttt{child}) \end{array}$$

[Axis = `parent`]
By definition of axis seletion $[\![\texttt{parent}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle} = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m} \mid (\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\uparrow}\}$ and since if $(\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\uparrow}$ implies $\mathbf{n}_X \Rightarrow_E \mathbf{m}_X$ by Lemma (4.4) we have that

$$\begin{aligned}
\chi([\![\texttt{parent}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) &\subseteq \bigcup_{\mathbf{n}_X\in\chi(\vec{\mathbf{n}})} \{\mathbf{m}_X \mid \mathbf{m}_X \Rightarrow_E \mathbf{n}_X\} \\
\text{and by hypothesis} &\subseteq \bigcup_{\mathbf{n}_X\in\tau} \{\,\mathbf{m}_X \mid \mathbf{m}_X \Rightarrow_E \mathbf{n}_X\,\} \\
&= \mathbf{A}_E(\tau, \texttt{parent})
\end{aligned}$$

[Axis = `descendant`]
By definition of axis seletion $[\![\texttt{descendant}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle} = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m} \mid (\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\uparrow+}\}$ and since if $(\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\uparrow+}$ implies $\mathbf{n}_X \Rightarrow_E^+ \mathbf{m}_X$ by Lemma (4.5)

$$\begin{aligned}
\chi([\![\texttt{descendant}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) &\subseteq \bigcup_{\mathbf{n}_X\in\chi(\vec{\mathbf{n}})} \{\mathbf{m}_X \mid \mathbf{n}_X \Rightarrow_E^+ \mathbf{m}_X\} \\
\text{and by hypothesis} &\subseteq \bigcup_{\mathbf{n}_X\in\tau} \{\,\mathbf{m}_X \mid \mathbf{n}_X \Rightarrow_E^+ \mathbf{m}_X\,\} \\
&= \mathbf{A}_E(\tau, \texttt{descendant})
\end{aligned}$$

[Axis = `ancestor`]
By definition of axis seletion $[\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle} = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m} \mid (\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\uparrow+}\}$ and since if $(\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\uparrow+}$ implies $\mathbf{n}_X \Rightarrow_E^+ \mathbf{m}_X$ by Lemma (4.5) we have that

$$\begin{aligned}
\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) &\subseteq \bigcup_{\mathbf{n}_X\in\chi(\vec{\mathbf{n}})} \{\mathbf{m}_X \mid \mathbf{m}_X \Rightarrow_E^+ \mathbf{n}_X\} \\
\text{and by hypothesis} &\subseteq \bigcup_{\mathbf{n}_X\in\tau} \{\,\mathbf{m}_X \mid \mathbf{m}_X \Rightarrow_E^+ \mathbf{n}_X\,\} \\
&= \mathbf{A}_E(\tau, \texttt{ancestor})
\end{aligned}$$

[Axis = `following–sibling`]
By definition of axis seletion $[\![\texttt{fs}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle} = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m} \mid (\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\rightarrow+}\}$ and since if $(\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\rightarrow+}$ implies $\mathbf{n}_X >_E^+ \mathbf{m}_X$ by Lemma (4.16) we have that

$$\begin{aligned}
\chi([\![\texttt{fs}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) &\subseteq \bigcup_{\mathbf{n}_X\in\chi(\vec{\mathbf{n}})} \{\mathbf{m}_X \mid \mathbf{n}_X >_E^+ \mathbf{m}_X\} \\
\text{and by hypothesis} &\subseteq \bigcup_{\mathbf{n}_X\in\tau} \{\,\mathbf{m}_X \mid \mathbf{n}_X >_E^+ \mathbf{m}_X\,\} \\
&= \mathbf{A}_E(\tau, \texttt{following–sibling})
\end{aligned}$$

[Axis = `preceding–sibling`]
By definition of axis seletion $[\![\texttt{ps}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle} = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m} \mid (\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\rightarrow+}\}$ and since if $(\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\rightarrow+}$ implies $\mathbf{n}_X >_E^+ \mathbf{m}_X$ by Lemma (4.16) we have that

$$\begin{array}{rcl} \chi([\![\texttt{ps}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) & \subseteq & \bigcup\limits_{\mathbf{n}_X\in\chi(\vec{\mathbf{n}})} \{\mathbf{m}_X \mid \mathbf{m}_X >_E^+ \mathbf{n}_X\} \\ \text{and by hypothesis} & \subseteq & \bigcup\limits_{\mathbf{n}_X\in\tau} \{\,\mathbf{m}_X \mid \mathbf{m}_X >_E^+ \mathbf{n}_X\,\} \\ & = & \mathbf{A}_E(\tau, \texttt{preceding–sibling}) \end{array}$$

[Axis = `following`]
By definition of axis seletion $[\![\texttt{f}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle} = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m} \mid (\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\rightarrow\!\!\!\!\rightarrow}\}$ and since if $(\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\rightarrow\!\!\!\!\rightarrow}$ implies $\mathbf{n}_X \gg \mathbf{m}_X$ by Lemma (4.18) we have that

$$\begin{array}{rcl} \chi([\![\texttt{f}]\!]_{\langle\eta,c\rangle}) & \subseteq & \bigcup\limits_{\mathbf{n}_X\in\chi(\vec{\mathbf{n}})} \{\mathbf{m}_X \mid \mathbf{n}_X \gg \mathbf{m}_X\} \\ \text{and by hypothesis} & \subseteq & \bigcup\limits_{\mathbf{n}_X\in\tau} \{\,\mathbf{m}_X \mid \mathbf{n}_X \gg \mathbf{m}_X\,\} \\ & = & \mathbf{A}_E(\tau, \texttt{f}) \end{array}$$

[Axis = `preceding`]
By definition of axis seletion $[\![\texttt{p}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle} = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m} \mid (\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\rightarrow\!\!\!\!\rightarrow}\}$ and since if $(\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\rightarrow\!\!\!\!\rightarrow}$ implies $\mathbf{n}_X \gg \mathbf{m}_X$ by Lemma (4.18) we have that

$$\begin{array}{rcl} \chi([\![\texttt{p}]\!]_{\langle\eta,c\rangle}) & \subseteq & \bigcup\limits_{\mathbf{n}_X\in\chi(\vec{\mathbf{n}})} \{\mathbf{m}_X \mid \mathbf{m}_X \gg \mathbf{n}_X\} \\ \text{and by hypotesis} & \subseteq & \bigcup\limits_{\mathbf{n}_X\in\tau} \{\,\mathbf{m}_X \mid \mathbf{m}_X \gg \mathbf{n}_X\,\} \\ & = & \mathbf{A}_E(\tau, \texttt{preceding}) \quad \square \end{array}$$

7.2 Completeness of axis selection

The main problem related proving completeness is related to the irregularity of the data. For example, given the production

$$Y \rightarrow \mathtt{a}[(Z|X|V)]$$

we have that a possible datamodel instance is the one that contains a node labeled as a and typed as Z, X or V in an exclusive way. Our analysis always infers $\{Z, X, V\}$ when encounter the above content model. Then we need a *—guarded DTD such as

$$Y \rightarrow \mathtt{a}[(Z|X|V)^*]$$

because for each content model (i.e. regular expression over types in $DN(E)$) we want to be possible to build a completion of the tree that justifies the inference of $\{Z, X, V\}$ as complete type.
As little note, for the above content model holds the following relation between each pair of distinguished elements.

Lemma 7.1 (Complete Selection) *Given a DTD (W, E) (a) *–guarded, (b) non-recursive, and (c) parent-unambiguous, there exists a tree t such that for all sequence $\vec{\mathbf{n}}$ of nodes belonging to t*

1. $\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\, \mathbf{m}_X \mid (\mathbf{n}_X, \mathbf{m}_X) \in \varepsilon_t^{\uparrow}\} = \bigcup_{Y\in\chi(\vec{\mathbf{n}})} \{\, \mathbf{m}_X \mid \mathbf{n}_X \Rightarrow_E \mathbf{m}_X\}$
2. $\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\, \mathbf{m}_X \mid (\mathbf{m}_X, \mathbf{n}_X) \in \varepsilon_t^{\uparrow}\} = \bigcup_{Y\in\chi(\vec{\mathbf{n}})} \{\, \mathbf{m}_X \mid \mathbf{m}_X \Rightarrow_E \mathbf{n}_X\}$
3. $\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\, \mathbf{m}_X \mid (\mathbf{n}_X, \mathbf{m}_X) \in \varepsilon_t^{\uparrow+}\} = \bigcup_{Y\in\chi(\vec{\mathbf{n}})} \{\, \mathbf{m}_X \mid \mathbf{n}_X \Rightarrow_E^+ \mathbf{m}_X\}$
4. $\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\, \mathbf{m}_X \mid (\mathbf{m}_X, \mathbf{n}_X) \in \varepsilon_t^{\uparrow+}\} = \bigcup_{Y\in\chi(\vec{\mathbf{n}})} \{\, \mathbf{m}_X \mid \mathbf{m}_X \Rightarrow_E^+ \mathbf{n}_X\}$
5. $\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\, \mathbf{m}_X \mid (\mathbf{n}_X, \mathbf{m}_X) \in \varepsilon_t^{\uparrow*}\} = \bigcup_{Y\in\chi(\vec{\mathbf{n}})} \{\, \mathbf{m}_X \mid \mathbf{n}_X \Rightarrow_E^* \mathbf{m}_X\}$
6. $\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\, \mathbf{m}_X \mid (\mathbf{m}_X, \mathbf{n}_X) \in \varepsilon_t^{\uparrow*}\} = \bigcup_{Y\in\chi(\vec{\mathbf{n}})} \{\, \mathbf{m}_X \mid \mathbf{m}_X \Rightarrow_E^* \mathbf{n}_X\}$

7. $\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\ \mathbf{m}_X \mid (\mathbf{n}_X,\mathbf{m}_X)\ \in\ \varepsilon_t^{\rightarrow}\} = \bigcup_{Y\in\chi(\vec{\mathbf{n}})}\{\ \mathbf{m}_X \mid \mathbf{m}_X >_E \mathbf{n}_X\}$

8. $\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\ \mathbf{m}_X \mid (\mathbf{n}_X,\mathbf{m}_X)\ \in\ \varepsilon_t^{\rightarrow+}\} = \bigcup_{Y\in\chi(\vec{\mathbf{n}})}\{\ \mathbf{m}_X \mid \mathbf{n}_X >_E^+ \mathbf{m}_X\}$

9. $\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\ \mathbf{m}_X \mid (\mathbf{m}_X,\mathbf{n}_X)\in\ \varepsilon_t^{\twoheadrightarrow}\} = \bigcup_{Y\in\chi(\vec{\mathbf{n}})}\{\ \mathbf{m}_X \mid \mathbf{m}_X \gg \mathbf{n}_X\}$

Proof 1) By hypothesis (a) the DTD is *-–guarded, every union is in the scope of a * or a +. This implies that when two types are in relation $Y \Rightarrow_E Z$ then not only there exists a tree t valid wrt (W,E) such that $\mathbf{n},\mathbf{m}\in \mathrm{Nodes}(t)$ such that $(\mathbf{n},\mathbf{m})\in\varepsilon_t^{\uparrow}$, $\mathbf{n}_X = Y$ and $\mathbf{m}_X = Z$, but for each tree t valid wrt (W,E) there can exists two distinct nodes $\mathbf{n},\mathbf{m}$ such that $(\mathbf{n},\mathbf{m})\in\varepsilon_t^{\uparrow}$, $\mathbf{n}_X = Y$ and $\mathbf{m}_X = Z$. This means that from a completeness point of view, given $\mathbf{n}\in\vec{\mathbf{n}}$ we must take all the types $Z\in DN(E)$ such that $\mathbf{n}_X \Rightarrow_E Z$ and then

$$\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\ \mathbf{m}_X \mid (\mathbf{n}_X,\mathbf{m}_X)\ \in\ \varepsilon_t^{\uparrow}\} \ = \ \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\ \mathbf{m}_X \mid \mathbf{n}_X \Rightarrow_E \mathbf{m}_X\}$$

and since nodes with common types yields the set to the same types

$$= \bigcup_{Y\in\chi(\vec{\mathbf{n}})}\{\ \mathbf{m}_X \mid Y \Rightarrow_E \mathbf{m}_X\}$$

Proof 2) Since t is valid wrt (W,E) we have that for each node $\mathbf{n}\in\vec{\mathbf{n}}$ exists at most one node $\mathbf{m}\in\mathrm{Nodes}(t)$ such that $(\mathbf{m},\mathbf{n})\in\varepsilon_t^{\uparrow}$ (since it can be the root). Let us consider two distinct nodes $\mathbf{n}_1,\mathbf{n}_2\in\mathrm{Nodes}(t)$ with the same type. Then there exists at most two nodes $\mathbf{m}_1,\mathbf{m}_2$ such that $(\mathbf{m}_1,\mathbf{n}_1)\in\varepsilon_t^{\uparrow}$ and $(\mathbf{m}_2,\mathbf{n}_2)\in\varepsilon_t^{\uparrow}$. Moreover since by hypothesis (c) the dtd is non-recursive for each $Y\in DN(E)$ $Y \not\Rightarrow_E Y$. Then we have that $\mathbf{m}_1\neq\mathbf{n}_1$ and $\mathbf{m}_2\neq\mathbf{n}_2$. For hypothesis (b) the DTD is also parent-unambiguous: for each $Y\in DN(E)$ there exists at most one $Z\in DN(E)$ such that $Y\Rightarrow_E Z$. This means that since $\mathbf{n}_{1X} = \mathbf{n}_{2X}$ we have that $\mathbf{m}_{1X} = \mathbf{m}_{2X}$. We notice that we don't know if $\mathbf{m}_1 = \mathbf{m}_2$ because they ids can be either different or equal, in every case they have the same type and then

$$\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\ \mathbf{m}_X \mid (\mathbf{m}_X,\mathbf{n}_X)\ \in\ \varepsilon_t^{\uparrow}\} \ = \ \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\ \mathbf{m}_X \mid \mathbf{m}_X \Rightarrow_E Y\}$$

and since nodes with common types yields the set to the same types

$$= \bigcup_{Y\in\chi(\vec{\mathbf{n}})}\{\ \mathbf{m}_X \mid \mathbf{m}_X \Rightarrow_E \mathbf{n}_X\}$$

Proof 3) This proof is by induction on the definition of the relation $\varepsilon_t^{\uparrow+}$. For each pair of nodes $\mathbf{n}, \mathbf{m} \in \text{Nodes}(t)$, $\mathbf{n} \in \vec{\mathbf{n}}$ such that $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\uparrow+}$ there are two main cases:

- $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\uparrow}$ and then by the point 1) of the lemma the thesis follows immediately.
- there exists a node $\mathbf{o}$ such that $(\mathbf{n}, \mathbf{o}) \in \varepsilon_t^{\uparrow+}$ and $(\mathbf{o}, \mathbf{m}) \in \varepsilon_t^{\uparrow}$. By inductive hypothesis on the definition of $\varepsilon_t^{\uparrow+}$ we have that

  $$\{\, \mathbf{o}_X \mid (\mathbf{n}, \mathbf{o}) \in \varepsilon_t^{\uparrow+} \} = \{\, \mathbf{o}_X \mid \mathbf{n}_X \Rightarrow_E^+ \mathbf{o}_X \}$$

  moreover by point 1)

  $$\{\, \mathbf{m}_X \mid (\mathbf{o}, \mathbf{m}) \in \varepsilon_t^{\uparrow} \} = \{\, \mathbf{m}_X \mid \mathbf{o}_X \Rightarrow_E \mathbf{m}_X \}$$

  and then follows

  $$\{\, \mathbf{m}_X \mid (\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\uparrow+} \} = \{\, \mathbf{m}_X \mid \mathbf{n}_X \Rightarrow_E^+ \mathbf{m}_X \}$$

  Moreover we remember that this holds for every $\mathbf{n} \in \vec{\mathbf{n}}$ and that nodes with common types yields the set to the same types, thus obtaining

  $$\bigcup_{\mathbf{n} \in \vec{\mathbf{n}}} \{\, \mathbf{m}_X \mid (\mathbf{n}_X, \mathbf{m}_X) \in \varepsilon_t^{\uparrow+} \} \quad = \quad \bigcup_{Y \in \chi(\vec{\mathbf{n}})} \{\, \mathbf{m}_X \mid \mathbf{n}_X \Rightarrow_E^+ \mathbf{m}_X \}$$

Proof 4) This proof is by induction on the definition of the relation $\varepsilon_t^{\uparrow+}$. For each pair of nodes $\mathbf{n}, \mathbf{m} \in \text{Nodes}(t)$, $\mathbf{n} \in \vec{\mathbf{n}}$ such that $(\mathbf{m}, \mathbf{n}) \in \varepsilon_t^{\uparrow+}$ there are two main cases:

- $(\mathbf{m}, \mathbf{n}) \in \varepsilon_t^{\uparrow}$ and then by the point 1) of the lemma the thesis follows immediately.
- there exists a node $\mathbf{o}$ such that $(\mathbf{m}, \mathbf{o}) \in \varepsilon_t^{\uparrow+}$ and $(\mathbf{o}, \mathbf{n}) \in \varepsilon_t^{\uparrow}$. By inductive hypothesis on the definition of $\varepsilon_t^{\uparrow+}$ we have that

  $$\{\, \mathbf{o}_X \mid (\mathbf{m}, \mathbf{o}) \in \varepsilon_t^{\uparrow+} \} = \{\, \mathbf{o}_X \mid \mathbf{m}_X \Rightarrow_E^+ \mathbf{o}_X \}$$

  moreover by point 1)

  $$\{\, \mathbf{m}_X \mid (\mathbf{o}, \mathbf{n}) \in \varepsilon_t^{\uparrow} \} = \{\, \mathbf{m}_X \mid \mathbf{o}_X \Rightarrow_E \mathbf{n}_X \}$$

and then follows

$$\{\ \mathbf{m}_X \mid (\mathbf{m},\mathbf{n})\ \in\ \varepsilon_t^{\uparrow +}\} = \{\ \mathbf{m}_X \mid \mathbf{m}_X \Rightarrow_E^+ \mathbf{n}_X\}$$

Moreover we remember that this holds for every $\mathbf{n} \in \vec{\mathbf{n}}$ and that nodes with common types yields the set to the same types, thus obtaining

$$\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\ \mathbf{m}_X \mid (\mathbf{m}_X,\mathbf{n}_X)\ \in\ \varepsilon_t^{\uparrow +}\} \quad = \quad \bigcup_{Y\in\chi(\vec{\mathbf{n}})}\{\ \mathbf{m}_X \mid \mathbf{m}_X \Rightarrow_E^+ \mathbf{n}_X\}$$

