Indexing for XML Types

by

Karima Echihabi

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Indexing for XML Types

Karima Echihabi

Master of Science

Graduate Department of Computer Science

University of Toronto

2002

XML is increasingly gaining ground as a standard for data representation and exchange over the web. XML data most often has some embedded structure; therefore, it is interesting to exploit this structure whenever present. The structure of an XML document can be modeled by DTDs, XML Schema, or XQuery types. Both XML Schema and XQuery have a type system that supports a subtyping relation between types.

The purpose of this thesis is to develop an indexing mechanism, called Tindex, that efficiently retrieves supertypes of an XML type. We consider XQuery types and DTDs, and our indexing mechanism can be extended to XML Schema. The main research challenge for indexing XML types is the complex nature of terms to index, i.e. terms are regular expressions. Tindex can be used in query optimization, storage optimization, data integration, and XML data repositories.

Acknowledgements

I was first inspired to pursue my graduate studies when I was working at Microsoft Corp. Redmond, Washington as a Software Engineer. So, I enrolled as a part-time student at the University of Washington. After taking a database course, I decided that I wanted to spend more time learning about the database field specifically XML data management. So, when the opportunity came to study full-time at the University of Toronto, I decided to seize it. Going back to school was full of challenges, but also full of reward. I am happy I went through this endeavor.

I am thankful to my Supervisor Prof. Alberto Mendelzon for his guidance, advice and patience during my research. I would also like to extend my thanks to my second reader Prof. Ken Sevcik for his valuable time and comments.

My sincere thanks also go to my officemates for their support during stressful times, and my mentors Susan Sim from UofT, and Ann Gruhn from IBM for their advice on how to succeed in graduate school.

Finally, my deepest gratitude goes to my mother Zainab, my father Lahoussaine, my sister Sofia, my brothers Abdelhamid and Ali, and my fiance Younes for their emotional support and encouragement. I dedicate this thesis to all of them with all my love.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 XML

A markup language is a mechanism to describe the structure of a document. The international standard in markup languages is called SGML [2]. It is a very powerful language that can describe arbitrarily structured documents. However, its flexibility makes it very complex to use, especially, on the World Wide Web (WWW). The Extensible Markup Language (XML) [48] was proposed as a subset of SGML that will simplify the definition of document structure, and the development of programs that process these documents.

The XML language describes a class of data objects called XML documents. The basic component of an XML document is an *element*. Each element is enclosed between a *start-tag* and an *end-tag*. The start-tag is the element name enclosed between the delimiters “$<$” and “$>$”; and the end-tag is the element name enclosed between the delimiters “$</$” and “$>$”. The text between the start-tag and the end-tag is called the *content* of the element. An element can have a set of *attributes*, and each attribute has a name and a value.

Figure 1.1 shows a fragment of an XML document that belongs to the SIGMOD IndexTermsPage database [20]. The first line of the document is *the XML declaration* and should always be included. It defines the XML version and character encoding of the document; in this case the document conforms to the 1.0 specification of XML, and it uses the 'ISO-8859-1' character set. The second line of the document is the *Document Type Definition (DTD)*, which will be explained in the next section. The third line describes the root element of the document, which is called $IndexTermsPage$ and has several subelements in its content. One of its subelements is the element *title* which has content "Quickly Generating Billion-Record Synthetic Databases", and attribute *id*. The attribute *id* has the value "00925021". The element *title* is enclosed between the start-tag $< title >$ and the end-tag $< /title >$.

XML is important to the database community for several reasons: The presentation of an XML document is independent from its content; XML documents are self-describing; and an XML document may comply with a schema. These characteristics make XML not only a markup language but also a powerful instrument for data exchange.

In the rest of this section, we will present a brief survey of some of the developments in XML database research:

Data Modeling: DTDs introduce a notion of data modeling for XML similar to the object-oriented data models, where an XML element can be viewed as an object and its corresponding DTD element as its object class. XML Schema introduces additional data modeling features that are missing in DTDs, for instance, class hierarchies, typed object references, and some integrity constraints. Interesting research areas related to data models include: inferring the structure or schema of an XML document; checking equivalence of different schemas; and studying more complex relationships between schemas like subsumption. Work in this area includes the XDuce system developed by Hosoya and Pierce [33]; the research on reasoning about DTDs by Calvanese, Giacomo, and Lenzerini [11]; the YAT system proposed by Cluet, Delobel, Siméon, and Smaga [16]; the research on type inference by Papakonstantinou, and Vianu [44] and Milo, Suciu, and Vianu [43]; and the research work on graph simulations by Buneman, Davidson, Fernandez, and Suciu [9]. We will describe these proposals in more detail in Chapter 2.

Query Languages: Several query languages were proposed for XML: XPath 1.0 [47], which is a W3C Recommendation; XQL [53]; XML-QL [22]; Lorel [4]; and XML-GL [13]. These languages were independent early efforts to create an XML query language. Most of the authors of these proposals are now working within the W3C XML Query Working Group to produce a single W3C XML query language which will be called XQuery [25].

Algebras and Query Optimization: In addition to query languages, there were also several research proposals for an XML Algebra: the algebra for XML proposed by Beech, Malhotra, and Rys [6]; the SAL algebra proposed by Beeri and Tzaban [7]; the TAX algebra proposed by Jagadish, Lakshmaman, Srivastava, and Thompson [37]; and the TQL algebra presented by Conforti, Ferrara, and Ghelli [17]. These studies aim at defining a set of minimal algebraic operators that express a query, study the equivalence of algebraic expressions and propose optimization techniques for these algebraic expressions. Query optimization for XML is also an active area of research, proposals include: the research of McHugh and Widom [42]; the proposal of Deutsh and Tannen [3]; and the work of Fernandez and Suciu [26].

Indexing and Storage: An important challenge in querying XML is large XML data stores, thus the need for appropriate indexing and storage schemes to speed up query evaluation. Several mechanisms for indexing XML data were proposed: the index ToXin developed by Rizzolo and Mendelzon [52]; the index proposed by Cooper, Sample, Franklin, Hjaltason, and Shadmon [18]; the Index Fabric developed by Cooper and Shadmon [19] and the DataGuides index developed by Goldman and Widom [29]. Storage proposals include the work of Kanne and Moerkotte [38]; the STORED storage proposal developed by Deutsh, Fernandez, and Suciu [21]; the research of Florescu and Kossmann [27]; and the survey by Shanmugasundaram, Tufte, He, Zhang, and Witt on storing XML documents in relational systems [55].

1.2 Document Type Definitions (DTDs) and XML Schema

In this section, we will give an overview of DTDs and XML Schema. DTDs and XML Schema provide a formal description of an XML document, and are important to the database community because they will simplify operations like: data exchange, data integration, query optimization and storage optimization.

DTDs

A DTD defines the list of legal elements that make up an XML document, and describes how these elements are structured together. There are different types of declarations in a DTD, and the most common ones are *element type declarations*, *attribute-list declarations*, and *entity declarations*. An element type declaration describes the name and content of an element. For instance, the element type declaration of the element *title* in the previous XML example (Figure 1.1) is:

```
<!ELEMENT title (#PCDATA)>
```

The keyword PCDATA stands for *parsed character data* which means that the content of element *title* is character data not containing markup. In addition to PCDATA, an element declaration can have other types of content: EMPTY represents an element with no content; ANY represents an element that can have any possible content; *element content* means that the content of the element includes only child elements (also called subelements); and *mixed content* allows the content of the element to have child elements interspersed with character data. A *content model* is a grammar that describes constraints on the child elements in an element content. The content model includes a choice ("|") operator, a sequence operator (","), and occurrence operators which signal how many times an element occurs ("*" for zero or more , "+" for one or more, and

“?” for zero or one times). For example, the following declaration states that element *authors* is made of zero or more elements of type *author*:

`<!ELEMENT authors (author)*>`

An attribute-list declaration specifies the attributes associated with a particular element. The element *title* has the following attribute-list declaration:

`<!ATTLIST title id CDATA #IMPLIED>`

This declaration states that element *title* has an attribute called *id*, and the content of attribute *id* is CDATA, i.e. any arbitrary character data. The difference between PCDATA and CDATA is that CDATA text will not be parsed by the XML parser. The keyword IMPLIED means that no default value is provided for the attribute *id*.