Proof 5) This proof is by induction on the definition of the relation $\varepsilon_t^{\uparrow *}$. For each pair of nodes $\mathbf{n},\mathbf{m} \in \mathrm{Nodes}(t),\ \mathbf{n} \in \vec{\mathbf{n}}$ such that $(\mathbf{n},\mathbf{m})\ \in\ \varepsilon_t^{\uparrow *}$ there are three main cases:

- $(\mathbf{n},\mathbf{m})\ \in\ \varepsilon_t^{\uparrow}$ and then by the point 1) of the lemma the thesis follows immediately.
- $(\mathbf{n},\mathbf{m})\ \in\ \varepsilon_t^{\uparrow +}$ and then by the point 3) of the lemma the thesis follows immediately.
- $(\mathbf{n},\mathbf{n})\ \in\ \varepsilon_t^{\uparrow *}$ and then the thesis follows immediately since

$$\textstyle\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\ \mathbf{n}_X \mid (\mathbf{n},\mathbf{n})\ \in\ \varepsilon_t^{\uparrow *}\} = \chi(\vec{\mathbf{n}}) = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\ \mathbf{n}_X \mid \mathbf{n} \Rightarrow_E^* \mathbf{n}\}$$

Proof 6) This proof is by induction on the definition of the relation $\varepsilon_t^{\uparrow *}$. For each pair of nodes $\mathbf{n},\mathbf{m} \in \mathrm{Nodes}(t),\ \mathbf{n} \in \vec{\mathbf{n}}$ such that $(\mathbf{m},\mathbf{n})\ \in\ \varepsilon_t^{\uparrow *}$ there are three main cases:

- $(\mathbf{m},\mathbf{n})\ \in\ \varepsilon_t^{\uparrow}$ and then by the point 2) of the lemma the thesis follows immediately.
- $(\mathbf{m},\mathbf{n})\ \in\ \varepsilon_t^{\uparrow +}$ and then by the point 4) of the lemma the thesis follows immediately.
- $(\mathbf{n},\mathbf{n})\ \in\ \varepsilon_t^{\uparrow *}$ and then the thesis follows immediately since

$$\textstyle\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\ \mathbf{n}_X \mid (\mathbf{n},\mathbf{n})\ \in\ \varepsilon_t^{\uparrow *}\} = \chi(\vec{\mathbf{n}}) = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\ \mathbf{n}_X \mid \mathbf{n} \Rightarrow_E^* \mathbf{n}\}$$

Proof 7) By hypothesis (a) the DTD is *-–guarded, by hypothesis (b) is parent–unambiguous and by hypothesi (c) is non recursive. This implies that when two types are in relation $Y >_E Z$ then there for each tree t valid wrt (W,E) there can exists two distinct nodes $\mathbf{n},\mathbf{m}$ such that $(\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\rightarrow}$,

$\mathbf{n}_X = Y$ and $\mathbf{m}_X = Z$. This is because $\varepsilon_t^{\rightarrow}$ is builded upon $\varepsilon_t^{\uparrow}$ and the hypothesis (a-c) are needed to be complete as shown in 1) and 2). This means that for a completeness point of view, given $\mathbf{n} \in \vec{\mathbf{n}}$ we satisfies the equivalence if we take all the types $Z \in DN(E)$ such that $\mathbf{n}_X >_E Z$ and then

$$\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\, \mathbf{m}_X \mid (\mathbf{n}_X, \mathbf{m}_X) \in \varepsilon_t^{\rightarrow}\} \;=\; \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\, \mathbf{m}_X \mid \mathbf{n}_X >_E \mathbf{m}_X\}$$

and since nodes with common types yields the set to the same types

$$= \bigcup_{Y\in\chi(\vec{\mathbf{n}})} \{\, \mathbf{m}_X \mid \mathbf{n}_X >_E \mathbf{m}_X\}$$

Proof 8) This proof is by induction on the definition of the relation $\varepsilon_t^{\rightarrow+}$. For each pair of nodes $\mathbf{n}, \mathbf{m} \in \text{Nodes}(t)$, $\mathbf{n} \in \vec{\mathbf{n}}$ such that $(\mathbf{m}, \mathbf{n}) \in \varepsilon_t^{\rightarrow+}$ there are two main cases:

- $(\mathbf{m}, \mathbf{n}) \in \varepsilon_t^{\rightarrow}$ and then by the point 7) of the lemma the thesis follows immediately.
- there exists a node $\mathbf{o}$ such that $(\mathbf{m}, \mathbf{o}) \in \varepsilon_t^{\rightarrow+}$ and $(\mathbf{o}, \mathbf{n}) \in \varepsilon_t^{\rightarrow}$. By inductive hypothesis on the definition of $\varepsilon_t^{\rightarrow+}$ we have that

  $$\{\, \mathbf{o}_X \mid (\mathbf{m}, \mathbf{o}) \in \varepsilon_t^{\rightarrow+}\} = \{\, \mathbf{o}_X \mid \mathbf{m}_X >_E^+ \mathbf{o}_X\}$$

  moreover by point 1)

  $$\{\, \mathbf{m}_X \mid (\mathbf{o}, \mathbf{n}) \in \varepsilon_t^{\rightarrow}\} = \{\, \mathbf{m}_X \mid \mathbf{o}_X >_E \mathbf{n}_X\}$$

  and then follows

  $$\{\, \mathbf{m}_X \mid (\mathbf{m}, \mathbf{n}) \in \varepsilon_t^{\rightarrow+}\} = \{\, \mathbf{m}_X \mid \mathbf{m}_X >_E \mathbf{n}_X\}$$

  Moreover we remember that this holds for every $\mathbf{n} \in \vec{\mathbf{n}}$ and that nodes with common types yields the set to the same types, thus obtaining

  $$\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\, \mathbf{m}_X \mid (\mathbf{m}_X, \mathbf{n}_X) \in \varepsilon_t^{\rightarrow+}\} \;=\; \bigcup_{Y\in\chi(\vec{\mathbf{n}})} \{\, \mathbf{m}_X \mid \mathbf{m}_X >_E^+ \mathbf{n}_X\}$$

Proof 9) We recall the definition of the relation

$$\varepsilon_t^{\twoheadrightarrow} = \{(\mathbf{n}, \mathbf{m}) \mid \; \exists\, \mathbf{m}, \mathbf{o}\,.\;\; (\mathbf{m}, \mathbf{n}) \in \varepsilon_t^{\uparrow *} \;\; \textit{and} \;\; (\mathbf{m}, \mathbf{o}) \in \varepsilon_t^{\rightarrow+} \;\; \textit{and} \;\; (\mathbf{m}, \mathbf{o}) \in \varepsilon_t^{\uparrow *}\}$$

and then by the point 5), 6) and 8) of the lemma it follows that

$$\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\ \mathbf{m}_X \mid (\mathbf{m},\mathbf{o})\in\varepsilon_t^{\uparrow *} \quad and \quad (\mathbf{o},\mathbf{m})\in\varepsilon_t^{\rightarrow +} \quad and \quad (\mathbf{m},\mathbf{o})\in\varepsilon_t^{\uparrow *}\}$$
$$=\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\ \mathbf{m}_X \mid \mathbf{m}\Rightarrow_E^* \mathbf{o} \quad and \quad \mathbf{o}>_E^+ \mathbf{m} \quad and \quad \mathbf{m}\Rightarrow_E^* \mathbf{o}\}$$

and since nodes with common types yields the set to the same types

$$=\bigcup_{Y\in\chi(\vec{\mathbf{n}})}\{\ \mathbf{m}_X \mid \mathbf{m}\Rightarrow_E^* \mathbf{o} \quad and \quad \mathbf{o}>_E^+ \mathbf{m} \quad and \quad \mathbf{m}\Rightarrow_E^* \mathbf{o}\} \qquad \square$$

Theorem(4.32) (Axis Selection Completeness)
Let $\mathcal{A}=(\eta,\rho,\Gamma)$ an evaluation wrt a DTD (W,E) (a) *–guarded, (b) non-recursive, and (c) parent-unambiguous where $c_d\mapsto\vec{\mathbf{n}}\in\rho$, $c_s\mapsto(\tau,\kappa)\in\Gamma$ and $\tau=\chi(\vec{\mathbf{n}})$ then

$$\chi([\![\mathsf{Axis}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle})=\mathbf{A}_E(\tau,\mathsf{Axis})$$

Proof by case analysis of Axis.
[Axis = `self`]
By definition of axis selection we have that

$$\chi([\![\mathtt{self}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle})=\chi(\vec{\mathbf{n}})=\tau=\mathbf{A}_E(\tau,\mathtt{self})$$

[Axis = `child`]
By definition of axis selection $\chi([\![\mathtt{child}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle})=\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\mathbf{m}_X \mid (\mathbf{n},\mathbf{m})\in\varepsilon_t^{\uparrow}\}$ by hypothesis (a-c) holds Lemma (7.1) Complete Selection and then we have that

$$\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\mathbf{m}_X \mid (\mathbf{n},\mathbf{m})\in\varepsilon_t^{\uparrow}\} = \bigcup_{\mathbf{n}_X\in\chi(\vec{\mathbf{n}})}\{\mathbf{m}_X \mid \mathbf{n}_X\Rightarrow_E \mathbf{m}_X\}$$
$$=\mathbf{A}_E(\tau,\mathtt{child})$$

[Axis = `parent`]
By definition of axis selection $\chi([\![\mathtt{parent}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle})=\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\mathbf{m}_X \mid (\mathbf{m},\mathbf{n})\in\varepsilon_t^{\uparrow}\}$ by hypothesis (a-c) holds Lemma (7.1) Complete Selection and then we have that

$$\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\mathbf{m}_X \mid (\mathbf{m},\mathbf{n})\in\varepsilon_t^{\uparrow}\} = \bigcup_{\mathbf{n}_X\in\chi(\vec{\mathbf{n}})}\{\mathbf{m}_X \mid \mathbf{m}_X\Rightarrow_E \mathbf{n}_X\}$$
$$=\mathbf{A}_E(\tau,\mathtt{parent})$$

[Axis = `descendant`]
By definition of axis selection $\chi([\![\texttt{descendant}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m}_X \mid (\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\uparrow+}\}$ by hypothesis (a-c) holds Lemma (7.1) Complete Selection and then we have that

$$\begin{aligned}\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m}_X \mid (\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\uparrow+}\} &= \bigcup_{\mathbf{n}_X\in\chi(\vec{\mathbf{n}})} \{\mathbf{m}_X \mid \mathbf{n}_X \Rightarrow_E^+ \mathbf{m}_X\} \\ &= \mathbf{A}_E(\tau, \texttt{descendant})\end{aligned}$$

[Axis = `ancestor`]
By definition of axis selection $\chi([\![\texttt{descendant}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m}_X \mid (\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\uparrow+}\}$ by hypothesis (a-c) holds Lemma (7.1) Complete Selection and then we have that

$$\begin{aligned}\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m}_X \mid (\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\uparrow+}\} &= \bigcup_{\mathbf{n}_X\in\chi(\vec{\mathbf{n}})} \{\mathbf{m}_X \mid \mathbf{m}_X \Rightarrow_E^+ \mathbf{n}_X\} \\ &= \mathbf{A}_E(\tau, \texttt{ancestor})\end{aligned}$$

[Axis = `following–sibling`]
By definition of axis selection $\chi([\![\texttt{following–sibling}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m}_X \mid (\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\rightarrow+}\}$ by hypothesis (a-c) holds Lemma (7.1) Complete Selection and then we have that

$$\begin{aligned}\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m}_X \mid (\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\rightarrow+}\} &= \bigcup_{\mathbf{n}_X\in\chi(\vec{\mathbf{n}})} \{\mathbf{m}_X \mid \mathbf{n}_X >_E^+ \mathbf{m}_X\} \\ &= \mathbf{A}_E(\tau, \texttt{following–sibling})\end{aligned}$$

[Axis = `preceding–sibling`]
By definition of axis selection $\chi([\![\texttt{preceding–sibling}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m}_X \mid (\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\rightarrow+}\}$ by hypothesis (a-c) holds Lemma (7.1) Complete Selection and then we have that

$$\begin{aligned}\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m}_X \mid (\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\rightarrow+}\} &= \bigcup_{\mathbf{n}_X\in\chi(\vec{\mathbf{n}})} \{\mathbf{m}_X \mid \mathbf{m}_X >_E^+ \mathbf{n}_X\} \\ &= \mathbf{A}_E(\tau, \texttt{preceding–sibling})\end{aligned}$$

[Axis = `following`]
By definition of axis selection $\chi([\![\texttt{following}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{m}_X \mid (\mathbf{n},\mathbf{m}) \in$

$\varepsilon_t^{\twoheadrightarrow}\}$ by hypothesis (a-c) holds Lemma (7.1) Complete Selection and then we have that

$$\begin{aligned}\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\mathbf{m}_X \mid (\mathbf{n},\mathbf{m})\in\varepsilon_t^{\twoheadrightarrow}\} &= \bigcup_{\mathbf{n}_X\in\chi(\vec{\mathbf{n}})}\{\mathbf{m}_X \mid \mathbf{n}_X \gg \mathbf{m}_X\}\\ &= \mathbf{A}_E(\tau,\texttt{following})\end{aligned}$$

[Axis = `preceding`]
By definition of axis selection $\chi([\![\texttt{preceding}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\mathbf{m}_X \mid (\mathbf{m},\mathbf{n})\in \varepsilon_t^{\twoheadrightarrow}\}$ by hypothesis (a-c) holds Lemma (7.1) Complete Selection and then we have that

$$\begin{aligned}\bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\mathbf{m}_X \mid (\mathbf{m},\mathbf{n})\in\varepsilon_t^{\twoheadrightarrow}\} &= \bigcup_{\mathbf{n}_X\in\chi(\vec{\mathbf{n}})}\{\mathbf{m}_X \mid \mathbf{m}_X \gg \mathbf{n}_X\}\\ &= \mathbf{A}_E(\tau,\texttt{preceding})\end{aligned}$$

Lemma(4.33) (Test Filtering Completeness) Let $\mathcal{A} = (\eta,\rho,\Gamma)$ an evaluation wrt a DTD (W,E) where $c_d \mapsto \vec{\mathbf{n}} \in \rho$, $c_s \mapsto (\tau,\kappa) \in \Gamma$ and $\tau \supseteq \chi(\vec{\mathbf{n}})$ then

$$\chi([\![\mathsf{Test}]\!]_{\langle\eta,c_d\rangle}) \subseteq \mathbf{T}_E(\tau,\mathsf{Test})$$

Proof by case analysis of Test.
[Test = `node`]
By evaluation of test filtering, test filter selection and hypothesis we have that

$$\chi([\![\texttt{node}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) = \chi(\vec{\mathbf{n}}) = \tau = \mathbf{T}_E(\tau,\texttt{node})$$

[Test = *tag*] By semantics of test filtering

$$\begin{aligned}
[\![tag]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle} &= \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{n}_X \mid t@\mathbf{m} = label_{\mathbf{m}}[f],\ t \in \eta\} \\
(t \text{ is valid}) &= \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{n}_X \mid \exists\, Y \rightarrow label[r] \in E,\ and\ \mathbf{n}_X = Y\ \} \\
&= \bigcup_{Y\in\chi(\vec{\mathbf{n}})} \{\mathbf{n}_X \mid \exists\, Y \rightarrow label[r] \in E\} \\
&= \bigcup_{Y\in\tau} \{\mathbf{n}_X \mid \exists\, Y \rightarrow label[r] \in E\} \\
&= \mathbf{T}_E(\tau, tag)
\end{aligned}$$

[Test = `text`] By semantics of test filtering

$$\begin{aligned}
[\![\texttt{text}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle} &= \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{n}_X \mid t@\mathbf{m} = s_{\mathbf{m}},\ t \in \eta\} \\
(t \text{ is valid}) &= \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{n}_X \mid \exists\, Y \rightarrow string[\varepsilon] \in E,\ and\ \mathbf{n}_X = Y\ \} \\
&= \bigcup_{Y\in\chi(\vec{\mathbf{n}})} \{\mathbf{n}_X \mid \exists\, Y \rightarrow string[\varepsilon] \in E\} \\
&= \bigcup_{Y\in\tau} \{\mathbf{n}_X \mid \exists\, Y \rightarrow string[\varepsilon] \in E\} \\
&= \mathbf{T}_E(\tau, \texttt{text})
\end{aligned}$$

7.3 Soundness of Typing Rules

Lemma 7.2 (Containment) *Let $\mathcal{A} = (\eta, \rho, \Gamma)$ be an evaluation wrt a DTD (W, E). If $\Gamma \vdash_E \mathtt{q} : (\tau, \kappa)$ then $\tau \subseteq \kappa$.*

Proof follows checking the typing rules.

Remark 7.3 (Parent Ambiguous and Recursive DTD) *Following the definition of DTD it can happens that two different productions use the same type-name in their content model or introduce a recursive definition. This generates some problems in defining the set of type-names of the parent of a node. Because a type can be reachable from different names or by the name-self. We suppose two distinct nodes that have the same type $\mathbf{o}_X = \mathbf{m}_X$ but are generated from different productions of the DTD. We suppose also that only $\mathbf{m}$ belongs to the current dynamical context. Backward type axes selection selects types of ancestors of both the nodes because they have the same type. This result is sound but not highly precise because types of ancestors of $\mathbf{o}_X$ are not needed since the node is not in the current environment.*

Lemma 7.4 (Soundness of Axis Typing) *Let $\mathcal{A} = (\eta, \rho, \Gamma)$ be an evaluation wrt a DTD (W, E) where $c_d \mapsto \vec{\mathbf{m}} \in \rho$, $c_s \mapsto (\tau', \kappa') \in \Gamma$ and $\chi(\vec{\mathbf{m}}) \subseteq \tau'$. If $\Gamma \vdash_E \mathsf{Axis}::\mathtt{node} : (\tau, \kappa)$ and $\rho \Vdash_\eta \mathsf{Axis}::\mathtt{node} \Rrightarrow \vec{\mathbf{n}}$ then*

1. $\chi(\vec{\mathbf{n}}) \subseteq \tau$
2. $\chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \kappa$

We prove conditions (1) and (2) at the same time by induction on the derivation tree and by case distinction on the last applied rules.