An entity declaration defines a variable and assigns a value to it, which could be a text string (*internal entity*) or a filename that contains text information (*external entity*). Whenever the variable is encountered by the XML processor, it is replaced by its value.

A DTD can be declared inline in the XML document, or as an external reference. An external DTD reference is wrapped in a DOCTYPE definition as follows:

<!DOCTYPE *root_element* SYSTEM “*filename*”>

The second line in the XML document depicted in Figure 1.1 is an example of an external DTD declaration. The string *IndexTermsPage* is the name of the root element, or equivalently, we say that the XML document has type *IndexTermsPage*. The filename “*../DTD/IndexTermsPage.dtd*” contains the DTD element declarations, and is depicted in Figure 1.2 (comments are included between the delimiters “<!–” and “–>”).

An inline DTD declaration has the following syntax:

<!DOCTYPE *root_element* [*element_declarations*]>

where *element_declarations* is the list of element declarations.

The detailed syntax of DTDs can be found in the SIGMOD IndexTermsPage database [20].

XML Schema

In this section, we will give a brief overview of XML Schema. XML Schema [49] is a relatively new development that was proposed as a replacement to DTDs. It is not clear yet how and whether XML Schema will replace DTDs in practice, but there are clear advantages for using XML Schema over DTDs: XML Schema uses XML document syntax; it supports strong typing for elements and attributes; and it can represent default values for both elements and attributes.

In XML Schema, there are two different datatypes: *simple types*, and *complex types*. An XML element that has a simple type can have only text in its content (i.e., it cannot contain any other elements or attributes). The text can be a string or can be one of the other data types that are allowed in XML Schema (Boolean, date, etc.). An element of complex type (or complex element) is an XML element that contains other elements and/or attributes. There are four kinds of complex elements: empty elements; elements that contain only other elements; elements that contain only text; elements that contain both other elements and text. The following example illustrates a simple type declaration for element *title* that is equivalent to its DTD declaration from the previous section:

```
< xs:element name = ''title'' type = ''xs:string'' >
```

The prefix *xs* represents the *namespace* for XML Schema and points to the URL "http://www.w3.org/2001/XMLSchema". A namespace is a collection of names of elements and attributes. Namespaces are used to differentiate between element/attribute declarations written by different authors. For instance, in this case, the *xs* prefix is used to differentiate between XML Schema vocabulary and content written by the author of the document. The following declaration is a complex type declaration that is equivalent to the DTD declaration of element *authors* from the previous section:

```
<xs:element name="authors"/ >
   <xs:complexType>
      <xs:element name="author" type="xs:string"
            minOccurs ="0" maxOccurs = "unbounded"/>
   </xs:complexType>
</xs:element>
```

The keywords *minOccurs* and *maxOccurs* define how often an element can occur. In this case, element *author* can occur zero or more times. The following set of XML Schema declarations shows an alternate way for defining element *author* (i.e., with an explicit type declaration):

```
<xs:element name="authors" type = "authorsType" / >
```

where type *authorsType* is defined as follows:

```
<xs:complexType name = "authorsType">
      <xs:element name="author" type="xs:string"
            minOccurs ="0" maxOccurs = "unbounded"/>
</xs:complexType>
```

Chapter 2 will cover XML Schema types in more detail.

1.3 XQuery

XQuery is a query language for XML that is being developed by the W3C XML Query Working Group and currently has "Working Draft" status. In this section, we will first present an overview of the XQuery language [23], then we will describe the XQuery type system. Since the development of the type system is still in progress, we will base the discussion in this section on the W3C Working Draft [25].

The XQuery language is a functional language which uses a data model based on

XPath [47], and models an XML document as a tree of nodes. There are different types of expressions in XQuery including: *path expressions*; *element constructors*; *FLWR expressions*; and expressions that test or modify datatypes. These expressions can be nested together. The following fragment is a path expression that selects all books in document "mybooks.xml" written by "Jim Gray":

```
document("mybooks.xml")//book[author = ''Jim Gray'']
```

("//" selects all the *book* descendants of the document root).

A FLWR expression supports iteration and variable binding for intermediate results. The name "FLWR" stands for the keywords For, Let, Where, and Return, which are the four clauses that make up a FLWR expression. It can be used to compute joins between two or more documents. Consider three documents: "books.xml", "stores.xml" and "catalog.xml". The first document contains information about books, including the book's id and name (*bookid* and *bookname*). The second document contains information about bookstores including their ids and names (*storeid* and *storename*). The third document contains information about the relationships between books and bookstores, where each element *item* in the catalog document contains the subelements *bookid*, *storeid*, and *price*. The following FLWR expression performs a join among these three documents:

```
for $i in document("catalog.xml")//item,
   $b in document("books.xml")//book[bookid = $i/bookid],
   $s in document("stores.xml")//store[storeid = $i/storeid]
   return
      <item>
      {
         $b/bookname,
         $s/storename,
         $i/price
      }
```

```
        </item>
```

The "for" expression first binds the variables $\$i$, $\$b$, and $\$s$ with elements of type *item*, *book*, and *store*, then it constructs a new element <item> that contains the name of the book, the name of the bookstore that sells this book, and the price it sells it for. The curly brackets inside the <item> element construction expression are used to differentiate between literal text and enclosed XQuery expressions.

XQuery provides different operations on datatypes: the *instanceof* expression returns true if the value of its first operand matches the type named in its second operand; the *typeswitch* expression chooses which expressions to evaluate based on the dynamic type of an input value; the *cast* expression converts an input type to a target type (can only be applied to builtin types [24]); and the *treat as* and *assert* expressions check that the type of an expression e is a subtype of a data type t, where e and t are operands of the XQuery expression.

XQuery supports explicit element, attribute and type declarations. The following XQuery expression is equivalent to the XML Schema type declaration of element *title* from the previous section:

```
DEFINE ELEMENT title {xs:string}
```

and the following XQuery fragment is equivalent to the XML Schema declaration of element *authors* from the previous section:

```
DEFINE ELEMENT authors {authorsType}
DEFINE TYPE authorsType {
   ELEMENT author {xs:string}*
}
```

Element *authors* can also be declared without an explicit declaration for the type *authorsType* as follows:

```
DEFINE ELEMENT authors {
   ELEMENT author {xs:string}*
```

}

Chapter 2 will cover XQuery types in more detail.

1.4 ToX

The Toronto XML Server (ToX) [1] is a research effort led by the Database Group at the University of Toronto. ToX is a repository for XML data and metadata which manages XML documents, WebOQL queries, and views over documents in the repository. Documents can be stored in a file system or in a relational or object database, and the storage method is determined based on the structuredness of the document.

The system catalog contains metadata for the documents including schemas, which are used for query processing and optimization, and XML documents can be grouped as collections having a particular schema. ToX supports multiple query languages and allows queries to be issued over both the catalog and the documents.

The work in this thesis is part of the ToX research project. The indexing mechanism that we developed is used by the ToX catalog manager to recommend relevant collections when the user registers a new XML document. That is, only collections that have a schema that is a supertype of the schema of the input document will be suggested as candidate collections. Moreover, our sequential subtyping algorithm (which is a JAVA implementation of the XDuce subtyping algorithm) is also used by the ToX query processor to perform typechecking. Figure 1.3 shows the architecture of the ToX system, and our work is part of the type manager module.

1.5 Motivation

As we have pointed out earlier, XML data is self-describing, therefore, it is not necessary to have a schema describing it as is the case in the relational or object-oriented paradigms. However, since XML data most often has some embedded structure, we would like to exploit this structural information in a way similar to how relational systems use schema information for efficient data processing and storage. The work described in this thesis is an effort in that direction. Our objective is to develop an indexing mechanism, called Tindex, that efficiently retrieves supertypes of an XML type (XML types can be XQuery types, DTDs, or XML Schemas). We will also implement a JAVA version of the XDuce system, and compare its performance to that of our indexing mechanism.

The initial motivation of this work was to exploit the schema information of XML documents stored in the ToX system. Initially, when users registered XML documents in ToX, they had to specify explicitly the collection to which the document will belong. We decided to automate this process, and implement a mechanism that would recommend relevant collections when the user registers a document. Only collections that have a schema that is a supertype of the schema of the input document would be of interest. This could be done using a sequential algorithm that verifies a subtyping relation between the input document schema and the schemas of all collections. However, ToX can have many documents and collections. Therefore, the need for an indexing mechanism to speed up the subtyping checks became apparent. This work can also be useful in several other ways: it can be used in query and storage optimization, and in data integration systems. For instance, if we can infer the domain of a query, we can use the indexing mechanism to limit the data on which to run the query. Suppose we have a large repository of XML documents, and we would like to run the following query that will return titles of books that were published in year 2002:

```
for $i in document("books.xml")//book,
   WHERE $i/year = ‘‘2002’’
   RETURN $i/title
```

If we can infer that the domain d of this query is documents that have a *book* element and a *year* subelement, then, with the use of the type index, we can select the documents in our repository that have schemas containing the *book* and *year* elements. Without the use of the type index, we will need to query all documents in the repository. Besides, the relationship between schemas can be exploited to optimize storage. For instance, if a storage strategy was determined for a set of documents in our database, and we would like to register a new document and optimize its storage, we could use Tindex to efficiently select documents with equivalent schemas or with schemas that are supertypes of the schema of the input document from the database, and adopt one of the storage strategies already devised for the set of types returned by Tindex. Moreover, if we want to integrate data from different sources, and we know the schemas of these data sources, we can compare these schemas and choose the one that generalizes all of the data sources, or we we can use the structure to do data conversions. Tindex can be used to speed up both operations.

In this chapter, we introduced the XML language, DTDs, XML Schema, and XQuery. We also explained the objectives and motivation of this thesis. The outline of the rest of the thesis is as follows. The second chapter will define terminology related to types and the subtyping problem and will give an overview of the subtyping proposals for XML types. The third chapter will survey literature related to the type indexing problem and will give details of our indexing mechanism, Tindex, including algorithms and correctness proofs. The fourth chapter will describe the implementation details and the experimental results. Finally, the fifth chapter will provide a summary of the thesis and will suggest some extensions to Tindex.

```
<?xml version=’1.0’ encoding=’ISO-8859-1’ ?>
<!DOCTYPE IndexTermsPage SYSTEM ’../DTD/IndexTermsPage.dtd’>

<IndexTermsPage>
    <title id=”00925021”>Quickly Generating Billion-Record Synthetic Databases.</title>
    <authors>
        <author id=”00”>Jim Gray</author>
        <author id=”01”>Prakash Sundaresan</author>
    </authors>
    <confName>ACM SIGMOD</confName>
    <confYear>1994</confYear>
    <volume>23</volume>
    <number>2</number>
    <initPage>243</initPage>
    <endPage>252</endPage>
    <fullText href=”http://www.acm.org/pubs/articles/proceedings/p243-gray.pdf”></fullText>
    <abstract> This paper presents several database generation techniques...</abstract>
    <generalTerms>
        <term>Data, DATA STRUCTURES, Trees.</term>
        <term>Information Systems, DATABASE MANAGEMENT.</term>
    </generalTerms>
    <categoryAndSubjectDescriptors>
        <categoryAndSubjectDescriptorsTuple>
            <category>E.1</category>
            <content>Data, DATA STRUCTURES, Trees.</content>
        </categoryAndSubjectDescriptorsTuple>
        <categoryAndSubjectDescriptorsTuple>
            <category>H.2.1</category>
            <content> DATABASE MANAGEMENT, Logical Design.</content>
        </categoryAndSubjectDescriptorsTuple>
    </categoryAndSubjectDescriptors>
</IndexTermsPage>
```


Figure 1.1: SIGMOD IndexTermsPage XML Document [20]

```
<!– DTD generated by PENELOPE.
   1999 Araneus Group and University 'Roma Tre', Rome, ITALY –>
<!ENTITY % carSet SYSTEM 'CarSet.cfg'>
%carSet;
<!ELEMENT IndexTermsPage (title, authors, confName, confYear,volume
                                   number, initPage, endPage, fullText, abstract,
                                   generalTerms, categoryAndSubjectDescriptors)>
<!ELEMENT title (#PCDATA)>
<!ATTLIST title id CDATA #IMPLIED>
<!ELEMENT authors (author)*>
<!ELEMENT author (#PCDATA)>
<!ATTLIST author id CDATA #IMPLIED>
<!ELEMENT confName (#PCDATA)>
<!ELEMENT confYear (#PCDATA)>
<!ELEMENT volume (#PCDATA)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT initPage (#PCDATA)>
<!ELEMENT endPage (#PCDATA)>
<!ELEMENT fullText (size)?>
<!ELEMENT size (#PCDATA)>
<!ENTITY % Xlink
        “ xml:link CDATA #FIXED 'simple'
        href CDATA #IMPLIED
        inline (true — false) #FIXED 'true'
        ”
>
<!ATTLIST fullText %Xlink;>
<!ELEMENT abstract (#PCDATA)>
<!ELEMENT generalTerms (term)*>
<!ELEMENT term (#PCDATA)>
<!ELEMENT categoryAndSubjectDescriptors (categoryAndSubjectDescriptorsTuple)*>
<!ELEMENT categoryAndSubjectDescriptorsTuple (category,content)>
<!ELEMENT category (#PCDATA)>
<!ELEMENT content (#PCDATA)>
```

Figure 1.2: SIGMOD IndexTermsPage DTD [36]