Proof 1)
[T-Forwards, T-FolPreAxis] In this case we have $\tau = \mathbf{A}_E(\tau', \mathsf{Axis})$ and the following hypothesis:

$$\begin{array}{ll} c_d \mapsto \vec{\mathbf{m}} \in \rho & (a) \\ \vec{\mathbf{n}} = \mathrm{docOrder}_\eta([\![\mathsf{Axis}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) & (b) \end{array}$$

and by inductive hypothesis

$$\chi(\vec{\mathbf{m}}) \subseteq \tau' \quad (c)$$

Since serialize simply orders the elements of a set of nodes, we have that

$$\chi(\text{docOrder}_\eta([\![\mathsf{Axis}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle})) = \chi([\![\mathsf{Axis}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle})$$

Finally, using Theorem (4.27) Axis Selection Soundness and hypothesis (b) we have that

$$\chi(\vec{\mathbf{n}}) \subseteq \tau$$

[T-PareAnce] In this case we have that $\tau = \kappa' \cap \mathbf{A}_E(\tau', \mathsf{Axis})$ and the following hypothesis:

$$\begin{array}{ll} c_d \mapsto \vec{\mathbf{m}} \in \rho & (a) \\ \vec{\mathbf{n}} = \text{docOrder}_\eta([\![\mathsf{Axis}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle}) & (b) \end{array}$$

and by inductive hypothesis

$$\chi(\vec{\mathbf{m}}) \subseteq \tau' \quad (c)$$

By Theorem (4.27) Axis Selection Soundness we have that

$$\chi(\text{docOrder}_\eta(\vec{\mathbf{n}})) \subseteq \mathbf{A}_E(\tau', \mathsf{Axis})$$

Then by Remark (7.3) and by hypothesis that all the types of the ancestors of nodes in $\vec{\mathbf{m}}$ are both in κ' and in τ.

$$\chi(\text{docOrder}_\eta(\vec{\mathbf{n}})) \subseteq \kappa' \cap \mathbf{A}_E(\tau', \mathsf{Axis})$$

Proof 2)
For all the cases we have the following hypothesis:

$$\begin{array}{ll} c_s \mapsto (\tau', \kappa') \in \Gamma & (a) \\ c_d \mapsto \vec{\mathbf{m}} \in \rho & (b) \\ \vec{\mathbf{n}} = \text{docOrder}_\eta([\![\mathsf{Axis}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle}) & (c) \end{array}$$

and by inductive hypothesis

$$\begin{array}{ll} \chi(\vec{\mathbf{m}}) \subseteq \tau' & (d) \\ \chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle}) \subseteq \kappa' & (e) \end{array}$$

Then we have some cases depending on the last rule applied and the particular Axis.

[T-Forwards and Axis = `self`]
then following the definitions

$$\vec{\mathbf{n}} = \text{docOrder}_\eta([\![\texttt{self}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle}) = \text{docOrder}_\eta(\vec{\mathbf{m}})$$

and then $\vec{\mathbf{n}} = \vec{\mathbf{m}}$. By typing rule $\kappa = \mathbf{A}_E(\tau', \texttt{self}) \cup \kappa'$, but $\mathbf{A}_E(\tau', \texttt{self}) = \tau'$ and τ' is contained in κ by Lemma of Containment (7.2) then $\kappa = \kappa'$. By hypothesis (e) we have that $\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle}) \subseteq \kappa'$, and the thesis follows immediately for $\vec{\mathbf{n}}$.

[T-Forwards and Axis = `child`]
In this case $\vec{\mathbf{n}} = \text{docOrder}_\eta([\![\texttt{child}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle})$ and by hypothesis:

$$\begin{array}{ll} [\![\texttt{child}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle} = \bigcup_{\mathbf{m}\in\vec{\mathbf{m}}}\{\mathbf{n} | (\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\uparrow}\} & (f) \\ \kappa = \mathbf{A}_E(\tau', \texttt{child}) \cup \kappa' & (g) \end{array}$$

and all the types of ancestors of nodes $\mathbf{n}$ are in κ'. This is, because for each node $\mathbf{n} \in \vec{\mathbf{n}}$ we have by definition of child axis selection that its parent is a node $\mathbf{m} \in \vec{\mathbf{m}}$, and by hypothesis (d) the type of the parent node $\mathbf{m}_X$ belongs to τ', that in turn belongs to κ' by Lemma of Containment (7.2). Moreover by hypothesis (e) all the types of ancestors of nodes in $\vec{\mathbf{m}}$ are in κ' and, since there are no nodes between a node $\mathbf{n}$ and its parent $\mathbf{m}$, as stated in (f), then by transitivity all the types of ancestors of nodes in $\vec{\mathbf{n}}$ are in $\kappa' \cup \mathbf{A}_E(\tau', \texttt{self})$, then by (g)

$$\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) \subseteq \kappa$$

[T-ForwAxis and Axis = `descendant`]
In this case we have that $\vec{\mathbf{n}} = \text{docOrder}_\eta([\![\texttt{descendant}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle})$ and by hypothesis

$$\begin{array}{ll} [\![\texttt{descendant}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle} = \bigcup_{\mathbf{m}\in\vec{\mathbf{m}}}\{\mathbf{n} | (\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\uparrow^+}\} & (f) \\ \kappa = \mathbf{A}_E(\tau', \texttt{descendant}) \cup \kappa' & (g) \end{array}$$

and types of the ancestors of the nodes in $\vec{\mathbf{n}}$ can belong both to $\mathbf{A}_E(\tau', \texttt{descendant})$ and to κ'. This is, because for each node $\mathbf{n} \in \vec{\mathbf{n}}$ by definition of descendant axis selection (f) we know only that *one* of its ancestors, always exists, and it

is a node $\mathbf{m} \in \vec{\mathbf{m}}$. We also know that the type of that node $\mathbf{m}_X$ and the types of all its ancestors are in κ' by Lemma of Containment (7.2) and hypothesis (e) respectively. If $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\uparrow}$ then the proof follows as in the `child` case. If $(\mathbf{n}, \mathbf{m}) \notin \varepsilon_t^{\uparrow}$ then there exists some nodes $\mathbf{m}$ such that $(\mathbf{m}, \mathbf{m}) \in \varepsilon_t^{\uparrow+}$ and $(\mathbf{m}, \mathbf{n}) \in \varepsilon_t^{\uparrow+}$. Each of this nodes $\mathbf{m}$ it is a descendant of $\mathbf{m}$ and then its type $\mathbf{n}'_X$ is contained in $\mathbf{A}_E(\tau', \texttt{descendant})$ by Theorem of Axis Selection Soundness (4.27) and hypothesis (d). Then by (g)

$$\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \kappa$$

[T-PareAnceAxis and Axis = `parent`]
In this case we have that $\vec{\mathbf{n}} = \text{docOrder}_\eta([\![\texttt{parent}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle})$ and by hypothesis

$$\begin{aligned} \kappa &= \kappa' \cap \mathbf{A}_E(\tau', \texttt{ancestor}) && (f) \\ [\![\texttt{parent}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle} &= \bigcup_{\mathbf{n} \in \vec{\mathbf{n}}} \{\mathbf{p} | (\mathbf{p}, \mathbf{n}) \in \varepsilon_t^{\uparrow}\} && (g) \end{aligned}$$

and we rewrite the last term

$$\mathbf{A}_E(\tau', \texttt{ancestor}) = \mathbf{A}_E(\mathbf{A}_E(\tau', \texttt{parent}), \texttt{ancestor}) \cup \mathbf{A}_E(\tau', \texttt{parent})$$

By Theorem of Axis Selection Soundness (4.27) and hypothesis (d) we have that

$$\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \mathbf{A}_E(\mathbf{A}_E(\tau', \texttt{parent}), \texttt{ancestor})$$

Then by Remark (7.3), by Lemma of Containment (7.2) and hypothesis (e) we have that all the types of the ancestors of nodes in $\vec{\mathbf{m}}$ are both in κ' and in τ. Moreover by hypothesis (g) and definition of `ancestor` axis selection

$$\begin{aligned} [\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle} &= \bigcup_{\mathbf{n} \in \vec{\mathbf{n}}} \{\mathbf{p} | (\mathbf{p}, \mathbf{n}) \in \varepsilon_t^{\uparrow+}\} \\ &\subseteq \bigcup_{\mathbf{m} \in \vec{\mathbf{m}}} \{\mathbf{p} | (\mathbf{p}, \mathbf{m}) \in \varepsilon_t^{\uparrow+}\} \\ &= [\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle} \end{aligned}$$

because by definition of parent backward axes selection for each $\mathbf{n} \in \vec{\mathbf{n}}$ always exists $\mathbf{m} \in \vec{\mathbf{m}}$ such that $(\mathbf{m}, \mathbf{n}) \in \varepsilon_t^{\uparrow}$, and by hypothesis (e) we have that $\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) \subseteq \kappa'$. This means that all the types of the ancestors of a node in $\vec{\mathbf{n}}$ are yet in κ'. We have found two supersets of $\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle})$ then

$$\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \kappa$$

[T-ParеAnceAxis and $\mathsf{Axis} = \texttt{ancestor}$]
In this case we have that $\vec{\mathbf{n}} = \text{docOrder}_\eta([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle})$ and

$$\begin{array}{ll} \kappa = \kappa' \cap \mathbf{A}_E(\tau', \texttt{ancestor}) & (f) \\ [\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle} = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{p} | (\mathbf{p},\mathbf{n}) \in \varepsilon_t^{\uparrow+}\} & (g) \end{array}$$

and we rewrite the last term

$$\mathbf{A}_E(\tau', \texttt{ancestor}) = \mathbf{A}_E(\mathbf{A}_E(\tau', \texttt{ancestor}), \texttt{ancestor}) \cup \mathbf{A}_E(\tau', \texttt{ancestor})$$

and by Theorem of Axis Selection Soundness (4.27) and hypothesis (d) we have that

$$\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) \subseteq \mathbf{A}_E(\mathbf{A}_E(\tau', \texttt{ancestor}), \texttt{ancestor})$$

Then by Remark (7.3), by Lemma of Containment (7.2) and hypothesis (e) we have that all the types of the ancestors of nodes in $\vec{\mathbf{m}}$ are both in κ' and in τ. By definition

$$\begin{array}{rcl} [\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle} & = & \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{p} | (\mathbf{p},\mathbf{n}) \in \varepsilon_t^{\uparrow+}\} \\ & \subseteq & \bigcup_{\mathbf{m}\in\vec{\mathbf{m}}} \{\mathbf{p} | (\mathbf{p},\mathbf{m}) \in \varepsilon_t^{\uparrow+}\} \\ & = & [\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle} \end{array}$$

because by definition of backward axes selection for each $\mathbf{n} \in \vec{\mathbf{n}}$ always exists $\mathbf{m} \in \vec{\mathbf{m}}$ such that $(\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\uparrow+}$, and by hypothesis (e) we have that

$$\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle}) \subseteq \kappa'$$

This means that all the types of the ancestors of a node in $\vec{\mathbf{n}}$ are yet in κ'. We have found two supersets of $\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle})$ then

$$\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) \subseteq \kappa$$

[T-ForwAxis and $\mathsf{Axis} = \texttt{following-sibling}$] In this case we have that $\vec{\mathbf{n}} = \text{docOrder}_\eta([\![\mathsf{Axis}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle})$ and by hypothesis

$$\begin{array}{ll} \kappa = \kappa' \cup \mathbf{A}_E(\tau', \mathsf{Axis}) & (f) \\ [\![\texttt{following-sibling}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle} = \bigcup_{\mathbf{m}\in\vec{\mathbf{m}}} \{\mathbf{n} \mid (\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\rightarrow+}\} & (g) \end{array}$$

By hypothesis (e) we have that all the types of the nodes ancestors of nodes in the current dynamic context are in κ'. Moreover by (g) and by the relation $\varepsilon_t^{\rightarrow +}$ we have that $\varepsilon_t^{\uparrow +}(\mathbf{n}) = \varepsilon_t^{\uparrow +}(\mathbf{m})$ and this implies that

$$[\![\texttt{ancestor}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} = [\![\texttt{ancestor}]\!]_{\langle \eta;\vec{\mathbf{m}}\rangle}$$

and then by (f)

$$\chi([\![\texttt{ancestor}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle}) = \kappa$$

[T-ForwAxis and $\mathsf{Axis} = \texttt{preceding–sibling}$] In this case we have that $\vec{\mathbf{n}} = \mathrm{docOrder}_\eta([\![\mathsf{Axis}]\!]_{\langle \eta;\vec{\mathbf{m}}\rangle})$ and by hypothesis

$$\begin{array}{ll} \kappa = \kappa' \cup \mathbf{A}_E(\tau', \mathsf{Axis}) & (f) \\ [\![\texttt{preceding–sibling}]\!]_{\langle \eta;\vec{\mathbf{m}}\rangle} = \bigcup_{\mathbf{m}\in\vec{\mathbf{m}}}\{\mathbf{n} \mid (\mathbf{n},\mathbf{m}) \in \varepsilon_t^{\rightarrow +}\} & (g) \end{array}$$

By hypothesis (e) we have that all the types of the nodes ancestors of nodes in the current dynamic context are in κ'. Moreover by (g) and by the relation $\varepsilon_t^{\rightarrow +}$ we have that $\varepsilon_t^{\uparrow +}(\mathbf{n}) = \varepsilon_t^{\uparrow +}(\mathbf{m})$ and this implies that

$$[\![\texttt{ancestor}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} = [\![\texttt{ancestor}]\!]_{\langle \eta;\vec{\mathbf{m}}\rangle}$$

and then by (f)

$$\chi([\![\texttt{ancestor}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle}) = \kappa$$

[T-FolPreAxis and $\mathsf{Axis} = \{\texttt{following}, \texttt{preceding}\}$] In this case we have that $\vec{\mathbf{n}} = \mathrm{docOrder}_\eta([\![\texttt{following}]\!]_{\langle \eta;\vec{\mathbf{m}}\rangle})$ and

$$\kappa = \mathbf{A}_E(\tau, \mathsf{Axis}) \ \cup \ (\mathbf{A}_E(\mathbf{A}_E(\tau, \mathsf{Axis}), \texttt{ancestor}) \ \cap \ \kappa) \quad (f)$$

We have to show that

$$\chi([\![\texttt{ancestor}]\!] \ \subseteq \ (\mathbf{A}_E(\mathbf{A}_E(\tau, \mathsf{Axis}), \texttt{ancestor}) \ \cap \ \kappa))$$

We have that all the ancestors of types of nodes selected by following axis selection, except the root node of the DTD are in $\tau = \mathbf{A}_E(\tau, \mathsf{Axis})$. The ancestor axis selection wrt to τ selects all the type of the ancestor-nodes, this means that it takes also the type of the root node and other not-useful

types as stated in Remarks (7.3), but the intersection $\tau \cap \kappa'$ gives only the type of the root node of the DTD. In this way we have all the type ancestors of the selected nodes and then

$$\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) = \kappa$$

Lemma 7.5 (Soundness of test filtering typing) *Let an evaluation wrt a DTD* (W, E) $\mathcal{A} = (\eta, \rho, \Gamma)$ *if* $\Gamma \vdash_E \texttt{self}::\mathsf{Test} : (\tau, \kappa)$ *and* $\rho \Vdash_\eta \texttt{self}::\mathsf{Test} \Rightarrow \vec{\mathbf{n}}$ *then*

1. $\chi(\vec{\mathbf{n}}) \subseteq \tau$
2. $\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) \subseteq \kappa$

Proof. 1)

[T-TEST and $\mathsf{Test} \in \{\texttt{node}, \texttt{tag}, \texttt{text}\}$] In this case we have that $\tau = \mathbf{T}_E(\tau', \mathsf{Test})$ and by hypothesis

$$\begin{array}{ll} \Gamma(c_s) = (\tau', \kappa') & (a) \\ \tau = \mathbf{T}_E(\tau', \mathsf{Test}) & (b) \\ \vec{\mathbf{n}} = \mathrm{docOrder}_\eta([\![\mathsf{Test}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) & (c) \end{array}$$

and since serialize simply orders elements $\chi(\mathrm{docOrder}_\eta([\![\mathsf{Test}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle})) = \chi([\![\mathsf{Test}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle})$. Then by Theorem of Test Filtering Soundness (4.28) follows

$$\chi(\mathrm{docOrder}_\eta([\![\mathsf{Test}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle})) \subseteq \tau$$

Proof. 2) By hypothesis $\Gamma(c_s) = (\tau', \kappa')$ and $\rho(c_d) = \vec{\mathbf{m}}$ and we have that

$$\begin{array}{ll} \chi(\vec{\mathbf{m}}) \subseteq \tau' & (a) \\ \chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle}) \subseteq \kappa' & (b) \end{array}$$

In this case we have that $\vec{\mathbf{n}} = \mathrm{docOrder}_\eta([\![\mathsf{Test}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle})$ and

$$\kappa = (\kappa' \cap \mathbf{A}_E(\mathbf{A}_E(\tau', \mathsf{Test}), \texttt{ancestor})) \cup \mathbf{T}_E(\tau, \mathsf{Test}) \quad (c)$$

By Theorem of Axis Selection Soundness (4.27) and Theorem of Test Selection Soundness 4.33 and hypothesis (a) we have that

$$\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) \subseteq \mathbf{A}_E(\mathbf{T}_E(\tau', \mathsf{Test}), \texttt{ancestor})$$

Then by Remark (7.3), by Lemma of Containment (7.2) and hypothesis (b) we have that all the types of the ancestors of nodes in $\vec{\mathbf{m}}$ are in κ' then

$$\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) \subseteq \kappa' \cap \mathbf{A}_E(\mathbf{A}_E(\tau', \mathsf{Test}), \texttt{ancestor}))$$

and then by hypothesis (c)

$$\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) \subseteq \kappa \quad \square$$

Lemma 7.6 (Soundness of condition typing) *Let an evaluation wrt a DTD* (W, E) $\mathcal{A} = (\eta, \rho, \Gamma)$ *if* $\Gamma \vdash_E \mathsf{Cond} : (\tau, \kappa)$ *and* $\rho \Vdash_\eta \mathsf{Cond} \Mapsto \vec{\mathbf{n}}$ *then*

1. $\chi(\vec{\mathbf{n}}) \subseteq \tau$
2. $\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) \subseteq \kappa$

We prove 1) and 2) on the derivation tree and case distinction on the last applied rule.

[T-CondTrue] We have that $\rho(c_d) = \vec{\mathbf{m}}$ and $\Gamma(c_s) = (\tau', \kappa')$ and by hypothesis the environments are consistent. By semantics rule S-CondTrue and T-CondTrue follows that $\chi(\vec{\mathbf{n}}) \subseteq \Sigma_\tau$.

[T-CondFalse] By definition $\chi(()) \subseteq \varnothing$.

[T-CondDisj] By inductive hypothesis on rules S-CondDisj and T-CondDisj $\chi(\vec{\mathbf{n}}_1) \subseteq \Sigma_{\tau_1}$ and $\chi(\vec{\mathbf{n}}_2) \subseteq \Sigma_{\tau_2}$, then $\chi(\vec{\mathbf{n}}_1, \vec{\mathbf{n}}_2) \subseteq \Sigma_{\tau_1} \cup \Sigma_{\tau_2}$

[T-CondQry] By inductive hypothesis on rules S-CondQry and T-CondQry.

[T-CondIsEmptyVar] By consistency of Γ and ρ and inductive hypothesis on rules S-CondIsEmpty and T-CondIsEmpty.

Proof 2)

[T-CondTrue] By hypothesis $\rho(c_d) = \vec{\mathbf{m}}$ and $\Gamma(c_s) = (\tau', \kappa')$ and the environments are consistent. By semantics rule S-CondTrue and T-CondTrue follows that $\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) \subseteq \kappa$.

[T-CondFalse] By definition $\chi([\![\texttt{ancestor}]\!]_{\langle\eta;()\rangle}) \subseteq \varnothing$.

[T-CondDisj] By inductive hypothesis on rules S-CondDisj and T-CondDisj we have that

$$\chi([\![\texttt{ancestor}]\!]_{\langle \eta;\, \vec{\mathbf{n}}_1 \rangle}) \subseteq \Sigma_{\tau_1} \text{ and } \chi([\![\texttt{ancestor}]\!]_{\langle \eta;\, \vec{\mathbf{n}}_2 \rangle}) \subseteq \Sigma_{\tau_2}$$

then

$$\chi([\![\texttt{ancestor}]\!]_{\langle \eta;\, \vec{\mathbf{n}}_1, \vec{\mathbf{n}}_2 \rangle} \subseteq \Sigma_{\tau_1} \cup \Sigma_{\tau_2})$$

[T-CondQry] By inductive hypothesis on rules S-CondQry and T-CondQry.

[T-CondIsEmptyVar] By consistency of Γ and ρ and inductive hypothesis on rules S-CondIsEmpty and T-CondIsEmpty.

Lemma 7.7 (Soundness of path typing) *Let an evaluation wrt a DTD* (W, E) $\mathcal{A} = (\eta, \rho, \Gamma)$ *if* $\Gamma \vdash_E \mathsf{Path} : (\tau, \kappa)$ *and* $\rho \Vdash_\eta \mathsf{Path} \Rightarrow \vec{\mathbf{n}}$ *then*

1. $\chi(\vec{\mathbf{n}}) \subseteq \tau$
2. $\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \kappa$

We prove both the condition by induction on the derivation tree and case distinction on last applied rule.

[T-AxisTest] In this case we have that $\Gamma \vdash_E \mathsf{Axis::Test} : (\tau, \kappa)$ and by inductive hypothesis we have that

$$\begin{array}{ll} \Gamma \vdash_E \mathsf{Axis::node/self::Test} : (\tau, \kappa) & (a) \\ \rho \Vdash_\eta \mathsf{Axis::node/self::Test} \Rightarrow \vec{\mathbf{n}} & (b) \\ \chi(\vec{\mathbf{n}}) \subseteq \tau & (c) \\ \chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \kappa & (d) \end{array}$$

Since $\mathsf{Axis::node/self::Test}$ is equivalent to $\mathsf{Axis::Test}$ we have that $\rho \Vdash_\eta \mathsf{Axis::Test} \Rightarrow \vec{\mathbf{n}}'$ and $\vec{\mathbf{n}} = \vec{\mathbf{n}}'$. By all hypothesis (a-d) both statements (1) and (2) holds.

[T-UnfoldedPath]
In this case we have that when $c_s \mapsto (\{Y_1, \ldots, Y_n\}, \kappa') \in \Gamma$ then

$$\Gamma \vdash_E \mathsf{Path} : (\tau, \kappa)$$

and by inductive hypothesis we have that

$$\begin{array}{lll}
\rho(c_d) = \vec{\mathbf{m}} & & (a)\\
\Gamma(c_s) = (\tau', \kappa') & & (b)\\
\text{for all } i = 1..n & c_s \mapsto (\{Y_1\}, \kappa') \in \Gamma \quad \text{and} \quad \Gamma \vdash_E \mathsf{Path} : (\tau_i, \kappa_i) & (c)\\
\text{for all } i = 1..n & c_d \mapsto \mathbf{m}_i \in \rho \quad \text{and} \quad \rho \Vdash_\eta \mathsf{Path} \Mapsto \vec{\mathbf{n}}_i & (d)\\
\text{for all } i = 1..n & \chi(\vec{\mathbf{n}}_i) \subseteq \tau_i & (e)\\
\text{for all } i = 1..n & \chi([\![\texttt{ancestor}]\!]_{\langle \eta;\, \vec{\mathbf{n}}_i \rangle}) \subseteq \kappa_i & (f)
\end{array}$$

moreover by the typing rule T-UnfoldedPath

$$\begin{array}{ll}
\tau = \bigcup_{i=1..n,\, \tau_i \neq \varnothing} \tau_i & (g)\\
\kappa = \bigcup_{i=1..n,\, \tau_i \neq \varnothing} \kappa_i & (h)
\end{array}$$

and when $c_d \mapsto \mathbf{m} \in \rho$ the evaluation of the query is

$$\rho \Vdash_\eta \mathsf{Path} \Mapsto \vec{\mathbf{n}}$$

where $\vec{\mathbf{n}} = (\vec{\mathbf{n}}_1, \ldots, \vec{\mathbf{n}}_n)$ and then by hypothesis (a-e,g) follows immediately

$$\chi(\vec{\mathbf{n}}) \subseteq \tau$$

and by hypothesis (a-d,f,h) and the fact that if $\tau_i = \varnothing$ then there are no ancestors also

$$[\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle} \subseteq \kappa$$

[T-Cond]
In this case we have that when $c_s \mapsto (\{Y_1, \ldots, Y_n\}, \kappa') \in \Gamma$ then

$$\Gamma \vdash_E \mathsf{Cond} : (\tau, \kappa)$$

and by inductive hypothesis we have that

$$\begin{array}{lll}
\rho(c_d) = \vec{\mathbf{m}} & & (a)\\
\Gamma(c_s) = (\tau', \kappa') & & (b)\\
\text{for all } i = 1..n & c_s \mapsto (\{Y_i\}, \kappa') \in \Gamma \quad \text{and} \quad \Gamma \vdash_E \mathsf{Cond} : (\tau_i, \kappa_i) & (c)\\
\text{for all } i = 1..n & c_d \mapsto \mathbf{m}_i \in \rho \quad \text{and} \quad \rho \Vdash_\eta \mathsf{Cond} \Mapsto \vec{\mathbf{n}}_i & (d)\\
\text{for all } i = 1..n & \chi(\vec{\mathbf{n}}_i) \subseteq \tau_i & (e)\\
\text{for all } i = 1..n & \chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}}_i \rangle}) \subseteq \kappa_i & (f)
\end{array}$$

moreover by the typing rule T-Cond

$$\tau = \bigcup_{i=1..n,\, \tau_i \neq \varnothing} \tau_i \quad (g)$$
$$\kappa = \bigcup_{i=1..n,\, \tau_i \neq \varnothing} \kappa_i \quad (h)$$

and when $c_d \mapsto \mathbf{m} \in \rho$ the evaluation of the query is

$$\rho \Vdash_\eta \mathsf{Path} \Rightarrow \vec{\mathbf{n}}$$

where $\vec{\mathbf{n}} = (\vec{\mathbf{n}}_1, \ldots, \vec{\mathbf{n}}_n)$ and then by hypothesis (a-e,g) follows immediately

$$\chi(\vec{\mathbf{n}}) \subseteq \tau$$

and by hypothesis (a-d,f,h) and the fact that if $\tau_i = \varnothing$ then there are no ancestors also

$$[\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle} \subseteq \kappa$$

[T-StepCond] In this case we have that $\Gamma \vdash_E \mathsf{Step[Cond]} : (\tau, \kappa)$ and by inductive hypothesis we have that

$$\Gamma \vdash_E \mathsf{Step/self::node[Cond]} : (\tau, \kappa) \quad (a)$$
$$\rho \Vdash_\eta \mathsf{Step/self::node[Cond]} \Rightarrow \vec{\mathbf{n}} \quad (b)$$
$$\chi(\vec{\mathbf{n}}) \subseteq \tau \quad (c)$$
$$\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \kappa \quad (d)$$

Since $\mathsf{Step/self::node[Cond]}$ is equivalent to $\mathsf{Step[Cond]}$ we have that $\rho \Vdash_\eta \mathsf{Step[Cond]} \Rightarrow \vec{\mathbf{n}}'$ and $\vec{\mathbf{n}} = \vec{\mathbf{n}}'$. By all hypothesis (a-d) both statements (1) and (2) holds.

[T-StepPath]
By inductive hypothesis on T-StepPath and S-StepPath we have that

$$\Gamma \vdash_E \mathsf{Step} : (\tau', \kappa') \quad (a)$$
$$\rho \Vdash_\eta \mathsf{Step} \Rightarrow \vec{\mathbf{m}} \quad (b)$$
$$\chi(\vec{\mathbf{m}}) \subseteq \tau' \quad (c)$$
$$\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) \subseteq \kappa' \quad (d)$$

$$\Gamma[c_s \mapsto \Sigma'] \vdash_E \mathsf{Path} : (\tau, \kappa) \quad (e)$$
$$\rho[c_d \mapsto \vec{\mathbf{m}}] \Vdash_\eta \mathsf{Path} \Rightarrow \vec{\mathbf{n}} \quad (f)$$
$$\chi(\vec{\mathbf{n}}) \subseteq \Sigma_\tau \quad (g)$$
$$\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle} \subseteq \kappa' \quad (h)$$

and since $\Gamma \vdash_E \mathsf{Step/Path} : (\tau, \kappa)$ and $\rho \Vdash_\eta \mathsf{Step/Path} \Mapsto \vec{\mathbf{n}}$ by hypothesis (a-h) both condition (1) and (2) holds.

[T-StepCondPath]
By inductive hypothesis on T-StepCondPath and S-StepCondPath we have that

$$\begin{array}{ll} \Gamma \vdash_E \mathsf{Step[Cond]} : (\tau', \kappa') & (a) \\ \rho \Vdash_\eta \mathsf{Step[Cond]} \Mapsto \vec{\mathbf{m}} & (b) \\ \chi(\vec{\mathbf{m}}) \subseteq \tau' & (c) \\ \chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) \subseteq \kappa' & (d) \end{array}$$

$$\begin{array}{ll} \Gamma[c_s \mapsto \Sigma'] \vdash_E \mathsf{Path} : (\tau, \kappa) & (e) \\ \rho[c_d \mapsto \vec{\mathbf{m}}] \Vdash_\eta \mathsf{Path} \Mapsto \vec{\mathbf{n}} & (f) \\ \chi(\vec{\mathbf{n}}) \subseteq \Sigma_\tau & (g) \\ \chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle} \subseteq \kappa' & (h) \end{array}$$

and since $\Gamma \vdash_E \mathsf{Step[Cond]/Path} : (\tau, \kappa)$ and $\rho \Vdash_\eta \mathsf{Step[Cond]/Path} \Mapsto \vec{\mathbf{n}}$ by hypothesis (a-h) both condition (1) and (2) holds.

Theorem(4.29) (Soundness of Type System) Let an evaluation wrt a DTD (W, E) $\mathcal{A} = (\eta, \rho, \Gamma)$ if $\Gamma \vdash_E \mathsf{q} : (\tau, \kappa)$ and $\rho \Vdash_\eta \mathsf{q} \Mapsto \vec{\mathbf{n}}$ then

1. $\chi(\vec{\mathbf{n}}) \subseteq \tau$
2. $\chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \kappa$

Proof 1) by induction on the derivation tree and case distinction on the last applied rule.

[T-Empty]
By rules T-Empty and S-Empty we have that

$$\begin{array}{ll} \Gamma \vdash_E () : (\varnothing, \varnothing) & (a) \\ \rho \Vdash_\eta () \Mapsto () & (b) \end{array}$$

and since $\chi(()) \subseteq \varnothing$ follows immediately both conditions (1) and (2).

[T-Concat]
In this case we have that the enriched type is computed as union of the two enriched types of the sub-expressions $\Sigma = \Sigma^1 \cup \Sigma^2$. By inductive hypothesis

we have that

$$\begin{array}{lll}
\Gamma \vdash_E q_1 : \Sigma^1 \quad and \quad \rho \Vdash_\eta q_1 \Rrightarrow \vec{\mathbf{n}}_1 & (a) \\
\Gamma \vdash_E q_2 : \Sigma^2 \quad and \quad \rho \Vdash_\eta q_2 \Rrightarrow \vec{\mathbf{n}}_2 & (b) \\
\chi(\vec{\mathbf{n}}_1) \subseteq \Sigma^1_\tau & (c) \\
\chi(\vec{\mathbf{n}}_2) \subseteq \Sigma^2_\tau & (d) \\
\chi([\![\texttt{ancestor}]\!]_{\langle \eta, \vec{\mathbf{n}}_1 \rangle}) \subseteq \Sigma^1_\kappa & (e) \\
\chi([\![\texttt{ancestor}]\!]_{\langle \eta, \vec{\mathbf{n}}_1 \rangle}) \subseteq \Sigma^2_\kappa & (f)
\end{array}$$

then by hypothesis (a-f), rule T-Concat and S-Concat follows immediately both conditions (1) and (2).

[T-EltConstruction]
By rule (S-EltConstr) $\rho \Vdash_\eta$ <a>q $\Rrightarrow \mathbf{n}_a$ where $\mathbf{n}_{aX} = \bot$ because element construction assigned it null type, and it has no ancestors since it's done in the rightmost part of the query both conditions (1) and (2) follows immediately.

[T-Var]
By hypothesis on the evaluation $\mathcal{A}$ the variable x is bounded to an enriched type that satisfies both conditions (1) and (2).

[T-VarPath]
By inductive hypothesis on T-VarPath and S-VarPath we have that

$$\begin{array}{ll}
\Gamma \vdash_E \texttt{x} : (\tau', \kappa') & (a) \\
\rho \Vdash_\eta \texttt{x} \Rrightarrow \vec{\mathbf{m}} & (b) \\
\chi(\vec{\mathbf{m}}) \subseteq \tau' & (c) \\
\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) \subseteq \kappa' & (d) \\
 & \\
\Gamma[c_s \mapsto \Sigma'] \vdash_E \mathsf{Path} : (\tau, \kappa) & (e) \\
\rho[c_d \mapsto \vec{\mathbf{m}}] \Vdash_\eta \mathsf{Path} \Rrightarrow \vec{\mathbf{n}} & (f) \\
\chi(\vec{\mathbf{n}}) \subseteq \Sigma_\tau & (g) \\
\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle} \subseteq \kappa' & (h)
\end{array}$$

and since $\Gamma \vdash_E$ x/Path : (τ, κ) and $\rho \Vdash_\eta$ x/Path $\Rrightarrow \vec{\mathbf{n}}$ by hypothesis (a-h) both condition (1) and (2) holds.

[T-For]

By inductive hypotheses on rules T-For and S-For we have that

$$\begin{array}{ll}
\Gamma[c_s \mapsto \Sigma] \vdash_E q_1 : (\{Y_1, \ldots, Y_n\}, \kappa') & (a)\\
\rho \Vdash_\eta q_1 \Rightarrow \vec{\mathbf{m}} & (b)\\
\chi(\vec{\mathbf{m}}) \subseteq \{Y_1, \ldots, Y_n\} & (c)\\
\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle} \subseteq \kappa' & (d)
\end{array}$$

$$\begin{array}{ll}
\text{for each } Y_j \quad \Gamma[c_s \mapsto (\{Y_j\}, \kappa')] \vdash_E \mathtt{q}_2 : \Sigma^j & (e)\\
\text{for each } \mathbf{m}_j \in \vec{\mathbf{m}} \quad \rho[c_d \mapsto \mathbf{m}_j] \Vdash_\eta q_2 \Rightarrow \vec{\mathbf{n}}_j & (f)\\
\text{for each } j = 1..|\vec{\mathbf{m}}| \quad \chi(\vec{\mathbf{n}}_j) \subseteq \Sigma^j_\tau & (g)\\
\text{for each } j = 1..|\vec{\mathbf{m}}| \quad \chi([\![\texttt{ancestor}]\!]_{\langle \eta, \vec{\mathbf{n}}_j \rangle}) & (h)
\end{array}$$

then by hypothesis (a-h) follows that both conditions (1) and (2) holds.

[T-Let]
By inductive hypotheses on rules T-LetQry and S-LetQry we have that

$$\begin{array}{ll}
\Gamma \vdash_E q_1 : (\tau', \kappa') & (a)\\
\rho \Vdash_\eta q_1 \Rightarrow \vec{\mathbf{m}} & (b)\\
\chi(\vec{\mathbf{m}}) \subseteq \tau' & (c)\\
\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle} \subseteq \kappa' & (d)
\end{array}$$

$$\begin{array}{ll}
\Gamma[\mathtt{x} \mapsto \tau'] \vdash_E \mathtt{q}_2 : (\tau, \kappa) & (e)\\
\rho[\mathtt{x} \mapsto \mathbf{m}] \Vdash_\eta q_2 \Rightarrow \vec{\mathbf{n}} & (f)\\
\chi(\vec{\mathbf{n}}) \subseteq \tau & (g)\\
\chi([\![\texttt{ancestor}]\!]_{\langle \eta, \vec{\mathbf{n}} \rangle}) \subseteq \kappa & (h)
\end{array}$$

then by hypothesis (a-h) follows that both conditions (1) and (2) holds.