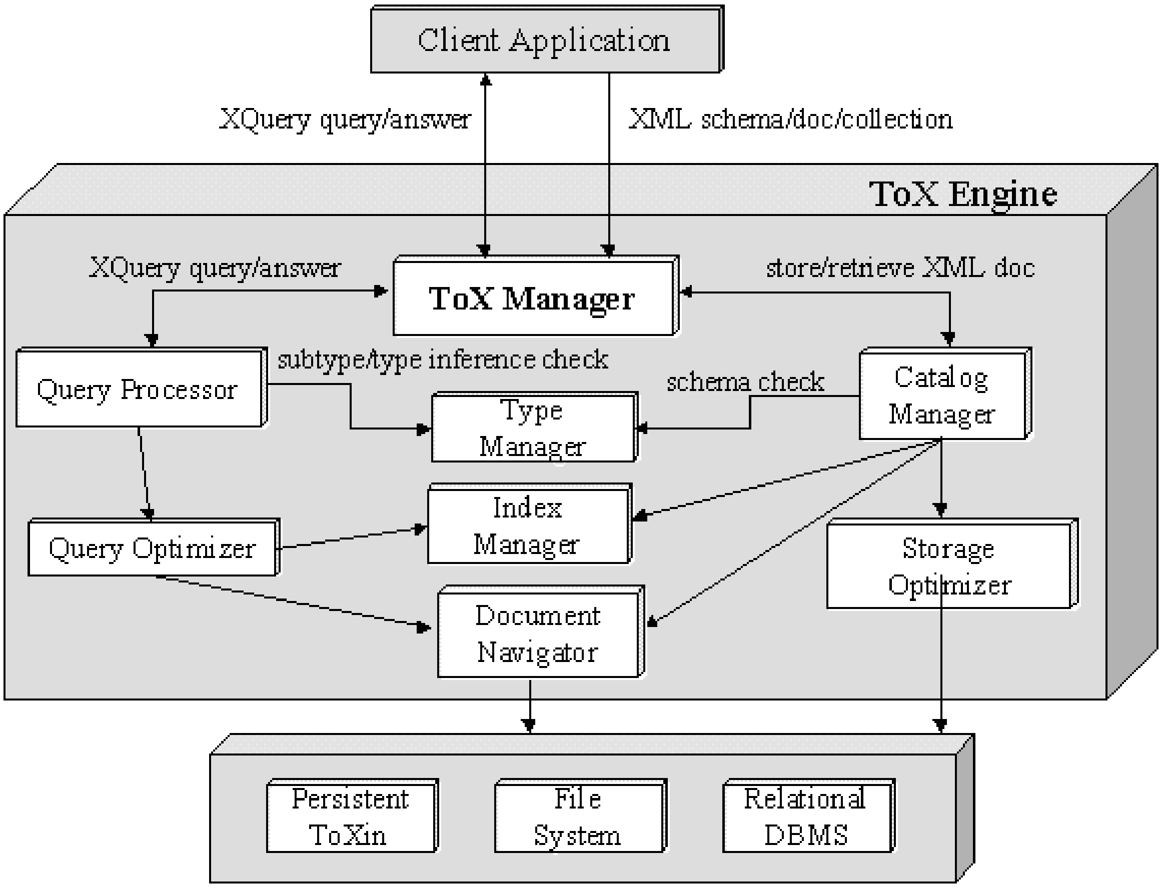


Figure 1.3: ToX System Architecture

Chapter 2

XML Types and Subtyping

In this chapter, we will first introduce some terminology related to type systems, then we will survey the major subtyping proposals for XML types.

2.1 Introduction

A *type* is a characterization of the range of values that a program variable can have during the execution of a program. A *typed language* is a language where types can be assigned to variables. A *type system* is the component of a typed language that keeps track of the types of variables and expressions in the program. The purpose of a type system is to prevent execution errors during the running of a program. Typed languages detect errors at compile time by performing a checking process called *typechecking*. A type system is made of a set of *judgments* and a collection of *type rules*. Judgments are formal statements of the form:

$$\Gamma \vdash \Lambda$$

where Λ is an assertion, and the free variables of Λ are declared in the static typing environment Γ. A *typing judgment* is a judgment that asserts that a term M has a type

A in environment Γ:

$$\Gamma \vdash M : A$$

A judgment is either *valid* (e.g., $\Gamma \vdash false : Boolean$), or *invalid* (e.g., $\Gamma \vdash false : Integer$). In order to assert the validity of certain judgments based on other judgments, judgments can be organized as a set of *type rules*, where each type rule is written as follows:

$$\frac{\Gamma_1 \vdash \Lambda_1 ... \Gamma_n \vdash \Lambda_n}{\Gamma \vdash \Lambda}$$

This rule states that if the judgments above the horizontal line (known as *assumptions*) are valid, then the judgment below the line (known as *the conclusion judgment*) is also valid. For instance, the following rule states that if both A and B are integers, then the type of the expression $A + B$ is also an integer:

$$\frac{\Gamma \vdash A : Integer \quad \Gamma \vdash B : Integer}{\Gamma \vdash A + B : Integer}$$

Most type systems support two important relations between types called *type equivalence* and *subtyping*. The type equivalence relation, written as $X \equiv Y$, consists of checking that two type expressions X and Y have either structural equivalence or name equivalence. For instance the following types X and Y are structurally equivalent but they have different variable names so they are not equivalent by name:

```
type X = Integer
type Y = Integer
```

The subtyping relation, written as $X <: Y$, consists of checking that a type expression X is included in a type expression Y, where types are considered a set of values. We say that type X is a subtype of type Y, or equivalently, type Y is a supertype of type X. For instance, X is a subtype of Y, when X and Y are defined as:

```
type X = Integer
```

```
type Y = Any
```

Where *Any* means any type.

Cardelli presents a detailed overview of type systems and the equivalence and subtyping problems [12].

2.2 Subtyping Proposals for XML

In this section, we will present a survey of subtyping proposals for XML types. Then, we describe one of these proposals called XDuce in more detail (section 2.2.8) because we adopt their subtyping definition in this thesis. We chose the XDuce algorithm because it was adopted by the XQuery team as a W3C standard [25].

2.2.1 XML Schema

We have introduced XML Schema in chapter 1. In this section, we will present XML Schema types in more detail, and we will discuss the different notions of subtyping that are present in XML Schema. XML Schema types are used to create XML Schema elements in the same way that classes are used to create objects in object-oriented languages. There are two categories of XML Schema types: simple and complex types. The W3C recommendations [51] [50] provide a detailed description of simple and complex types. A simple type has three characteristics: a *value space*; a *lexical space*; and a set of *facets*. The value space is the set of distinct values that the type can have. The lexical space is the set of valid literals for the datatype. A facet is a property that characterizes the value space of the datatype. A facet can be *fundamental* (i.e., it semantically characterizes the value space) or *constraining* (i.e., it restricts the value space of an existing datatype to a defined subset). Simple types are defined by derivation of other datatypes though the use of (i) a *list*, a finite-length sequence of values of an atomic datatype; (ii) a *union*, a disjunction of types; or (iii) a constraining facet. In the last case, the type is said to be *derived by restriction*. The following simple type declaration defines a constraining facet on the length of the string of element *bookTitleSimple*, limiting the length of the content of element *bookTitleSimple* to at most eight characters:

```
<simpleType name="bookTitleSimple">
   <restriction base= "xs:string">
      <xs:maxLength value="8"/>
   </restriction>
</simpleType>
```

Each complex type is derived either from a simple type or another complex type. Two different kinds of constraints can be applied to complex types: an *order constraint* and an *occurrence constraint*. The first defines the appearance of elements within the complex types with the use of three indicators. The indicator *all* written as "<xs:all>" means child elements can appear in any order. The indicator *sequence* written as "<xs:sequence>" states the exact order of child elements. And the indicator *choice* written as "<xs:choice>" means that only one of the child elements can occur at once. An occurrence constraint defines how often an element can occur using the indicators *minOccurs* and *maxOccurs*. The following example is a type declaration for complex type *bookTitleComplex* that is derived from the simple type *bookTitleSimple* described above:

```
<complexType name="bookTitleComplex">
   <xs:choice>
      <xs:element name = "title" type="bookTitleSimple"/>
      <xs:element name = "title" type="xs:string"/>
   </xs:choice>
</complexType>
```

XML Schema allows the definition of *model groups*, which consist of element and attribute declarations that can be used to build a complex type. A group consists of *particles*, which can be element declarations or model groups themselves. For example, the following XML Schema fragment declares the element group "bookGroup":

```
<xs:group name="bookGroup">
  <xs:sequence>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="author" type="xs:string"/>
  </xs:sequence>
</xs:group>
```

The element declaration <xs:element name="title" type="xs:string"/> is one of the particles in the group declaration. The following complex type declaration uses the group "bookGroup" to define the type "book":

```
<xs:complexType name="book">
  <xs:sequence>
    <xs:group ref="bookGroup"/>
    <xs:element name="year" type="xs:date"/>
  </xs:sequence>
</xs:complexType>
```

A group can be a sequence of particles as we have seen in the previous example, or it can be a choice between a set of particles or an unordered set of particles.

XML Schema supports two notions of subtyping: *restriction subtyping*, and *extension subtyping*.