[T-If]
By inductive hypotheses on rules T-If and S-If we have that

$$\begin{array}{ll}
\text{for all } i = 1..n \quad c_s \mapsto (\{Y_i\}, \kappa') \in \Gamma \quad \text{and} \quad \Gamma \vdash_E \mathsf{Cond} : (\tau_i, \kappa_i) & (c)\\
\text{for all } i = 1..n \quad c_d \mapsto \mathbf{m}_i \in \rho \quad \text{and} \quad \rho \Vdash_\eta \mathsf{Cond} \Rightarrow \vec{\mathbf{n}}_i & (d)\\
\text{for all } i = 1..n \quad \chi(\vec{\mathbf{n}}_i) \subseteq \tau_i & (e)\\
\text{for all } i = 1..n \quad \chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}}_i \rangle}) \subseteq \kappa_i & (f)
\end{array}$$

moreover by the typing rule T-If

$$\tau = \bigcup_{i=1..n, \tau_i \neq \varnothing} \tau_i \quad (g)$$

and

$$\text{if } \tau' \neq \varnothing \text{ then } \Gamma \vdash_E \mathtt{q} : \Sigma \text{ else } \Sigma = (\varnothing, \varnothing) \quad (h)$$

where Σ is the result of the typing and satisfies both conditions (1) and (2).

7.4 Completeness of Typing Rules

Lemma(4.34) (Completeness of Axis Typing) Let $\mathcal{A} = (\eta, \rho, \Gamma)$ an evaluation wrt a DTD (W, E) (a) *–guarded, (b) non-recursive, and (c) parent-unambiguous, if $\Gamma \vdash_E \mathtt{q} : (\tau, \kappa)$ and $\rho \Vdash_\eta \mathtt{q} \Rightarrow \vec{\mathbf{n}}$ then

1. $\chi(\vec{\mathbf{n}}) = \tau$
2. $\chi([\![\mathtt{ance\text{–}or\text{–}self}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \kappa$

We prove conditions (1) and (2) by induction on the derivation tree and by case distinction on the last applied rules.

Proof 1)
[T-Forwards, T-FolPreAxis] In this case we have $\tau = \mathbf{A}_E(\tau', \mathsf{Axis})$ and the following hypothesis:

$$\begin{array}{ll} c_d \mapsto \vec{\mathbf{m}} \in \rho & (a) \\ \vec{\mathbf{n}} = \mathrm{docOrder}_\eta([\![\mathsf{Axis}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) & (b) \end{array}$$

and by inductive hypothesis

$$\chi(\vec{\mathbf{m}}) = \tau' \quad (c)$$

Since serialize simply orders the elements of a set of nodes, we have that

$$\chi(\mathrm{docOrder}_\eta([\![\mathsf{Axis}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle})) = \chi([\![\mathsf{Axis}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle})$$

Finally, using Theorem (4.32) Axis Selection Completeness and hypothesis (b,c) we have that

$$\chi(\vec{\mathbf{n}}) = \tau$$

[T-PareAnce] In this case we have that $\tau = \kappa' \cap \mathbf{A}_E(\tau', \mathsf{Axis})$ and the following hypothesis:

$$\begin{array}{ll} c_d \mapsto \vec{\mathbf{m}} \in \rho & (a) \\ \vec{\mathbf{n}} = \mathrm{docOrder}_\eta([\![\mathsf{Axis}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) & (b) \end{array}$$

and by inductive hypothesis

$$\begin{array}{ll} \chi(\vec{\mathbf{m}}) = \tau' & (c) \\ \chi(\mathbf{A}_E(\vec{\mathbf{m}}, \mathtt{ance\text{–}or\text{–}self})) = \kappa' & (c) \end{array}$$

By Theorem (4.32) Axis Selection Completeness we have that

$$\chi(\text{docOrder}_\eta(\vec{\mathbf{n}})) = \mathbf{A}_E(\tau', \mathsf{Axis})$$

Then by Remark (7.3) and by hypothesis that all the types of the ancestors of nodes in $\vec{\mathbf{m}}$ are both in κ' and in τ.

$$\chi(\text{docOrder}_\eta(\vec{\mathbf{n}})) = \kappa' \cap \mathbf{A}_E(\tau', \mathsf{Axis})$$

Proof 2)
For all the cases we have the following hypothesis:

$$\begin{array}{ll} c_s \mapsto (\tau', \kappa') \in \Gamma & (a) \\ c_d \mapsto \vec{\mathbf{m}} \in \rho & (b) \\ \vec{\mathbf{n}} = \text{docOrder}_\eta([\![\mathsf{Axis}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) & (c) \end{array}$$

and by inductive hypothesis

$$\begin{array}{ll} \chi(\vec{\mathbf{m}}) = \tau' & (d) \\ \chi([\![\texttt{ance–or–self}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) = \kappa' & (e) \end{array}$$

Then we have some cases depending on the last rule applied and the particular Axis.

[T-Forwards and $\mathsf{Axis} = \texttt{self}$]
then following the definitions

$$\vec{\mathbf{n}} = \text{docOrder}_\eta([\![\texttt{self}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) = \text{docOrder}_\eta(\vec{\mathbf{m}})$$

and then $\vec{\mathbf{n}} = \vec{\mathbf{m}}$. By typing rule $\kappa = \mathbf{A}_E(\tau', \texttt{self}) \cup \kappa'$, but $\mathbf{A}_E(\tau', \texttt{self}) = \tau'$ and τ' is contained in κ by Lemma of Containment (7.2) then $\kappa = \kappa'$. By hypothesis (e) we have that $\chi([\![\texttt{ance–or–self}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) = \kappa'$, and the thesis follows immediately for $\vec{\mathbf{n}}$.

[T-Forwards and $\mathsf{Axis} = \texttt{child}$]
In this case $\vec{\mathbf{n}} = \text{docOrder}_\eta([\![\texttt{child}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle})$ and by hypothesis:

$$\begin{array}{ll} [\![\texttt{child}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle} = \bigcup_{\mathbf{m} \in \vec{\mathbf{m}}} \{\mathbf{n} | (\mathbf{m}, \mathbf{n}) \in \varepsilon_t^{\uparrow}\} & (f) \\ \kappa = \mathbf{A}_E(\tau', \texttt{child}) \cup \kappa' & (g) \end{array}$$

and all the types of ancestors of nodes $\mathbf{n}$ are in κ'. This is, because for each node $\mathbf{n} \in \vec{\mathbf{n}}$ we have by definition of child axis selection that its parent is a node $\mathbf{m} \in \vec{\mathbf{m}}$, and by hypothesis (d) the type of the parent node $\mathbf{m}_X$ belongs to τ', that in turn belongs to κ' by Lemma of Containment (7.2). Moreover by hypothesis (e) all the types of ancestors of nodes in $\vec{\mathbf{m}}$ are in κ' and, since there are no nodes between a node $\mathbf{n}$ and its parent $\mathbf{m}$, as stated in (f), then by transitivity all the types of ancestor-or-self nodes in $\vec{\mathbf{n}}$ are in $\kappa' \cup \mathbf{A}_E(\tau', \texttt{child})$, then by (g)

$$\chi([\![\texttt{ance–or–self}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle}) = \kappa$$

[T-ForwAxis and Axis = `descendant`]
In this case we have that $\vec{\mathbf{n}} = \text{docOrder}_\eta([\![\texttt{descendant}]\!]_{\langle \eta;\vec{\mathbf{m}}\rangle})$ and by hypothesis

$$\begin{array}{ll} [\![\texttt{descendant}]\!]_{\langle \eta;\vec{\mathbf{m}}\rangle} = \bigcup_{\mathbf{m}\in\vec{\mathbf{m}}}\{\mathbf{n}|(\mathbf{m},\mathbf{n}) \in \varepsilon_t^{\uparrow +}\} & (f) \\ \kappa = \mathbf{A}_E(\tau', \texttt{descendant}) \cup \kappa' & (g) \end{array}$$

and types of the ancestors of the nodes in $\vec{\mathbf{n}}$ can belong both to $\mathbf{A}_E(\tau', \texttt{descendant})$ and to κ'. This is, because for each node $\mathbf{n} \in \vec{\mathbf{n}}$ by definition of descendant axis selection (f) we know only that *one* of its ancestors, always exists, and it is a node $\mathbf{m} \in \vec{\mathbf{m}}$. We also know that the type of that node $\mathbf{m}_X$ and the types of all its ancestors are in κ' by Lemma of Containment (7.2) and hypothesis (e) respectively. If $(\mathbf{n}, \mathbf{m}) \in \varepsilon_t^{\uparrow}$ then the proof follows as in the `child` case. If $(\mathbf{n}, \mathbf{m}) \notin \varepsilon_t^{\uparrow}$ then there exists some nodes $\mathbf{m}$ such that $(\mathbf{m}, \mathbf{m}) \in \varepsilon_t^{\uparrow +}$ and $(\mathbf{m}, \mathbf{n}) \in \varepsilon_t^{\uparrow +}$. Each of this nodes $\mathbf{m}$ it is a descendant of $\mathbf{m}$ and then its type $\mathbf{n}'_X$ is contained in $\mathbf{A}_E(\tau', \texttt{descendant})$ by Theorem of Axis Selection Completeness (4.32) and hypothesis (d). Then by (g)

$$\chi([\![\texttt{ance–or–self}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle}) = \kappa$$

[T-PareAnceAxis and Axis = `parent`]
In this case we have that $\vec{\mathbf{n}} = \text{docOrder}_\eta([\![\texttt{parent}]\!]_{\langle \eta;\vec{\mathbf{m}}\rangle})$ and by hypothesis

$$\begin{array}{ll} \kappa = \kappa' \cap \mathbf{A}_E(\tau', \texttt{ancestor}) & (f) \\ [\![\texttt{parent}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} = \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\mathbf{p}|(\mathbf{p},\mathbf{n}) \in \varepsilon_t^{\uparrow}\} & (g) \end{array}$$

and we rewrite the last term

$$\mathbf{A}_E(\tau', \texttt{ancestor}) = \mathbf{A}_E(\mathbf{A}_E(\tau', \texttt{parent}), \texttt{ancestor}) \cup \mathbf{A}_E(\tau', \texttt{parent})$$

By Theorem of Axis Selection Completeness (4.32) and hypothesis (d) we have that

$$\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) = \mathbf{A}_E(\mathbf{A}_E(\tau', \texttt{parent}), \texttt{ancestor})$$

Then by Remark (7.3), by Lemma of Containment (7.2) and hypothesis (e) we have that all the types of the ancestors of nodes in $\vec{\mathbf{m}}$ are both in κ' and in τ. Moreover by hypothesis (g) and definition of `ancestor` axis selection

$$\begin{aligned}
[\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle} &= \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{p} | (\mathbf{p}, \mathbf{n}) \in \varepsilon_t^{\uparrow+}\} \\
&= \bigcup_{\mathbf{m}\in\vec{\mathbf{m}}} \{\mathbf{p} | (\mathbf{p}, \mathbf{m}) \in \varepsilon_t^{\uparrow+}\} \\
&= [\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle}
\end{aligned}$$

because by definition of parent backward axes selection for each $\mathbf{n} \in \vec{\mathbf{n}}$ always exists $\mathbf{m} \in \vec{\mathbf{m}}$ such that $(\mathbf{m}, \mathbf{n}) \in \varepsilon_t^{\uparrow}$, and by hypothesis (e) we have that $\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle}) = \kappa'$. This means that all the types of the ancestors of a node in $\vec{\mathbf{n}}$ are yet in κ'. We have found two sets equal to $\chi([\![\texttt{ance–or–self}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle})$ then

$$\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) = \kappa$$

[T-PareAnceAxis and Axis = `ancestor`]
In this case we have that $\vec{\mathbf{n}} = \text{docOrder}_\eta([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle})$ and

$$\begin{aligned}
\kappa &= \kappa' \cap \mathbf{A}_E(\tau', \texttt{ancestor}) && (f) \\
[\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle} &= \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}} \{\mathbf{p} | (\mathbf{p}, \mathbf{n}) \in \varepsilon_t^{\uparrow+}\} && (g)
\end{aligned}$$

and we rewrite the last term

$$\mathbf{A}_E(\tau', \texttt{ancestor}) = \mathbf{A}_E(\mathbf{A}_E(\tau', \texttt{ancestor}), \texttt{ancestor}) \cup \mathbf{A}_E(\tau', \texttt{ancestor})$$

and by Theorem of Axis Selection Soundness (4.27) and hypothesis (d) we have that

$$\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) = \mathbf{A}_E(\mathbf{A}_E(\tau', \texttt{ancestor}), \texttt{ancestor})$$

Then by Remark (7.3), by Lemma of Containment (7.2) and hypothesis (e) we have that all the types of the ancestors of nodes in $\vec{\mathbf{m}}$ are both in κ' and in τ. By definition

$$\begin{aligned}[\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle} &= \bigcup_{\mathbf{n}\in\vec{\mathbf{n}}}\{\mathbf{p}|(\mathbf{p},\mathbf{n})\in\varepsilon_t^{\uparrow+}\}\\ &= \bigcup_{\mathbf{m}\in\vec{\mathbf{m}}}\{\mathbf{p}|(\mathbf{p},\mathbf{m})\in\varepsilon_t^{\uparrow+}\}\\ &= [\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle}\end{aligned}$$

because by definition of backward axes selection for each $\mathbf{n}\in\vec{\mathbf{n}}$ always exists $\mathbf{m}\in\vec{\mathbf{m}}$ such that $(\mathbf{m},\mathbf{n})\in\varepsilon_t^{\uparrow+}$, and by hypothesis (e) we have that

$$\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle}) = \kappa'$$

This means that all the types of the ancestors of a node in $\vec{\mathbf{n}}$ are yet in κ'. We have two sets equal $\chi([\![\texttt{ance–or–self}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle})$ then

$$\chi([\![\texttt{ance–or–self}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) = \kappa$$

[T-ForwAxis and Axis = `following–sibling`] In this case we have that $\vec{\mathbf{n}} = \mathrm{docOrder}_\eta([\![\mathsf{Axis}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle})$ and by hypothesis

$$\begin{array}{ll}\kappa = \kappa' \cup \mathbf{A}_E(\tau', \mathsf{Axis}) & (f)\\ [\![\texttt{following–sibling}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle} = \bigcup_{\mathbf{m}\in\vec{\mathbf{m}}}\{\mathbf{n}\mid(\mathbf{m},\mathbf{n})\in\varepsilon_t^{\rightarrow+}\} & (g)\end{array}$$

By hypothesis (e) we have that all the types of the nodes ancestors of nodes in the current dynamic context are in κ'. Moreover by (g) and by the relation $\varepsilon_t^{\rightarrow+}$ we have that $\varepsilon_t^{\uparrow+}(\mathbf{n}) = \varepsilon_t^{\uparrow+}(\mathbf{m})$ and this implies that

$$[\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle} = [\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle}$$

and then by (f)

$$\chi([\![\texttt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) = \kappa$$

[T-ForwAxis and Axis = `preceding–sibling`] In this case we have that $\vec{\mathbf{n}} = \mathrm{docOrder}_\eta([\![\mathsf{Axis}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle})$ and by hypothesis

$$\begin{array}{ll}\kappa = \kappa' \cup \mathbf{A}_E(\tau', \mathsf{Axis}) & (f)\\ [\![\texttt{preceding–sibling}]\!]_{\langle\eta;\vec{\mathbf{m}}\rangle} = \bigcup_{\mathbf{m}\in\vec{\mathbf{m}}}\{\mathbf{n}\mid(\mathbf{n},\mathbf{m})\in\varepsilon_t^{\rightarrow+}\} & (g)\end{array}$$

By hypothesis (e) we have that all the types of the nodes ancestors of nodes in the current dynamic context are in κ'. Moreover by (g) and by the relation $\varepsilon_t^{\rightarrow +}$ we have that $\varepsilon_t^{\uparrow +}(\mathbf{n}) = \varepsilon_t^{\uparrow +}(\mathbf{m})$ and this implies that

$$[\![\texttt{ance–or–self}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle} = [\![\texttt{ancestor}]\!]_{\langle \eta;\vec{\mathbf{m}}\rangle}$$

and then by (f)

$$\chi([\![\texttt{ance–or–self}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle}) = \kappa$$

[T-FolPreAxis and $\mathsf{Axis} = \{\texttt{following}, \texttt{preceding}\}$] In this case we have that $\vec{\mathbf{n}} = \mathrm{docOrder}_\eta([\![\texttt{following}]\!]_{\langle \eta;\vec{\mathbf{m}}\rangle})$ and

$$\kappa = \mathbf{A}_E(\tau, \mathsf{Axis}) \ \cup \ (\mathbf{A}_E(\mathbf{A}_E(\tau, \mathsf{Axis}), \texttt{ancestor}) \ \cap \ \kappa) \quad (f)$$

We have to show that

$$\chi([\![\texttt{ance–or–self}]\!] \ = \ (\mathbf{A}_E(\mathbf{A}_E(\tau, \mathsf{Axis}), \texttt{ancestor}) \ \cap \ \kappa))$$

We have that all the ancestors of types of nodes selected by following axis selection, except the root node of the DTD are in $\tau = \mathbf{A}_E(\tau, \mathsf{Axis})$. The ancestor axis selection wrt to τ selects all the type of the ancestor-nodes, this means that it takes also the type of the root node and other not-useful types as stated in Remarks (7.3), but the intersection $\tau \cap \kappa'$ gives only the type of the root node of the DTD. In this way we have all the type ancestors of the selected nodes and then

$$\chi([\![\texttt{ancestor}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle}) = \kappa$$

Lemma 7.8 (Completeness of test filtering typing) *Let an evaluation wrt a DTD* (W, E) $\mathcal{A} = (\eta, \rho, \Gamma)$ *if* $\Gamma \vdash_E \texttt{self}::\mathsf{Test} : (\tau, \kappa)$ *and* $\rho \Vdash_\eta \texttt{self}::\mathsf{Test} \Rightarrow \vec{\mathbf{n}}$ *then*

1. $\chi(\vec{\mathbf{n}}) = \tau$
2. $\chi([\![\texttt{ance–or–self}]\!]_{\langle \eta;\vec{\mathbf{n}}\rangle}) = \kappa$

Proof. 1)

[T-TEST and $\mathsf{Test} \in \{\mathtt{node}, \mathtt{tag}, \mathtt{text}\}$] In this case we have that $\tau = \mathbf{T}_E(\tau', \mathsf{Test})$ and by hypothesis

$$\begin{array}{ll} \Gamma(c_s) = (\tau', \kappa') & (a) \\ \tau = \mathbf{T}_E(\tau', \mathsf{Test}) & (b) \\ \vec{\mathbf{n}} = \mathrm{docOrder}_\eta([\![\mathsf{Test}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) & (c) \end{array}$$

and since serialize simply orders elements $\chi(\mathrm{docOrder}_\eta([\![\mathsf{Test}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle})) = \chi([\![\mathsf{Test}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle})$. Then by Theorem of Test Filtering Soundness (4.28) follows

$$\chi(\mathrm{docOrder}_\eta([\![\mathsf{Test}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle})) = \tau$$

Proof. 2) By hypothesis $\Gamma(c_s) = (\tau', \kappa')$ and $\rho(c_d) = \vec{\mathbf{m}}$ and we have that

$$\begin{array}{ll} \chi(\vec{\mathbf{m}}) = \tau' & (a) \\ \chi([\![\mathtt{ance\text{–}or\text{–}self}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) = \kappa' & (b) \end{array}$$

In this case we have that $\vec{\mathbf{n}} = \mathrm{docOrder}_\eta([\![\mathsf{Test}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle})$ and

$$\kappa = (\kappa' \cap \mathbf{A}_E(\mathbf{A}_E(\tau', \mathsf{Test}), \mathtt{ancestor})) \cup \mathbf{T}_E(\tau, \mathsf{Test}) \quad (c)$$

By Theorem of Axis Selection Completeness (4.32) and Theorem of Test Filtering Completeness (4.33) and hypothesis (a) we have that

$$\chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \mathbf{A}_E(\mathbf{T}_E(\tau', \mathsf{Test}), \mathtt{ancestor})$$

Then by Remark (7.3), by Lemma of Containment (7.2) and hypothesis (b) we have that all the types of the ancestors of nodes in $\vec{\mathbf{m}}$ are in κ' then

$$\chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \kappa' \cap \mathbf{A}_E(\mathbf{A}_E(\tau', \mathsf{Test}), \mathtt{ancestor}))$$

and then by hypothesis (c)

$$\chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \kappa \quad \square$$

Lemma 7.9 (Completeness of condition typing) *Let an evaluation wrt a DTD (W, E) $\mathcal{A} = (\eta, \rho, \Gamma)$ if $\Gamma \vdash_E \mathsf{Cond} : (\tau, \kappa)$ and $\rho \Vdash_\eta \mathsf{Cond} \Rightarrow \vec{\mathbf{n}}$ then*

1. $\chi(\vec{\mathbf{n}}) = \tau$
2. $\chi([\![\mathtt{ance\text{–}or\text{–}self}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \kappa$

We prove proposition (1) and (2) by induction on the derivation tree and case distinction on the last applied rule.