Definition 1 (Restriction Subtyping) *A type t is a subtype of a type t', written as $t <:res\ t'$, if the instances of type t are a subset of the instances of type t'.*

Definition 2 (Extension Subtyping) *A type t is a subtype of a type t', written as $t <:ext\ t'$, if type t is derived from type t' by appending particles at the end of the base type definition or adding attribute declarations.*

For instance, the type *bookTitleSimple* described previously is a subtype of the builtin type *xs : string* by restriction subtyping, and the type *extendedBook* defined below is a subtype of the type *book* defined above by extension subtyping:

```
<xs:complexType name="extendedBook">
   <xs:complexContent>
      <xs:extension base = "book">
         <xs:sequence>
            <xs:element name="publisher" type="xs:string"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

Subtyping in XML Schema can be controlled with the use of the *final* attribute in a type definition. The following example prevents any form of derivation (extension, restriction or both) from type "book":

```
<xs:complexType name="book" final="#all">
   <xs:sequence>
      <xs:group ref="bookGroup"/>
      <xs:element name="anotherElementInBook" type="xs:string"/>
   </xs:sequence>
</xs:complexType>
```

A more granular mechanism for controlling subtyping can be achieved with the attribute *fixed*, which specifies facets that are not allowed to be derived. For example, the following type declaration prevents the length of the element "fixedLength" from being redefined:

```
<xs:simpleType name="fixedLength">
   <xs:restriction base="xs:string">
```

```
      <xs:maxLength value="3" fixed="true"/>
    </xs:restriction>
  </xs:simpleType>
```

Therefore, the following definition is illegal:

```
  <xs:simpleType name="illegalDerivation">
    <xs:restriction base="fixedLength">
      <xs:maxLength value="10"/>
    </xs:restriction>
  </xs:simpleType>
```

XML Schema also provides a mechanism called *substitution groups* that allows an element name, called the *head element*, to be replaced by other element names. Elements in a substitution group must have the same type as the head element, or they can have a type that has been derived from the head element's type. For instance, the following XML Schema fragment states that whenever an element of type "postCode" is expected, elements of type "USZipCode" or "CanadaPostalCode"can be used instead:

```
<element name="USZipCode" type="string" substitutionGroup="postCode"/>
<element name="CanadaPostalCode" type="string" substitutionGroup="postCode"/>
```

where "postCode" is defined as follows:

```
<element name="postCode" type="string"/>
```

2.2.2 XQuery

We introduced the XQuery language in chapter 1. In this section, we will discuss the XQuery type system in more detail.

XQuery types are defined as follows:

Type Declaration	d	::=	"DEFINE ELEMENT" Name{t}	**Element Declaration**
		|	"DEFINE ATTRIBUTE" Name{t}	**Attribute Declaration**
		|	"DEFINE TYPE" Name{t}	**Type Declaration**
Unit Type	u	::=	p	**Simple Type**
		|	"ATTRIBUTE" a{t}	**Attribute**
		|	"ATTRIBUTE" *{t}	**Wildcard Attribute**
		|	"ELEMENT" a{t}	**Element**
		|	"ELEMENT" *{t}	**Wildcard Element**
Type	t	::=	u	**Unit Type**
		|	()	**Empty Sequence**
		|	t , t	**Sequence**
		|	t | t	**Choice**
		|	t & t	**Interleave**
		|	t?	**Optional**
		|	t*	**Zero or More**
		|	t+	**One or More**
		|	x	**Type Reference**

XQuery supports type declarations for elements and attributes and an explicit type declaration that declares types that can be used later to build other element and/or attribute declarations. A unit type is either an XML Schema simple type (e.g., "$xs : integer$" or "$xs : string$"), an attribute or element type with name a and content in t, or an element or attribute type with any name ("*") and content in t. The empty sequence matches only the empty document. The XQuery operators ",", "&", and "|" are equivalent to the XML Schema operators <xs:sequence>, <xs:all>, and <xs:choice> respectively. The Kleene operators "+", "*", and "?" in XQuery are a subset of the occurrence constraints achieved by XML Schema attributes *minOccurs* and *maxOccurs*.

The XQuery type system [25] is based on the XML Schema type system. The main difference between the two type systems is that XML Schema is based on both named and structural typing, whereas XQuery captures only structural typing. For instance, consider the following two XQuery elements defined in two different schemas :

```
DEFINE ELEMENT entry { TYPE Book }
DEFINE TYPE Book {
   ELEMENT title { xs:string },
   ELEMENT author { xs:string }*
}

DEFINE ELEMENT entry { TYPE Article }
DEFINE TYPE Article {
   ELEMENT title { xs:string },
   ELEMENT author { xs:string }*
}
```

The following XML document will conform to both types, whereas in XML Schema, the XML document will validate only with one of the two types:

```
<entry>
   <title>Quickly Generating Billion-Record Synthetic Databases</title>
   <author>Jim Gray</author>
</entry>
```

The reason is that in XQuery the names of the types in the "`DEFINE TYPE`" declarations are ignored and only the structural information is preserved.

The type system of XQuery is exactly the same as the XDuce type system [25]. Moreover, the XQuery team adopted the XDuce subtyping algorithm in their implementation of the query language. Therefore, we will not describe the details of the XQuery subtyping rules since they are equivalent to the subtyping rules of XDuce that will be presented in section 2.2.8.

2.2.3 DTD Containment

Calvanese, Giacomo, and Lenzerini [11] present a notion of containment between DTDs based on Description Logic. The Description Language DL that they use to represent DTDs consists of a set of unary and binary predicates called *concepts* and *roles* respectively. For instance, the name of a DTD element is considered a *concept*, and the relation between the start-tag and end-tag of the element is modeled with a *role*. Atomic roles include roles "f" and "r", which stand for "first" and "rest" respectively. *Complex concepts* can be built from *atomic concepts* using *constructs*. The most important constructs are "$\neg$" for negation, "$\sqcap$" for Boolean AND, "$\sqcup$" for Boolean OR, "$\exists$" for existential quantification, "$\forall$" for universal quantification, "$\circ$" for concatenation, and *id* for the identity construct. For instance, The following DTD declaration:

```
<!ELEMENT title (#PCDATA)>
```

is written in DL as:

$$Title_M \equiv \exists f.StartTitle \sqcap \exists(r \circ id(\exists f.\#PCDATA) \circ r).EndTitle$$

The start-tag of element *title* is represented by $f.StartTitle$. The second component in the element declaration, which specifies the content of element *title* as PCDATA, is modeled by $r \circ id(\exists f.\#PCDATA) \circ r$. That is, any PCDATA concept is delimited by two concepts which are the start-tag and end-tag. The end-tag is described by $EndTitle$.

Two different subtyping definitions were proposed:

Definition 3 (Strong Inclusion) *A DTD d_1 is strongly included in DTD d_2 if the set of all documents conforming to d_1 is included in the set of all documents conforming to d_2.*

Definition 4 (Structural Inclusion) *A DTD d_1 is structurally included in DTD d_2 if, after replacing all the tags in every document conforming to the two DTDs by the unnamed tags* <> *and* < / >, *d_1 is strongly included in d_2.*

Calvanese et al. [11] presented the formal framework for these two subtyping definitions. However, they did not discuss the implementation details of their algorithms. Moreover, the formalism that they use is less expressive than the ones we will discuss in the subsequent sections.

2.2.4 YAT

Cluet, Delobel, Siméon, and Smaga [16] propose a data conversion system called YAT, in which different translators can be combined together into a single conversion program. In order to perform data conversion successfully, programs are typechecked to avoid conflicting translations. For instance, a conflict occurs when the same object has different conversion rules in the different input programs. YAT performs typechecking by providing some support for checking conformance and subtyping relations between unions of trees. However, this support is limited because it does not capture all characteristics of XML types. For instance, the types in Figure 2.1 will not be recognized as equivalent because YAT does not support distributivity laws.

As an extension to the YAT system, Kuper and Siméon [39] present a syntactic definition for subtyping for XML types, similar to the notion of subtyping in programming languages, which consider structural relationships as a basis for subtyping. This notion

DEFINE TYPE t_1{
ELEMENT author {
ELEMENT name {xs:string} |
ELEMENT email {xs:string}
}
}

DEFINE TYPE t_1{ 2
ELEMENT author {
ELEMENT name {xs:string}
} |
ELEMENT author {
ELEMENT email {xs:string}
}
}

Figure 2.1: Types t_1 and t_2 in XQuery Syntax

DEFINE TYPE t_1{
ELEMENT author {
ELEMENT name {xs:string},
ELEMENT email {xs:string}
}
}

DEFINE TYPE t_2{
ELEMENT author {
ELEMENT name {xs:string},
ELEMENT email {xs:string},
ELEMENT address {xs:string}
}
}

Figure 2.2: Types t_1 and t_2 in XQuery Syntax

supports the refinement and extension concepts of XML Schema, but is less powerful than the full inclusion concept.

For instance, although type t_1 is not, intuitively speaking, a subtype of type t_2 (Figure 2.2), they will be considered as subtypes according to the definition of Kuper et al. [39].

2.2.5 Graph Simulation

Buneman, Davidson, Fernandez, and Suciu [9] describe a different approach for subtyping based on graph simulation. Types are modeled as schema graphs, where a schema graph is a rooted, labeled graph, in which the edges are labeled with type names (e.g., *title*), or alternations of type names (e.g., $title|bookname$). Leaf nodes are labeled with atomic data types (e.g., xs:string), and internal nodes are labeled with object identifiers. The subtyping relation between schema graphs is defined as follows:

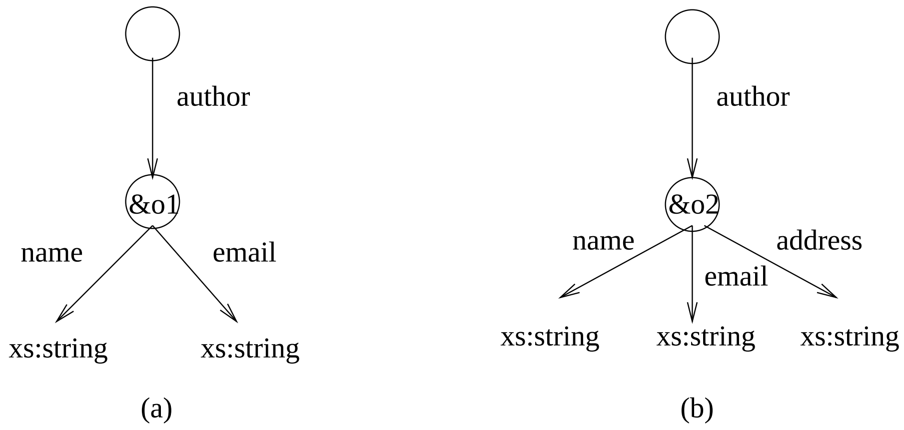


Figure 2.3: Graph Schemas

Definition 5 (Subsumption of Graph Schemas) *Let U be the universe of constants ($U = Int \cup String \cup Bool \cup \ldots$).Given two graph schemas S, S' we say that S subsumes S', in notation $S <: S'$, if there exists a binary relation $<:$ between the nodes of S and S' such that: (1) $v_0 <: v_0'$, where v_0, v_0' are the roots of S,S', (2) whenever $u <: u'$, for every labeled edge (u, p, v) in S and every $a \in U$ s.t. $U \models p(a)$, there exists an edge (u', p', v') in S' s.t. $U \models p'(a)$ and $v <: v'$. S and S' are equivalent if $S <: S'$ and $S' <: S$.*

Figure 2.3 shows the graph schema representation for types t_1 (graph (a)) and t_2 (graph (b)) depicted in Figure 2.2. According to the definition above, graph (b) simulates graph (a). Therefore, graph (a) is a subtype of graph (b).

2.2.6 Union Types for Semistructured Data

Buneman and Pierce [10] propose a type system for semistructured data that is based on traditional complex object types (i.e., set and record types), with the addition of an untagged union type. The subtyping rules for sets and records are similar to subtyping in conventional type systems, and the subtyping of untagged union types supports distributivity laws. This makes this subtyping proposal more powerful than the ones described thus far (except for XQuery and XML Schema).

However, the limitations of this proposal are that it does not handle recursive types, and that it considers record types as unordered types, whereas in XML, sequences are considered ordered.

2.2.7 Labeled Ordered Tree Definitions

Papakonstantinou and Vianu [44] and Milo, Suciu, and Vianu [43] analyze the complexity of typechecking for XML, and describe a similar notion of types and subtyping.

Papakonstantinou et al. model DTDs as labeled ordered tree definitions (LTDs) [44]. LTDs are more expressive than regular DTDs because they allow context free languages instead of regular languages, and they introduce a notion of subtyping based on union types. LTDs are as expressive as YAT's type representation, and they are more powerful than graph schemas [9].

Milo et al. model DTDs using regular tree languages, and perform typechecking using a k-pebble transducer [43]. A k-pebble transducer is an abstract model that expresses transformations on labeled ordered trees. The typechecking between an input type t_1 and a type t_2 is performed by checking that the tree resulting from transformation of t_1 using the k-pebble transducer is included in the tree representation of t_2.

Both studies present a complete theoretical analysis of the typechecking problem. However, they did not address implementation issues.

2.2.8 XDuce

In this section, we describe the subtyping proposal that we adopt in the rest of this thesis.

Hosoya and Pierce [33] propose an ML-like functional programming language for XML called XDuce. XDuce has two important features: a subtyping algorithm for regular expression types [35], and a type inference algorithm for regular expression pattern matching [34]. The type system and subtyping algorithms of XDuce were adopted by the XQuery team as a W3C standard [25]. (The support for attributes, the interleave operator, and the atomic type "xs:AnyType" were left by the authors as future work.)

In this section, we will give a detailed description of the XDuce system. First, we will describe XDuce regular expression types. Then, we will explain the equivalence and subtyping rules in XDuce.

XDuce types

Let E be the set of type definitions that define the bindings of *type names*. E contains definitions of the form:

$$\texttt{type} \quad X = T$$

where X is a type name (or variable) and T is an XDuce type described as follows (base types are omitted for brevity):

T	::=	()	**empty sequence**
	|	X	**type name**
	|	l[T]	**label**
	|	T , T	**concatenation**
	|	T | T	**union**

We can see that the XDuce types are equivalent to XQuery types. The XQuery operators "," and "|" map to the *concatenation* and *union* XDuce operators, and the Kleene operators "*", "+", and "?" can be represented using *concatenation*, *union*, and *type names*. We will see an example that illustrates the representation of Kleene operators in XDuce shortly.

Type definitions in E can contain any previously defined type name and can be recursive. For instance, consider the following type declaration written in XQuery syntax:

```
DEFINE TYPE t1 {
   ELEMENT person {
      ELEMENT name {xs:string},
```

```
        ELEMENT tel {xs:string}*
    }
}
```

This declaration is equivalent to the XDuce type declaration:

```
type t1 =
    person [
        name [String],
        X_tel*
    ]
```

where X_{tel*} is declared as follows:

$$\texttt{type} \quad X_{tel*} = tel[String], X_{tel*} \quad | \quad ()$$

Note that the Kleene operators "*", "+", and "?" can be expressed in the XDuce language, but were not included in the formal XDuce type representation that we have just seen because they can be derived from it. We have seen the XDuce representation of the Kleene operator "*". Below we show the XDuce representation of both "+" and "?":

$$\begin{aligned} \texttt{type} \quad X_{tel+} &= tel[String], X_{tel*} \\ \texttt{type} \quad X_{tel?} &= tel[String] | () \end{aligned}$$

Types in XDuce are represented by unranked trees (i.e., trees where a node may have an arbitrary number of children). To simplify formalisms and proofs, Hosoya, Vouillon, and Pierce [35] propose a translation mechanism that converts the XDuce types defined earlier, called external types, into an internal form. To represent internal type expressions, we assume a countably infinite set of (internal) type states ranged over by X. The definition of an internal type expressions T is:

$$
\begin{array}{lll}
T ::= \emptyset & & \text{empty set} \\
\quad\ \ \epsilon & & \text{leaf} \\
\quad\ \ \beta & & \text{builtin} \\
\quad\ \ T|T & & \text{union} \\
\quad\ \ l(X,X) & & \text{label}
\end{array}
$$

There is an isomorphism between internal types and external XDuce types: ϵ is called *leaf* and corresponds to the empty sequence; β represents builtin types; and $l(t,t')$ corresponds to a sequence where l is the label of its head, t is the content of l and t' is the remainder of the sequence. For instance, the type t_1 that we just described has the following internal representation:

$$
\begin{array}{ll}
person(X_1, X_0) & \text{such that:} \\
 & M(X_0) = \epsilon \\
 & M(X_1) = name(X_2, X_3) \\
 & M(X_2) = \beta \\
 & M(X_3) = tel(X_2, X_3)|\epsilon
\end{array}
$$

where M is a finite mapping from type states to internal type expressions.

The details of the translation function are given by Hosoya et al. [35].

Since XQuery types map naturally to XDuce types, we will use this internal representation in the remainder of this thesis to describe XML types.

XDuce Equivalence and Subtyping Rules

The equivalence rules used in the XDuce type system are defined as follows:

$$
\begin{array}{lcl}
T \mid \emptyset & \equiv & T \\
T \mid T & \equiv & T \\
(T \mid U) \mid R & \equiv & T \mid (U \mid R) \\
T \mid U & \equiv & U \mid T \\
T \mid U & \equiv & T \mid R \;\; if \;\; U \equiv R
\end{array}
$$

The first rule states that whenever an empty type occurs in a union of types, it can be eliminated safely. The second rule eliminates redundancy in a union. The third and fourth rules specify associativity and commutativity of the union operator, and the fifth rule states that two types that are structurally equivalent are substitutable in a union operation.

In addition to these equivalence rules, XDuce includes a formalism for subtyping. The XDuce subtyping definition is simple and intuitive:

Definition 6 (Subtyping Definition) *The set of a type t is the range of values that a variable of type t can assume. A type t_1 is a subtype of type t_2, written as $t_1 <: t_2$, if the set of t_1 is included in the set of t_2.*

Figure 2.4 depicts the subtyping rules of XDuce. The first rule, called EMPTY, states that the null type is a subtype of any type. The expression $\Pi \vdash^{\bullet} \emptyset <: T \Rightarrow \Pi'$ states that the initial environment of types represented by Π is augmented with the new rule $\emptyset <: T$ to produce a new environment represented by Π'. The SPLIT rule handles the case where the left-hand side of the subtyping relation is a union type. In this case, the subtyping relation holds if each type in the union is a subtype of the right-hand side. The rule LEAF states that the empty sequence type (i.e., the ϵ type) can only be a subtype of a type that contains ϵ. These three rules are followed by a definition of the cardinality of a type ($card$), which is used in the three remaining subtyping rules: PRUNE, PRUNE-LEAF, and REC. Rules PRUNE and PRUNE-LEAF remove from the right-hand side a leaf type ϵ and all types that have a label that differs from the label of the left-hand side, or is not a supertag of it, where the supertag relation is the inverse of the subtag relation, which is similar to the XML Schema relation between elements declared in a substitution group g and the head element of g. (See section 2.2.1 for more details on substitution groups.) The condition $card(R') < card(R)$ makes sure that the algorithm terminates by assuming a smaller type with each rule application. All the remaining cases are handled by the rule REC: The right-hand side is the union of types $l_j(Y_j, Y'_j)$ that have labels l_j that are equal to or supertags of l.

To better understand the intuition behind the REC rule, first consider this special case:

$$l(T, U) \quad <: \quad l(R_1, S_1) \quad | \quad l(R_2, S_2) \tag{2.1}$$

The two arguments of any labeled type $l(R, S)$ can be represented as the cross product $R \times S$, or equivalently, as $(R \times \tau) \cap (\tau \times S)$, where τ is the maximal type that denotes all possible types. So, the right-hand side of the subtyping relation 2.1 can be rewritten as:

$$(l(R_1, \tau) \cap l(\tau, S_1)) \quad | \quad (l(R_2, \tau) \cap l(\tau, S_2))$$

This is equivalent in turn to:

$$\begin{array}{l} (l(R_1, \tau)|l(R_2, \tau)) \cap \\ (l(R_1, \tau)|l(\tau, S_2)) \cap \\ (\tau, S_1)|l(R_2, \tau)) \cap \\ (\tau, S_1)|l(\tau, S_2)) \end{array} \tag{2.2}$$

Each of these four clauses can be rewritten as:

$$(|_{i \in I} l(R_i, \tau)) \quad | \quad (|_{i \in \overline{I}} l(\tau, S_i)) \tag{2.3}$$

where I is a subset of $\{1,2\}$ (because the union in Equation 2.1 has two operands) and $\overline{I} = \{1,2\}|I$. Expression 2.2 is the intersection of expressions of the form 2.3 for all subsets of $I = \{1,2\}$. Therefore, the original subtyping relation 2.1 reduces to checking the following subtyping relation for all subsets of I:

$$l(T,U) \quad <: \quad (|_{i\in I} l(R_i,\tau)) \quad | \quad (|_{i\in\overline{I}} l(\tau,S_i))$$

which is also equivalent to:

$$l(T,U) \quad <: \quad l(|_{i\in I} R_i,\tau) \quad | \quad l(\tau,|_{i\in\overline{I}} S_i))$$

Therefore, to verify the subtyping relation 2.1, we just need to check that:

$$T <: |_{i\in I} R_i \quad \text{or} \quad U <: |_{i\in\overline{I}} S_i$$

To understand why, suppose that T is not a subtype of R and U is not a subtype of S, such that $R = |_{i\in I} R_i$ and $S = |_{i\in\overline{I}} S_i$. We can find a type t such that $t <: T$ and t is not a subtype of R. We can also find a type u such that $u <: U$ and u is not a subtype of S. This means that $l(t,u) <: l(T,U)$ but neither $l(t,u) <: l(R,\tau)$ nor $l(t,u) <: l(\tau,S)$ which contradicts

$$l(t,u) \quad <: \quad l(T,U) <: \quad l(R,\tau) \quad | \quad l(\tau,S)$$

from equation 2.2.8. Therefore, to check the subtyping relation 2.1, we just need to check that, for all subsets of $I = \{1,2\}$, the following relation holds:

$$T <: |_{i\in I} R_i \quad \text{or} \quad U <: |_{i\in\overline{I}} S_i$$

This intuition is generalized in rule REC.

The complexity of the subtyping problem for XML is the same as the complexity of the language inclusion problem of tree automata, which is known to be EXPTIME-complete [54]. The XDuce algorithm is the only algorithm known so far that has a good performance for checking subtyping between XML types that occur in practice.

(EMPTY)
$$\frac{}{\Pi \vdash^{\bullet} \emptyset <: T \Rightarrow \Pi}$$

(SPLIT)
$$\frac{\Pi \vdash^{\bullet} T <: U \Rightarrow \Pi' \qquad \Pi' \vdash^{\bullet} T' <: U \Rightarrow \Pi''}{\Pi \vdash^{\bullet} T \mid T' <: U \Rightarrow \Pi''}$$

(LEAF)
$$\frac{T \equiv \epsilon \mid T'}{\Pi \vdash^{\bullet} \epsilon <: T \Rightarrow \Pi}$$

$$\begin{array}{lcl} card(T_1 \mid T_2) & = & card(T_1 + card(T_2)) \\ card(T) & = & 1 \quad if \quad T = \epsilon \quad \text{or} \quad l(X, X') \quad \text{or} \quad \emptyset \end{array}$$

(PRUNE)
$$\frac{\begin{array}{c} R \equiv l'(U, U') \mid R' \qquad card(R') < card(R) \\ l\ not\ subtag\ of\ l' \qquad \Pi \vdash^{*} l(T, T') <: R' \Rightarrow \Pi' \end{array}}{\Pi \vdash^{*} l(T, T') <: R \Rightarrow \Pi'}$$

(PRUNE-LEAF)
$$\frac{\begin{array}{c} R \equiv \epsilon \mid R' \qquad card(R') < card(R) \\ \Pi \vdash^{*} l(T, T') <: R' \Rightarrow \Pi' \end{array}}{\Pi \vdash^{*} l(T, T') <: R \Rightarrow \Pi'}$$

(REC)