Proof 1)
[$\mathsf{Cond} = \mathtt{true}$]
We have that $\rho(c_d) = \vec{\mathbf{m}}$ and $\Gamma(c_s) = (\tau', \kappa')$ and by hypothesis the environments are consistent. By semantics rule S-CondTrue and T-CondTrue follows that $\chi(\vec{\mathbf{n}}) = \Sigma_\tau$.

[T-CondTrue] By definition $\chi(()) = \varnothing$.

[T-CondDisj]
By inductive hypothesis on rules S-CondDisj and T-CondDisj $\chi(\vec{\mathbf{n}}_1) = \Sigma_{\tau_1}$ and $\chi(\vec{\mathbf{n}}_2) = \Sigma_{\tau_2}$, then $\chi(\vec{\mathbf{n}}_1, \vec{\mathbf{n}}_2) = \Sigma_{\tau_1} \cup \Sigma_{\tau_2}$

[T-CondQry]
By inductive hypothesis on rules S-CondQry and T-CondQry.

[T-CondIsEmptyVar]
By consistency of Γ and ρ and inductive hypothesis on rules S-CondIsEmpty and T-CondIsEmpty.
Proof 2)

[T-CondTrue] By hypothesis $\rho(c_d) = \vec{\mathbf{m}}$ and $\Gamma(c_s) = (\tau', \kappa')$ and the environments are consistent. By semantics rule S-CondTrue and T-CondTrue follows that $\chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \kappa$.

[T-CondFalse] By definition $\chi([\![\mathtt{ancestor}]\!]_{\langle \eta; () \rangle}) = \varnothing$.

[T-CondDisj]
By inductive hypothesis on rules S-CondDisj and T-CondDisj we have that

$$\chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}}_1 \rangle}) = \Sigma_{\tau_1} \text{ and } \chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}}_2 \rangle} = \Sigma_{\tau_2})$$

then

$$\chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}}_1, \vec{\mathbf{n}}_2 \rangle} = \Sigma_{\tau_1} \cup \Sigma_{\tau_2})$$

[T-CondQry] By inductive hypothesis on rules S-CondQry and T-

CondQry.

[T-CondIsEmptyVar]
By consistency of Γ and ρ and inductive hypothesis on rules S-CondIsEmpty and T-CondIsEmpty.

Lemma 7.10 (Completeness of path typing) *Let an evaluation wrt a DTD (W, E) $\mathcal{A} = (\eta, \rho, \Gamma)$ (a) *–guarded, (b) non-recursive, and (c) parent-unambiguous, if* $\Gamma \vdash_E \mathsf{Path} : (\tau, \kappa)$ *and* $\rho \Vdash_\eta \mathsf{Path} \Mapsto \vec{\mathbf{n}}$ *then*

1. $\chi(\vec{\mathbf{n}}) = \tau$
2. $\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \kappa$

We prove both the condition by induction on the derivation tree and case distinction on last applied rule.

[T-AxisTest] In this case we have that $\Gamma \vdash_E \mathsf{Axis::Test} : (\tau, \kappa)$ and by inductive hypothesis we have that

$$\begin{array}{ll} \Gamma \vdash_E \mathsf{Axis::}\texttt{node/self}\mathsf{::Test} : (\tau, \kappa) & (a) \\ \rho \Vdash_\eta \mathsf{Axis::}\texttt{node/self}\mathsf{::Test} \Mapsto \vec{\mathbf{n}} & (b) \\ \chi(\vec{\mathbf{n}}) = \tau & (c) \\ \chi([\![\texttt{ance–or–self}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \kappa & (d) \end{array}$$

Since $\mathsf{Axis::}\texttt{node/self}\mathsf{::Test}$ is equivalent to $\mathsf{Axis::Test}$ we have that $\rho \Vdash_\eta \mathsf{Axis::Test} \Mapsto \vec{\mathbf{n}}'$ and $\vec{\mathbf{n}} = \vec{\mathbf{n}}'$. By all hypothesis (a-d) both statements (1) and (2) holds.

[T-UnfoldedPath]
In this case we have that when $c_s \mapsto (\{Y_1, \ldots, Y_n\}, \kappa') \in \Gamma$ then

$$\Gamma \vdash_E \mathsf{Path} : (\tau, \kappa)$$

and by inductive hypothesis we have that

$$\begin{array}{lll} \rho(c_d) = \vec{\mathbf{m}} & & (a) \\ \Gamma(c_s) = (\tau', \kappa') & & (b) \\ \text{for all } i = 1..n & c_s \mapsto (\{Y_1\}, \kappa') \in \Gamma \quad \text{and} \quad \Gamma \vdash_E \mathsf{Path} : (\tau_i, \kappa_i) & (c) \\ \text{for all } i = 1..n & c_d \mapsto \mathbf{m}_i \in \rho \quad \text{and} \quad \rho \Vdash_\eta \mathsf{Path} \Mapsto \vec{\mathbf{n}}_i & (d) \\ \text{for all } i = 1..n & \chi(\vec{\mathbf{n}}_i) = \tau_i & (e) \\ \text{for all } i = 1..n & \chi([\![\texttt{ance–or–self}]\!]_{\langle \eta; \vec{\mathbf{n}}_i \rangle}) = \kappa_i & (f) \end{array}$$

moreover by the typing rule T-UnfoldedPath

$$\tau = \bigcup_{i=1..n,\, \tau_i \neq \varnothing} \tau_i \quad (g)$$
$$\kappa = \bigcup_{i=1..n,\, \tau_i \neq \varnothing} \kappa_i \quad (h)$$

and when $c_d \mapsto \mathbf{m} \in \rho$ the evaluation of the query is

$$\rho \Vdash_\eta \mathsf{Path} \Mapsto \vec{\mathbf{n}}$$

where $\vec{\mathbf{n}} = (\vec{\mathbf{n}}_1, \ldots, \vec{\mathbf{n}}_n)$ and then by hypothesis (a-e,g) follows immediately

$$\chi(\vec{\mathbf{n}}) = \tau$$

and by hypothesis (a-d,f,h) and the fact that if $\tau_i = \varnothing$ then there are no ancestors also

$$[\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle} = \kappa$$

[T-Cond]
In this case we have that when $c_s \mapsto (\{Y_1, \ldots, Y_n\}, \kappa') \in \Gamma$ then

$$\Gamma \vdash_E \mathsf{Cond} : (\tau, \kappa)$$

and by inductive hypothesis we have that

$$\begin{array}{lll}
\rho(c_d) = \vec{\mathbf{m}} & & (a)\\
\Gamma(c_s) = (\tau', \kappa') & & (b)\\
\text{for all } i = 1..n & c_s \mapsto (\{Y_i\}, \kappa') \in \Gamma \quad \text{and} \quad \Gamma \vdash_E \mathsf{Cond} : (\tau_i, \kappa_i) & (c)\\
\text{for all } i = 1..n & c_d \mapsto \mathbf{m}_i \in \rho \quad \text{and} \quad \rho \Vdash_\eta \mathsf{Cond} \Mapsto \vec{\mathbf{n}}_i & (d)\\
\text{for all } i = 1..n & \chi(\vec{\mathbf{n}}_i) = \tau_i & (e)\\
\text{for all } i = 1..n & \chi([\![\texttt{ance-or-self}]\!]_{\langle \eta; \vec{\mathbf{n}}_i \rangle}) = \kappa_i & (f)
\end{array}$$

moreover by the typing rule T-Cond

$$\tau = \bigcup_{i=1..n,\, \tau_i \neq \varnothing} \tau_i \quad (g)$$
$$\kappa = \bigcup_{i=1..n,\, \tau_i \neq \varnothing} \kappa_i \quad (h)$$

and when $c_d \mapsto \mathbf{m} \in \rho$ the evaluation of the query is

$$\rho \Vdash_\eta \mathsf{Path} \Mapsto \vec{\mathbf{n}}$$

where $\vec{\mathbf{n}} = (\vec{\mathbf{n}}_1, \ldots, \vec{\mathbf{n}}_n)$ and then by hypothesis (a-e,g) follows immediately

$$\chi(\vec{\mathbf{n}}) = \tau$$

and by hypothesis (a-d,f,h) and the fact that if $\tau_i = \varnothing$ then there are no ancestors also

$$[\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle} = \kappa$$

[T-StepCond] In this case we have that $\Gamma \vdash_E \mathsf{Step[Cond]} : (\tau, \kappa)$ and by inductive hypothesis we have that

$$\begin{array}{ll} \Gamma \vdash_E \mathsf{Step/}\texttt{self::node}\mathsf{[Cond]} : (\tau, \kappa) & (a) \\ \rho \Vdash_\eta \mathsf{Step/}\texttt{self::node}\mathsf{[Cond]} \Mapsto \vec{\mathbf{n}} & (b) \\ \chi(\vec{\mathbf{n}}) = \tau & (c) \\ \chi([\![\texttt{ance–or–self}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \kappa & (d) \end{array}$$

Since $\mathsf{Step/}\texttt{self::node}\mathsf{[Cond]}$ is equivalent to $\mathsf{Step[Cond]}$ we have that $\rho \Vdash_\eta \mathsf{Step[Cond]} \Mapsto \vec{\mathbf{n}}'$ and $\vec{\mathbf{n}} = \vec{\mathbf{n}}'$. By all hypothesis (a-d) both statements (1) and (2) holds.

[T-StepPath]
By inductive hypothesis on T-StepPath and S-StepPath we have that

$$\begin{array}{ll} \Gamma \vdash_E \mathsf{Step} : (\tau', \kappa') & (a) \\ \rho \Vdash_\eta \mathsf{Step} \Mapsto \vec{\mathbf{m}} & (b) \\ \chi(\vec{\mathbf{m}}) = \tau' & (c) \\ \chi([\![\texttt{ance–or–self}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) = \kappa' & (d) \\ & \\ \Gamma[c_s \mapsto \Sigma'] \vdash_E \mathsf{Path} : (\tau, \kappa) & (e) \\ \rho[c_d \mapsto \vec{\mathbf{m}}] \Vdash_\eta \mathsf{Path} \Mapsto \vec{\mathbf{n}} & (f) \\ \chi(\vec{\mathbf{n}}) = \Sigma_\tau & (g) \\ \chi([\![\texttt{ance–or–self}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \kappa' & (h) \end{array}$$

and since $\Gamma \vdash_E \mathsf{Step/Path} : (\tau, \kappa)$ and $\rho \Vdash_\eta \mathsf{Step/Path} \Mapsto \vec{\mathbf{n}}$ by hypothesis (a-h) both condition (1) and (2) holds.

[T-StepCondPath]

By inductive hypothesis on T-StepCondPath and S-StepCondPath we have that

$$\begin{array}{ll} \Gamma \vdash_E \mathsf{Step[Cond]} : (\tau', \kappa') & (a) \\ \rho \Vdash_\eta \mathsf{Step[Cond]} \Mapsto \vec{\mathbf{m}} & (b) \\ \chi(\vec{\mathbf{m}}) = \tau' & (c) \\ \chi([\![\texttt{ance–or–self}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) = \kappa' & (d) \end{array}$$

$$\begin{array}{ll} \Gamma[c_s \mapsto \Sigma'] \vdash_E \mathsf{Path} : (\tau, \kappa) & (e) \\ \rho[c_d \mapsto \vec{\mathbf{m}}] \Vdash_\eta \mathsf{Path} \Mapsto \vec{\mathbf{n}} & (f) \\ \chi(\vec{\mathbf{n}}) = \Sigma_\tau & (g) \\ \chi([\![\texttt{ance–or–self}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \kappa' & (h) \end{array}$$

and since $\Gamma \vdash_E \mathsf{Step[Cond]/Path} : (\tau, \kappa)$ and $\rho \Vdash_\eta \mathsf{Step[Cond]/Path} \Mapsto \vec{\mathbf{n}}$ by hypothesis (a-h) both condition (1) and (2) holds.

Theorem(4.29) (Completeness of Type System) Let an evaluation wrt a DTD (W, E) $\mathcal{A} = (\eta, \rho, \Gamma)$ if $\Gamma \vdash_E \mathsf{q} : (\tau, \kappa)$ and $\rho \Vdash_\eta \mathsf{q} \Mapsto \vec{\mathbf{n}}$ then

1. $\chi(\vec{\mathbf{n}}) = \tau$
2. $\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) = \kappa$

Proof 1) by induction on the derivation tree and case distinction on the last applied rule.

[T-Empty]
By rules T-Empty and S-Empty we have that

$$\begin{array}{ll} \Gamma \vdash_E () : (\varnothing, \varnothing) & (a) \\ \rho \Vdash_\eta () \Mapsto () & (b) \end{array}$$

and since $\chi(()) = \varnothing$ follows immediately both conditions (1) and (2).

[T-Concat]
In this case we have that the enriched type is computed as union of the two enriched types of the sub-expressions $\Sigma = \Sigma^1 \cup \Sigma^2$. By inductive hypothesis we have that

$$\begin{array}{ll} \Gamma \vdash_E q_1 : \Sigma^1 \quad and \quad \rho \Vdash_\eta q_1 \Mapsto \vec{\mathbf{n}}_1 & (a) \\ \Gamma \vdash_E q_2 : \Sigma^2 \quad and \quad \rho \Vdash_\eta q_2 \Mapsto \vec{\mathbf{n}}_2 & (b) \\ \chi(\vec{\mathbf{n}}_1) = \Sigma^1_\tau & (c) \\ \chi(\vec{\mathbf{n}}_2) = \Sigma^2_\tau & (d) \\ \chi([\![\texttt{ance–or–self}]\!]_{\langle \eta, \vec{\mathbf{n}}_1 \rangle}) = \Sigma^1_\kappa & (e) \\ \chi([\![\texttt{ance–or–self}]\!]_{\langle \eta, \vec{\mathbf{n}}_1 \rangle}) = \Sigma^2_\kappa & (f) \end{array}$$

then by hypothesis (a-f), rule T-Concat and S-Concat follows immediately both conditions (1) and (2).

[T-EltConstruction]
By rule (S-EltConstr) $\rho \Vdash_\eta$ <a>q $\Mapsto \mathbf{n}_a$ where $\mathbf{n}_{aX} = \bot$ because element construction assigned it null type, and it has no ancestors since it's done in the rightmost part of the query both conditions (1) and (2) follows immediately.

[T-Var]
By hypothesis on the evaluation $\mathcal{A}$ the variable $\mathtt{x}$ is bounded to an enriched type that satisfies both conditions (1) and (2).

[T-VarPath]
By inductive hypothesis on T-VarPath and S-VarPath we have that

$$\begin{array}{ll}
\Gamma \vdash_E \mathtt{x} : (\tau', \kappa') & (a)\\
\rho \Vdash_\eta \mathtt{x} \Mapsto \vec{\mathbf{m}} & (b)\\
\chi(\vec{\mathbf{m}}) = \tau' & (c)\\
\chi([\![\mathtt{ance\text{–}or\text{–}self}]\!]_{\langle \eta; \vec{\mathbf{m}}\rangle}) = \kappa' & (d)
\end{array}$$

$$\begin{array}{ll}
\Gamma[c_s \mapsto \Sigma'] \vdash_E \mathsf{Path} : (\tau, \kappa) & (e)\\
\rho[c_d \mapsto \vec{\mathbf{m}}] \Vdash_\eta \mathsf{Path} \Mapsto \vec{\mathbf{n}} & (f)\\
\chi(\vec{\mathbf{n}}) = \Sigma_\tau & (g)\\
\chi([\![\mathtt{ance\text{–}or\text{–}self}]\!]_{\langle \eta; \vec{\mathbf{n}}\rangle}) = \kappa' & (h)
\end{array}$$

and since $\Gamma \vdash_E \mathtt{x/Path} : (\tau, \kappa)$ and $\rho \Vdash_\eta \mathtt{x/Path} \Mapsto \vec{\mathbf{n}}$ by hypothesis (a-h) both condition (1) and (2) holds.

[T-For]
By inductive hypotheses on rules T-For and S-For we have that

$$\begin{array}{ll}
\Gamma[c_s \mapsto \Sigma] \vdash_E q_1 : (\{Y_1, \ldots, Y_n\}, \kappa') & (a)\\
\rho \Vdash_\eta q_1 \Mapsto \vec{\mathbf{m}} & (b)\\
\chi(\vec{\mathbf{m}}) = \{Y_1, \ldots, Y_n\} & (c)\\
\chi([\![\mathtt{ance\text{–}or\text{–}self}]\!]_{\langle \eta; \vec{\mathbf{n}}\rangle}) = \kappa' & (d)
\end{array}$$

$$\begin{array}{ll}
\text{for each } Y_j \quad \Gamma[c_s \mapsto (\{Y_j\}, \kappa')] \vdash_E \mathtt{q}_2 : \Sigma^j & (e)\\
\text{for each } \mathbf{m}_j \in \vec{\mathbf{m}} \quad \rho[c_d \mapsto \mathbf{m}_j] \Vdash_\eta q_2 \Mapsto \vec{\mathbf{n}}_j & (f)\\
\text{for each } j = 1..|\vec{\mathbf{m}}| \quad \chi(\vec{\mathbf{n}}_j) \subseteq \Sigma^j_\tau & (g)\\
\text{for each } j = 1..|\vec{\mathbf{m}}| \quad \chi([\![\mathtt{ancestor}]\!]_{\langle \eta, \vec{\mathbf{n}}_j\rangle}) & (h)
\end{array}$$

then by hypothesis (a-h) follows that both conditions (1) and (2) holds.

[T-LET]
By inductive hypotheses on rules T-LETQRY and S-LETQRY we have that

$$\begin{array}{ll}
\Gamma \vdash_E q_1 : (\tau', \kappa') & (a)\\
\rho \Vdash_\eta q_1 \Mapsto \vec{\mathbf{m}} & (b)\\
\chi(\vec{\mathbf{m}}) = \tau' & (c)\\
\chi([\![\texttt{ance–or–self}]\!]_{\langle \eta; \vec{\mathbf{n}}\rangle}) = \kappa' & (d)\\
\\
\Gamma[\mathbf{x} \mapsto \tau'] \vdash_E \mathsf{q}_2 : (\tau, \kappa) & (e)\\
\rho[\mathbf{x} \mapsto \mathbf{m}] \Vdash_\eta q_2 \Mapsto \vec{\mathbf{n}} & (f)\\
\chi(\vec{\mathbf{n}}) = \tau & (g)\\
\chi([\![\texttt{ance–or–self}]\!]_{\langle \eta, \vec{\mathbf{n}}\rangle}) = \kappa & (h)
\end{array}$$

then by hypothesis (a-h) follows that both conditions (1) and (2) holds.

[T-IF]
By inductive hypotheses on rules T-IF and S-IF we have that

$$\begin{array}{lll}
\text{for all } i = 1..n & c_s \mapsto (\{Y_i\}, \kappa') \in \Gamma \quad \text{and} \quad \Gamma \vdash_E \mathsf{Cond} : (\tau_i, \kappa_i) & (c)\\
\text{for all } i = 1..n & c_d \mapsto \mathbf{m}_i \in \rho \quad \text{and} \quad \rho \Vdash_\eta \mathsf{Cond} \Mapsto \vec{\mathbf{n}}_i & (d)\\
\text{for all } i = 1..n & \chi(\vec{\mathbf{n}}_i) = \tau_i & (e)\\
\text{for all } i = 1..n & \chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}}_i\rangle}) = \kappa_i & (f)
\end{array}$$

moreover by the typing rule T-IF

$$\tau = \bigcup_{i=1..n, \tau_i \neq \varnothing} \tau_i \quad (g)$$

and

$$\text{if } \tau' \neq \varnothing \text{ then } \Gamma \vdash_E \mathsf{q} : \Sigma \text{ else } \Sigma = (\varnothing, \varnothing) \quad (h)$$

where Σ is the result of the typing and satisfies both conditions (1) and (2).

7.5 Soundness of Type Projection

Theorem(5.6) (Soundness Projection Inference) Given an evaluation $\mathcal{A} = (\eta, \rho, \Gamma)$ wrt a DTD (W,E). If $\Gamma \Vdash_E^{[m]} \mathsf{q} : \pi$ then the following propositions hold.

1. π is a type projector for (W, E)
2. If $\rho \Vdash_\eta q \Mapsto \vec{\mathbf{n}}; \eta'$ then $\rho \Vdash_{\eta\backslash\pi} q \Mapsto \vec{\mathbf{m}}; \eta''$ and $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$

Proof by induction on the length of the derivation and case distinction on the last applied rule.

[P-Empty]
The set $\pi = \varnothing$ is by definition a type projector a for (W, E). By rule P-Empty the semantics of the empty query is always the empty sequence whatever store is used.

[P-Concat]
Let π_1 and π_2 type projectors for the subqueries, by rule P-Concat $\pi = \pi_1 \cup \pi_2$, and the type projector is closed respect to the union so π is a type projector too. We apply inductive hypothesis on subqueries $\mathtt{q}_1, \mathtt{q}_2$ obtaining:

$$\rho \Vdash_\eta \mathtt{q}_1 \Mapsto \vec{\mathbf{n}}', \quad \rho \Vdash_{\eta_{\setminus\pi}} \mathtt{q}_1 \Mapsto \vec{\mathbf{m}}' \quad \text{and} \quad \vec{\mathbf{n}}' \cong \vec{\mathbf{m}}'$$
$$\rho \Vdash_\eta \mathtt{q}_2 \Mapsto \vec{\mathbf{n}}'', \quad \rho \Vdash_{\eta_{\setminus\pi}} \mathtt{q}_2 \Mapsto \vec{\mathbf{m}}'' \quad \text{and} \quad \vec{\mathbf{n}}'' \cong \vec{\mathbf{m}}''$$

then the thesis follows by rule S-Concat.

[P-EltConstr]
By inductive hypothesis, π is a type projector for (W, E) and element construction does not introduce new types. By rule S-EltConstr $\rho \Vdash_\eta$ <a>q $\Mapsto \mathbf{n}_a$ and $\rho \Vdash_{\eta_{\setminus\pi}}$ <a>q $\Mapsto \mathbf{n}_a$ because the pruning does not affect $\mathbf{n}_a$.

[P-Var1]
By hypothesis (τ, κ) is an enriched type and by Theorem (4.29) Soundness of Type System we have that $\tau \cup \kappa$ is a type projector. The variable is not useful for the final result of the query. Moreover, whatever store is used (pruned or not) the semantics of the variable evaluation depends only by the binding into the dynamical environment ρ, so the second proposition holds too.

[P-Var2]
By hypothesis (τ, κ) is an enriched type and by Theorem (4.29) Soundness of Type System we have that $\mathbf{A}_E(\tau, \mathtt{dos}) \cup \kappa$ is a type projector. The variable is useful for the final result of the query then all types of descendants of nodes typed in τ must be taken to materialize the result. Whatever store is used (pruned or not) the semantics of the variable evaluation depends only by the binding into the dynamical environment ρ, so the second proposition holds too.

[AxTe]
By rule (S-AxTe) we know that Axis::node/self::Test and Axis::Test evaluations are the same, if done under the same dynamic environment. By inductive hypothesis we have that the theorem holds for Axis::node/self::Test, then π is a type projector and also if $\rho \Vdash_{\eta} \mathsf{Axis::Test} \Mapsto \vec{\mathbf{n}}$ then $\rho \Vdash_{\eta_{\backslash\pi}} \mathsf{Axis::Test} \Mapsto \vec{\mathbf{m}}$ and $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$.

[AxTeCond]
By rule (S-AxTeCond) we know that are equivalent Axis::node/self::Test[Cond] and Axis::Test[Cond] evaluations, if done under the same dynamic environment. By inductive hypothesis theorem holds for Axis::node/self::Test[Cond], then π is a type projector and also if $\rho \Vdash_{\eta} \mathsf{Axis::Test[Cond]} \Mapsto \vec{\mathbf{n}}$ then $\rho \Vdash_{\eta_{\backslash\pi}} \mathsf{Axis::Test[Cond]} \Mapsto \vec{\mathbf{m}}$ and $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$.

[P-VAx1]
We can apply Theorem of Axis Selection Soundness (4.27) on hypothesis $\Gamma \vdash_E \mathsf{Axis::node} : (\tau, \kappa)$ thus obtaining that if $\rho \Vdash_{\eta} \mathsf{Axis::node} \Mapsto \vec{\mathbf{n}}$ then

$$\begin{array}{ll} \chi(\vec{\mathbf{n}}) \subseteq \tau & (a) \\ \chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \kappa & (b) \end{array}$$

Since in this case $\pi = \tau \cup \kappa$ we have that π is trivially a type projector and

$$\eta_{\backslash\pi} = \{t_{\backslash\pi}, t_1, \ldots, t_p\}$$

because only the input tree is pruned.
If $\rho \Vdash_{\eta} \mathsf{Axis::node} \Mapsto (\mathbf{n}_1, \ldots, \mathbf{n}_q)$ the for all $i = 1..q$ we have two cases:

- $\mathbf{n}_i \in \text{Nodes}(t)$ and t is computed a run-time. This is never possible because we do not navigate over constructed elements.
- $\mathbf{n}_i \in \text{Nodes}(t)$ and t is the input tree. Since this is a vertical axes navigation and pruning discards nodes typed as not specified in $\tau \cup \pi$ then by hypothesis (b) all the ancestors of $\mathbf{n}_i$ are not discarded, and also by hypothesis (a) $\mathbf{n}_i$ is not discarded.

Finally, since the expression is not useful for the final result of the evaluation we do not need to consider types of descendants, and then follows that if $\rho \Vdash_{\eta} \mathsf{Axis::node} \Mapsto \vec{\mathbf{n}}$ then $\rho \Vdash_{\eta_{\backslash\pi}} \mathsf{Axis::node} \Mapsto \vec{\mathbf{n}}$ and $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$.

[P-VAx2]

We can apply Theorem of Axis Selection Soundness (4.27) on hypothesis $\Gamma \vdash_E \mathsf{Axis::node} : (\tau, \kappa)$ thus obtaining that if $\rho \Vdash_\eta \mathsf{Axis::node} \Mapsto \vec{\mathbf{n}}$ then

$$\chi(\vec{\mathbf{n}}) \subseteq \tau \quad (a)$$
$$\chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \kappa \quad (b)$$

Since in this case $\pi = \mathbf{A}_E(\mathtt{dos}, \tau) \cup \kappa$ we have that π is trivially a type projector and

$$\eta_{\backslash \pi} = \{t_{\backslash \pi}, t_1, \ldots, t_p\}$$

because only the input tree is pruned.
If $\rho \Vdash_\eta \mathsf{Axis::node} \Mapsto (\mathbf{n}_1, \ldots, \mathbf{n}_q)$ the for all $i = 1..q$ we have two cases:

- $\mathbf{n}_i \in \mathrm{Nodes}(t)$ and t is computed a run-time. This is never possible because we do not navigate over constructed elements.
- $\mathbf{n}_i \in \mathrm{Nodes}(t)$ and t is the input tree. Since this is a vertical axes navigation and pruning discards nodes typed as not specified in $\tau \cup \pi$ then by hypothesis (b) all the ancestors of $\mathbf{n}_i$ are not discarded, and also by hypothesis (a) $\mathbf{n}_i$ is not discarded.

Finally, since the expression is not useful for the final result of the evaluation we consider types of descendants that allows materialization of the result, and then follows that if $\rho \Vdash_\eta \mathsf{Axis::node} \Mapsto \vec{\mathbf{n}}$ then $\rho \Vdash_{\eta_{\backslash \pi}} \mathsf{Axis::node} \Mapsto \vec{\mathbf{n}}$ and $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$.

[P-HAx1]
We can apply Theorem of Axis Selection Soundness (4.27) on hypothesis $\Gamma \vdash_E \mathsf{Axis::node} : (\tau, \kappa)$ thus obtaining that if $\rho \Vdash_\eta \mathsf{Axis::node} \Mapsto \vec{\mathbf{n}}$ then

$$\chi(\vec{\mathbf{n}}) \subseteq \tau \quad (a)$$
$$\chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \kappa \quad (b)$$

Moreover by hypothesis on the evaluation $\mathcal{A}$ we know that $c_s \mapsto (\tau', \pi')$ and $c_d \mapsto \vec{\mathbf{m}}$ and

$$\chi(\vec{\mathbf{m}}) \subseteq \tau' \quad (d)$$
$$\chi([\![\mathtt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) \subseteq \kappa' \quad (e)$$

Since in this case $\pi = \tau' \cup \kappa' \cup \tau \cup \kappa$ we have that π is at the matter of facts the union of two type projectors that are $(\tau' \cup \kappa')$ and $(\tau \cup \kappa)$. Another time it happens that

$$\eta_{\backslash \pi} = \{t_{\backslash \pi}, t_1, \ldots, t_p\}$$

because only the input tree is pruned.
If $\rho \Vdash_\eta \mathsf{Axis::node} \Mapsto (\mathbf{n}_1, \ldots, \mathbf{n}_q)$ the for all $i = 1..q$ we have two cases:

- $\mathbf{n}_i \in \text{Nodes}(t)$ and t is computed a run-time. This is never possible because we do not navigate over constructed elements.

- $\mathbf{n}_i \in \text{Nodes}(t)$ and t is the input tree. Since this is an horizontal axes navigation it moves from a node that is typed with a name in τ' and all its ancestors are typed with a name in κ'. Those nodes must be reached during navigation, to do this we use hypothesis (c) and (d) imposing that $(\tau' \cup \pi') \subseteq \pi$. Moreover, pruning discards nodes typed as not specified in $\tau \cup \pi$ then by hypothesis (b) all the ancestors of $\mathbf{n}_i$ are not discarded, and also by hypothesis (a) $\mathbf{n}_i$ is not discarded.

Finally, since the expression is not useful for the final result of the evaluation we do not need to consider types of descendants, and then follows that if $\rho \Vdash_\eta \mathsf{Axis}\texttt{::node} \Mapsto \vec{\mathbf{n}}$ then $\rho \Vdash_{\eta_{\backslash\pi}} \mathsf{Axis}\texttt{::node} \Mapsto \vec{\mathbf{n}}$ and $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$.

[P-HAx2]
We can apply Theorem of Axis Selection Soundness (4.27) on hypothesis $\Gamma \vdash_E \mathsf{Axis}\texttt{::node} : (\tau, \kappa)$ thus obtaining that if $\rho \Vdash_\eta \mathsf{Axis}\texttt{::node} \Mapsto \vec{\mathbf{n}}$ then

$$\begin{array}{ll} \chi(\vec{\mathbf{n}}) \subseteq \tau & (a) \\ \chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \kappa & (b) \end{array}$$

Moreover by hypothesis on the evaluation $\mathcal{A}$ we know that $c_s \mapsto (\tau', \pi')$ and $c_d \mapsto \vec{\mathbf{m}}$ and

$$\begin{array}{ll} \chi(\vec{\mathbf{m}}) \subseteq \tau' & (d) \\ \chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{m}} \rangle}) \subseteq \kappa' & (e) \end{array}$$

Since in this case $\pi = \tau' \cup \kappa' \cup \mathbf{A}_E(\texttt{descendant–or–self}, \tau) \cup \kappa$ we have that π is at the matter of facts the union of two type projectors that are $(\tau' \cup \kappa')$ and ($\texttt{descendant–or–self}\tau \cup \kappa$). Another time it happens that

$$\eta_{\backslash\pi} = \{t_{\backslash\pi}, t_1, \ldots, t_p\}$$

because only the input tree is pruned.
If $\rho \Vdash_\eta \mathsf{Axis}\texttt{::node} \Mapsto (\mathbf{n}_1, \ldots, \mathbf{n}_q)$ the for all $i = 1..q$ we have two cases:

- $\mathbf{n}_i \in \text{Nodes}(t)$ and t is computed a run-time. This is never possible because we do not navigate over constructed elements.

- $\mathbf{n}_i \in \text{Nodes}(t)$ and t is the input tree. Since this is an horizontal axes navigation it moves from a node that is typed with a name in τ' and all its ancestors are typed with a name in κ'. Those nodes must be reached during navigation, to do this we use hypothesis (c) and (d)

imposing that $(\tau' \cup \pi') \subseteq \pi$. Moreover, pruning discards nodes typed as not specified in $\tau \cup \pi$ then by hypothesis (b) all the ancestors of $\mathbf{n}_i$ are not discarded, and also by hypothesis (a) $\mathbf{n}_i$ is not discarded.

Finally, since the expression is useful for the final result of the evaluation we need to consider types of descendants, and then follows that if $\rho \Vdash_\eta \mathsf{Axis}::\mathtt{node} \Mapsto \vec{\mathbf{n}}$ then $\rho \Vdash_{\eta_{\backslash\pi}} \mathsf{Axis}::\mathtt{node} \Mapsto \vec{\mathbf{n}}$ and $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$.

[P-SeTe1]
We can apply Theorem of Test Filtering Soundness (4.28) on hypothesis $\Gamma \vdash_E \mathtt{self}::\mathsf{Test} : (\tau, \kappa)$ thus obtaining that if $\rho \Vdash_\eta \mathtt{self}::\mathsf{Test} \Mapsto \vec{\mathbf{n}}$ then

$$\begin{array}{ll} \chi(\vec{\mathbf{n}}) \subseteq \tau & (a) \\ \chi([\![\mathtt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) \subseteq \kappa & (b) \end{array}$$

Since in this case $\pi = \tau \cup \kappa$ we have that π is trivially a type projector and

$$\eta_{\backslash\pi} = \{t_{\backslash\pi}, t_1, \ldots, t_p\}$$

because only the input tree is pruned.
If $\rho \Vdash_\eta \mathtt{self}::\mathsf{Test} \Mapsto (\mathbf{n}_1, \ldots, \mathbf{n}_q)$ the for all $i = 1..q$ we have two cases:

- $\mathbf{n}_i \in \mathrm{Nodes}(t)$ and t is computed a run-time. This is never possible because we do not navigate over constructed elements.
- $\mathbf{n}_i \in \mathrm{Nodes}(t)$ and t is the input tree. Since this is a vertical axes navigation and pruning discards nodes typed as not specified in $\tau \cup \pi$ then by hypothesis (b) all the ancestors of $\mathbf{n}_i$ are not discarded, and also by hypothesis (a) $\mathbf{n}_i$ is not discarded.

Finally, since the expression is not useful for the final result of the evaluation we do not need to consider types of descendants, and then follows that if $\rho \Vdash_\eta \mathtt{self}::\mathsf{Test} \Mapsto \vec{\mathbf{n}}$ then $\rho \Vdash_{\eta_{\backslash\pi}} \mathtt{self}::\mathsf{Test} \Mapsto \vec{\mathbf{n}}$ and $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$.

[P-SeTe2]
We can apply Theorem of Test Filtering Soundness (4.28) on hypothesis $\Gamma \vdash_E \mathtt{self}::\mathsf{Test} : (\tau, \kappa)$ thus obtaining that if $\rho \Vdash_\eta \mathtt{self}::\mathsf{Test} \Mapsto \vec{\mathbf{n}}$ then

$$\begin{array}{ll} \chi(\vec{\mathbf{n}}) \subseteq \tau & (a) \\ \chi([\![\mathtt{ancestor}]\!]_{\langle\eta;\vec{\mathbf{n}}\rangle}) \subseteq \kappa & (b) \end{array}$$

Since in this case $\pi = \mathbf{A}_E(\mathtt{dos}, \tau) \cup \kappa$ we have that π is trivially a type projector and

$$\eta_{\backslash\pi} = \{t_{\backslash\pi}, t_1, \ldots, t_p\}$$

because only the input tree is pruned.
If $\rho \Vdash_\eta \texttt{self}::\mathsf{Test} \Mapsto (\mathbf{n}_1, \ldots, \mathbf{n}_q)$ the for all $i = 1..q$ we have two cases:

- $\mathbf{n}_i \in \text{Nodes}(t)$ and t is computed a run-time. This is never possible because we do not navigate over constructed elements.
- $\mathbf{n}_i \in \text{Nodes}(t)$ and t is the input tree. Since this is a vertical axes navigation and pruning discards nodes typed as not specified in $\tau \cup \pi$ then by hypothesis (b) all the ancestors of $\mathbf{n}_i$ are not discarded, and also by hypothesis (a) $\mathbf{n}_i$ is not discarded.

Finally, since the expression is not useful for the final result of the evaluation we consider types of descendants that allows materialization of the result, and then follows that if $\rho \Vdash_\eta \texttt{self}::\mathsf{Test} \Mapsto \vec{\mathbf{n}}$ then $\rho \Vdash_{\eta_{\setminus\pi}} \texttt{self}::\mathsf{Test} \Mapsto \vec{\mathbf{n}}$ and $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$.

[P-SeNoCond1]
We can apply Theorem of Soundness Type System (4.29) on hypothesis $\Gamma \vdash_E \texttt{self}::\texttt{node}[\mathsf{Cond}] : (\tau, \kappa)$ thus obtaining that if $\rho \Vdash_\eta \texttt{self}::\texttt{node}[\mathsf{Cond}] \Mapsto \vec{\mathbf{n}}$ then

$$\begin{array}{ll} \chi(\vec{\mathbf{n}}) \subseteq \tau & (a) \\ \chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \kappa & (b) \end{array}$$

Since in this case $\pi = \tau \cup \kappa$ we have that π is trivially a type projector and

$$\eta_{\setminus\pi} = \{t_{\setminus\pi}, t_1, \ldots, t_p\}$$

because only the input tree is pruned.
If $\rho \Vdash_\eta \texttt{self}::\texttt{node}[\mathsf{Cond}] \Mapsto (\mathbf{n}_1, \ldots, \mathbf{n}_q)$ the for all $i = 1..q$ we have two cases:

- $\mathbf{n}_i \in \text{Nodes}(t)$ and t is computed a run-time. This is never possible because we do not navigate over constructed elements.
- $\mathbf{n}_i \in \text{Nodes}(t)$ and t is the input tree. Since this is a vertical axes navigation and pruning discards nodes typed as not specified in $\tau \cup \pi$ then by hypothesis (b) all the ancestors of $\mathbf{n}_i$ are not discarded, and also by hypothesis (a) $\mathbf{n}_i$ is not discarded.

Finally, since the expression is not useful for the final result of the evaluation we do not need to consider types of descendants, and then follows that if $\rho \Vdash_\eta \texttt{self}::\texttt{node}[\mathsf{Cond}] \Mapsto \vec{\mathbf{n}}$ then $\rho \Vdash_{\eta_{\setminus\pi}} \texttt{self}::\texttt{node}[\mathsf{Cond}] \Mapsto \vec{\mathbf{n}}$ and $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$.

[P-SeNoCond2]
We can apply Theorem Soundness of Type System (4.29) on hypothesis $\Gamma \vdash_E \texttt{self::node}[\mathsf{Cond}] : (\tau, \kappa)$ thus obtaining that if $\rho \Vdash_\eta \texttt{self::node}[\mathsf{Cond}] \Mapsto \vec{\mathbf{n}}$ then

$$\chi(\vec{\mathbf{n}}) \subseteq \tau \quad (a)$$
$$\chi([\![\texttt{ancestor}]\!]_{\langle \eta; \vec{\mathbf{n}} \rangle}) \subseteq \kappa \quad (b)$$

Since in this case $\pi = \mathbf{A}_E(\texttt{dos}, \tau) \cup \kappa$ we have that π is trivially a type projector and

$$\eta_{\backslash \pi} = \{t_{\backslash \pi}, t_1, \ldots, t_p\}$$

because only the input tree is pruned.
If $\rho \Vdash_\eta \texttt{self::node}[\mathsf{Cond}] \Mapsto (\mathbf{n}_1, \ldots, \mathbf{n}_q)$ the for all $i = 1..q$ we have two cases:

- $\mathbf{n}_i \in \text{Nodes}(t)$ and t is computed a run-time. This is never possible because we do not navigate over constructed elements.
- $\mathbf{n}_i \in \text{Nodes}(t)$ and t is the input tree. Since this is a vertical axes navigation and pruning discards nodes typed as not specified in $\tau \cup \pi$ then by hypothesis (b) all the ancestors of $\mathbf{n}_i$ are not discarded, and also by hypothesis (a) $\mathbf{n}_i$ is not discarded.

Finally, since the expression is not useful for the final result of the evaluation we consider types of descendants that allows materialization of the result, and then follows that if $\rho \Vdash_\eta \texttt{self::node}[\mathsf{Cond}] \Mapsto \vec{\mathbf{n}}$ then $\rho \Vdash_{\eta_{\backslash \pi}} \texttt{self::node}[\mathsf{Cond}] \Mapsto \vec{\mathbf{n}}$ and $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$.

[P-VarPath]
We know that if $(\mathtt{x} \mapsto \Sigma^{\mathtt{x}}) \in \Gamma$ we have that $\Gamma \vdash_E \mathtt{x}/\mathsf{Path} : \Sigma$ and $\Gamma[c_s \mapsto \Sigma^{\mathtt{x}}] \vdash_E \mathsf{Path} : \Sigma$. In other word rules yield to the same enriched type. This said, there are some types $Y \in \Sigma^{\mathtt{x}}_\tau$ such that

$$\Gamma[c_s \mapsto (\{Y\}, \mathbf{A}_E(\{Y\}, \texttt{ance-or-self}))] \vdash_E \mathsf{Path} : (\varnothing, -)$$

As stated in Corollary (4.30) this types yield to empty sequence in evaluation, and can be discarded optimizing the static inference. A light enriched type (τ, κ'), as described in the rule, is computed thus obtaining that $\Gamma[c_s \mapsto (\tau, \kappa')] \vdash_E \mathsf{Path} : \Sigma$. Then using inductive hypothesis on $\Gamma \Vdash^{[m]}_E \mathsf{Path} : \pi$ follows that $\rho \Vdash_\eta \mathsf{Path} \Mapsto \vec{\mathbf{n}}$ and $\rho \Vdash_{\eta_{\backslash \pi}} \mathsf{Path} \Mapsto \vec{\mathbf{m}}$ and $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$.

By inductive hypothesis on rule P-VarPath we have that π is a type projector. By Corollary (4.30) names that generates an empty set as result of static analysis can be discarded because their evaluation is the empty sequence. By inductive hypothesis on the same rule if $\rho \Vdash_{\eta} \mathsf{Path} \Rrightarrow \vec{\mathbf{n}}$ and $\rho \Vdash_{\eta_{\setminus\pi}} \mathsf{Path} \Rrightarrow \vec{\mathbf{n}}$ then $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$.

[P-StepPath]
We know that if $\Gamma \Vdash\!\vdash_E^{[m]} \mathsf{Step} : (\Sigma)$ we have that $\Gamma \vdash_E \mathsf{Step/Path} : \Sigma$ and $\Gamma[c_s \mapsto \Sigma] \vdash_E \mathsf{Path} : \Sigma$. In other word rules yield to the same enriched type. This said, there are some types $Y \in \Sigma_\tau$ such that

$$\Gamma[c_s \mapsto (\{Y\}, \mathbf{A}_E(\{Y\}, \texttt{ance-or-self}))] \vdash_E \mathsf{Path} : (\varnothing, -)$$

As stated in Corollary (4.30) this types yield to empty sequence in evaluation, and can be discarded optimizing the static inference. A light enriched type (τ, κ'), as described in the rule, is computed thus obtaining that $\Gamma[c_s \mapsto (\tau, \kappa')] \vdash_E \mathsf{Path} : \Sigma$. Then using inductive hypothesis on $\Gamma \Vdash\!\vdash_E^{[m]} \mathsf{Path} : \pi$ follows that $\rho \Vdash_{\eta} \mathsf{Path} \Rrightarrow \vec{\mathbf{n}}$ and $\rho \Vdash_{\eta_{\setminus\pi}} \mathsf{Path} \Rrightarrow \vec{\mathbf{m}}$ and $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$.

By inductive hypothesis on rule P-VarPath we have that π is a type projector. By Corollary (4.30) names that generates an empty set as result of static analysis can be discarded because their evaluation is the empty sequence. By inductive hypothesis on the same rule if $\rho \Vdash_{\eta} \mathsf{Path} \Rrightarrow \vec{\mathbf{m}}$ and $\rho \Vdash_{\eta_{\setminus\pi}} \mathsf{Path} \Rrightarrow \vec{\mathbf{n}}$ then $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$.

[P-StepCond]
We know that if $\Gamma \Vdash\!\vdash_E^{[m]} \mathsf{Step[Cond]} : (\Sigma)$ we have that $\Gamma \vdash_E \mathsf{Step[Cond]/Path} : \Sigma$ and $\Gamma[c_s \mapsto \Sigma] \vdash_E \mathsf{Path} : \Sigma$. In other word rules yield to the same enriched type. This said, there are some types $Y \in \Sigma_\tau$ such that

$$\Gamma[c_s \mapsto (\{Y\}, \mathbf{A}_E(\{Y\}, \texttt{ance-or-self}))] \vdash_E \mathsf{Path} : (\varnothing, -)$$

As stated in Corollary (4.30) this types yield to empty sequence in evaluation, and can be discarded optimizing the static inference. A light enriched type (τ, κ'), as described in the rule, is computed thus obtaining that $\Gamma[c_s \mapsto (\tau, \kappa')] \vdash_E \mathsf{Path} : \Sigma$. Then using inductive hypothesis on $\Gamma \Vdash\!\vdash_E^{[m]} \mathsf{Path} : \pi$ follows that $\rho \Vdash_{\eta} \mathsf{Path[Cond]} \Rrightarrow \vec{\mathbf{n}}$ and $\rho \Vdash_{\eta_{\setminus\pi}} \mathsf{Path[Cond]} \Rrightarrow \vec{\mathbf{m}}$ and $\vec{\mathbf{n}} \cong \vec{\mathbf{m}}$.

[P-Path] By inductive hypothesis the theorem holds for all the partial type

projection inference then thesis straightforward holds.

[P-For]
By inductive hypothesis we have that both π and π' are type projectors and that the evaluations of both the subexpressions of the for construct are sound wrt to projection. Moreover by Corollary (4.30) unuseful types are discarded as explained for the case rule [P-VarPath].

[P-Let]
By inductive hypothesis we have that both π and π' are type projectors and that the evaluations of both the subexpressions of the for construct are sound wrt to projection.

[P-If]
By inductive hypothesis we have that both π and π' are type projectors and that the evaluations of both the subexpressions of the for construct are sound wrt to projection.

Bibliography

[1] Benzaken, Castagna, Colazzo, Nguyên, *Type-based XML Projection*, VLDB, 2006

[2] Marian, Siméon, *Projecting XML Documents*, VLDB, 2003

[3] Dario Colazzo, *Path Correctness for XML Queries: Characterization and Static Type Checking*, Ph.D. Thesis, 2004

[4] Colazzo, Ghelli, Manghi, Sartiani *Static Analysis for Path Correctness of XML Queries*, Journal of Functional Programming, 2006

[5] Ghelli, Rose, SimCorollaryon, *Commutativity Analysis for XML Updates*, ACM Transactions on Database Systems, 2008

[6] Hidders, Paredaens, Vercammen, Demeyer *A Light but Formal Introduction to XQuery*, Proc. of the Second International XML Database Symposium, 2004

[7] W3C Working Group, XPath 2.0 and XQuery 1.0 formal semantics, `http://www.w3.org/TC/xquery-semantics`, 2007

[8] W3C Working Group, XML Query, `http://www.w3.org/XML/Query`

[9] W3C Working Group, XML Schema, `http://www.w3.org/XML/Schema`

[10] W3C Working Group, XQuery 1.0 and XPath 2.0 data model, `http://www.w3.org/TR/xpath-datamodel/`,W3C Recommendation, 2007

[11] W3C Working Group, XML Query Uses Cases, `http://www.w3.org/TC/xquery-use-cases`, 2007

[12] Massimo Franceschet, XPathMark: An XPath benchmark for XMark , `Technical Report`, 2004

[13] Anders Møller and Michael Schwartzbach, *An Introduction to XML and Web Technologies*, Addison Wesley, 2006

[14] D. Chamberlin *XQuery: An XML query language*, IBM Systems Journal, 2002

[15] C. M. Sperberg-McQueen, *XML <and Semi-Structured Data>*, ACM Queue, 2005

[16] Murata, Lee, Mani *Taxonomy of XML Schema Languages using Formal Language Theory*, Extreme Markup Languages, 2000

[17] Brüggermann-Klein, *Regular Expression into Finite Automata*, Theoretical Computer Science, 1993

[18] Glushkov, *The abstract theory of automata*, Russian Mathematical Surveys, 1961

[19] Brüggermann-Klein, Wood, *One Unambiguous Regular Expressions*, Information and computation , 1998

[20] S. Bressan, B. Catania, Z. Lacroix, Y-G Li and A. Maddalena, *Accelerating queries by pruning XML documents*, Data Knowledge Eng., 2005

[21] Hidders, Michiels, *Avoiding Unnecessary Ordering Operation in XPath*, VLDB, 2003

[22] Segoufin, Vianum *Validating Streaming XML Documents*, Symposium on Principles of Database Systems, 2002

[23] Frnandez, Hidders, Michiels, Siméon, Vercammen *Optimizing Sorting and Duplicate Elimination in XQuery Path Expressions*, Database and Expert Systems Applications, 2205

[24] Libkin, Kolahi, *Topics in DB: Foundations of XML Lecture 4*, Notes of the Course, 2005

[25] Colazzo, Sartiani *Detection of Corrupted Schema Mappings in XML Data Integration Systems*, ACM Journal, 2008

[26] Colazzo, Ghelli, Sartiani *Efficient Inclusion for a Class of XML Types with Interleaving an Counting*, ACM Queue, 2005

[27] Colazzo, Ghelli, Sartiani *Efficient Inclusion Between Regular Expression Types*, ACM, 2008

[28] Koch, *On the Complexity of Nonrecursive XQuery and Functional Query Lanaguages on Complex Values*, ACM Transactions on Database Systems, 2006

[29] Cheney, *Regular Expressions Subtyping for XML Query and Update Languages*, European Symposium on Programming, 2008

[30] Afanasiev, *Distributivity for XQuery expressions*, Technical report, 2007

[31] Afanasiev, Grust, Marx, Rittinger, Teubner *An Inflationary Fixed Poin Operator in XQuery*, International Conference on Data Engineering , 2008

[32] Fan, Chan, Garofalakis, *Secure XML Querying with SEcurity Views*, SIGMOD, 2004

Università Ca' Foscari - Venezia

ESTRATTO PER RIASSUNTO DELLA TESI DI LAUREA E DICHIARAZIONE DI CONSULTABILITA' (*)

Il sottoscritto/a _____

Matricola n. _____

Facoltà _____

iscritto al corso di laurea laurea magistrale/specialistica in:

_____

Titolo della tesi (**): _____

DICHIARA CHE LA SUA TESI E':

Riproducibile totalmente Non riproducibile Riproducibile parzialmente

Venezia, _____ Firma dello studente _____

(spazio per la battitura dell'estratto)

(**) il titolo deve essere quello definitivo uguale a quello che risulta stampato sulla copertina dell'elaborato consegnato al Presidente della Commissione di Laurea (*) Da inserire come ultima pagina della tesi. L'estratto non deve superare le mille battute

Informativa sul trattamento dei dati personali

Ai sensi dell'art. 10 della Legge n. 675/ 96 recante disposizioni a "Tutela delle persone e di altri soggetti rispetto al trattamento dei dati personali" si informa che:

1) I dati personali richiesti o acquisiti, i dati relativi alla carriera universitaria e comunque prodotti dall'Università Ca' Foscari di Venezia nello svolgimento delle proprie funzioni istituzionali, nonchè i dati derivanti dal trattamento automatizzato di entrambi, possono essere raccolti, trattati, comunicati e diffusi dall'Università tramite i propri uffici - sia durante la carriera universitaria dell'interessato che dopo la specializzazione - a soggetti esterni per finalità connesse allo svolgimento delle attività istituzionali dell'Università nonchè per comunicazione e diffusione rivolte esclusivamente ad iniziative di avviamento o orientamento al lavoro (stage e placement) e per attività di formazione post-laurea.
2) Il conferimento dei dati personali nell'ambito descritto al punto 1 è obbligatorio
3) Il trattamento dei dati può essere effettuato attraverso strumenti manuali, informatici e telematici atti a gestire i dati stessi ed avviene in modo da garantire la sicurezza e la riservatezza.