$$\frac{\begin{array}{l} \text{For all } 1 \leq j \leq n,\ l \prec l_j \\ \text{For all } 1 \leq i \leq 2^n, \text{ either} \\ \Pi_{i-1} \vdash T <: |_{j \in I_i^n} U_j \Rightarrow \Pi_i \quad \text{or} \quad \Pi_{i-1} \vdash T' <: |_{j \in \overline{I_i^n}} U'_j \Rightarrow \Pi_i \\ \text{where } T = M(X) \quad T' = M(X') \quad U_j = M(Y_j) \quad U'_j = M(Y'_j) \end{array}}{\Pi_0 \vdash^{\bullet} l(X, X') <: l_1(Y_1, Y'_1) \mid \ldots \mid l_n(Y_n, Y'_n) \Rightarrow \Pi_{2^n}}$$

Figure 2.4: XDuce Subtyping Rules [35]

Chapter 3

The ToX Type Index (Tindex)

Several indexing mechanisms have been devised for XML and semistructured data including the ToXin index developed by Rizzolo and Mendelzon [52], the Fast index developed by Cooper, Sample, Franklin, Hjaltason, and Shadmon [18], the Index Fabric proposed by Cooper and Shadmon [19], and the DataGuides index proposed by Goldman and Widom [29]. However, we are not aware of any indexing method that has been proposed for XML types or regular tree expressions in general. In this chapter, we will present a new indexing mechanism for XML types called Tindex. First, we will give an overview of the related literature, then we will describe the indexing method in more detail.

3.1 Related Work

The challenge involved in indexing XML types is dealing with the expressive power of the terms to be indexed, which are regular expression types. In this section, we will give an overview of the literature from the programming languages field that addresses the indexing problem for expressive terms like graphs or first-order terms.

3.1.1 Fingerprints

Fingerprints are short tags for large objects. They have the characteristic that if two fingerprints are different, then their corresponding objects are different, and there is a very small probability that two different objects have the same fingerprint. Rabin [8] proposes a fingerprinting scheme for Directed Acyclic Graphs (DAG), which was used by the software development environment $Vesta$ to identify objects derived during compilation or linking. Such a scheme could be extended to solve the problem of efficient lookup for an XML type. However, fingerprinting would not support the identification of supertypes of an XML type.

3.1.2 Theorem Proving

First, we will start by defining some terminology, then we will describe the main indexing methods used in the field of theorem proving.

Notation and Definitions

A *term* is either an individual constant, an individual *variable*, or a function whose arguments are also terms. A *variable* denotes any symbol whose referent varies or is unknown. We denote variables using the wildcard symbol '*'. For instance, $f(a, *)$ is a term that has a constant a as its first argument and a variable as its second argument.

A *clause* is a finite set of terms. A clause which is a singleton set is called a *unit clause*, and a clause that has cardinality ≥ 2 is called a *multiliteral clause*. Clauses are denoted either as a sequence of their terms, $t_1, ..., t_n$, or as a disjunction, $t_1|...|t_n$. For the purposes of this research, we will consider only clauses formed by a disjunction of terms.

Definition 7 (Term Indexing Problem) *Given a set S (called the set of indexed terms), a binary relation R over terms (called the retrieval condition), and a term t (called the query term), identify the subset M of S consisting of all terms l such that*

$R(l,t)$ holds [45].

Multiterm indexing is a generalization of term indexing, where the query or indexed terms can be multiliteral clauses.

Term (or multiterm) indexing is used to speed up many retrieval problems such as retrieving terms *unifiable* with the query term, *instances* of a query term, or *generalizations* of a query term. The first retrieval problem is a combination of the last two retrieval conditions. Given a query type t, and an indexed set S, the formal definitions of these retrieval problems are as follows:[46]

A *substitution* α is a mapping from variables to terms. $t\alpha$ denotes the term resulting from replacing all variables x in t by $\alpha(x)$.

The *instances problem* consists of selecting all terms l in S such that there exists a substitution α that satisfies $l = t\alpha$. For example, the term $f(a,b)$ is an instance of the term $f(a,*)$.

The *generalization problem* consists of selecting all terms l in S such that there exists a substitution α that satisfies $l\alpha = t$. For instance, the term $f(*,*)$ is a generalization of the term $f(a,*)$.

The *unification problem* consists of selecting all terms l in S such that there exists a substitution α that satisfies $l\alpha = t\alpha$. For instance, the terms $f(a,b)$ and $f(*,*)$ are both unifiable with the term $f(a,*)$.

Indexing Methods

In theorem proving, retrieval operations like unification, pattern matching or subsumption are performed on first-order logic terms. These operations often are performed on a query term and a large collection of terms. Therefore, several term indexing techniques have been proposed to speed up the retrieval of candidate terms. The use of term indexing techniques improves the performance of theorem provers like OTTER [15] by several orders of magnitude [41].

In path indexing [46] [40] [30], terms are represented using trees, and all the root-to-leaf paths of the term trees are merged into a trie which is used to perform the retrieval operation. The main advantages of path indexing are its small memory usage, its fast retrieval of candidates because it does not involve backtracking, and its efficient insertion and deletion operations. The main disadvantage of this approach is the overhead of set operations that are used heavily to produce intermediate results.

In discrimination tree indexing [31] [15] [40], instead of considering all root-to-leaf paths in the tree representation of the term, the whole term tree is traversed using an a-priori order (for example a preorder traversal), generating an encoding for the term. The encodings of all the terms are then merged into a trie. The advantage of this method is that it eliminates the set operations necessary in path indexing. However, it introduces additional overhead in the retrieval of candidate terms because of the backtracking involved. Moreover, compared to path indexing, the index insertion and deletion operations are more expensive, and the index uses much more space.

Automata-driven indexing [14] [46] is another indexing mechanism used in theorem provers. First, a string-matching automaton is built from the preorder strings representing the terms, then the query term passes through the automaton, and each final state corresponds to a set of terms. This method shares the same advantages and disadvantages with the path indexing technique. Moreover, automata-driven indexing offers the unique advantage of retrieving candidates of the subterms of a query in addition to the root match that the other methods provide. This technique improved the performance of the ALS Prolog system [46].

3.2 Tindex Overview

In this section, we will present a new type indexing technique called Tindex. Tindex is an extension to the automata-driven term indexing technique [46]. First we define some terminology, then we map the problem of supertype selection to the unification problem in term indexing. Next, we extend the automata-driven technique to support disjunctions of terms. We provide a modified algorithm, an example illustrating how Tindex works and a correctness proof showing that Tindex selects all possible supertypes from the collection.

3.2.1 Mapping Supertype Selection to the Unification Problem

First, we will give a general definition of the type indexing problem, then we will define the supertype selection problem by mapping it to the unification problem in term indexing.

Definition 8 (Type Indexing Problem) *Given a set S of types, a binary relation R over types, and a type t, identify the subset M of S consisting of all types l such that $R(l, t)$ holds.*

Note that an XML type, when converted to the internal representation, forms a multiliteral term. In particular, element types map to functions of arity two, builtin types and the empty sequence type ϵ to functions of arity zero, and unions to a disjunctive clause. Type states can be mapped to variables in the following way: First, each type state is replaced by its corresponding type expression. If the type state has a recursive type definition, only a first-level unrolling is performed. Then, type states that persist after these transformations are mapped to variables and are represented with a "$*$". The following example illustrates these transformations on type t, which is declared in XQuery syntax as :

```
DEFINE TYPE t {
   ELEMENT person {
      ELEMENT name {xs:string},
      ELEMENT tel {xs:string}*
   }
```

}

The internal representation of type t is:

$$person(X_1, X_0) \qquad \text{where:}$$

$$M(X_0) = \epsilon$$

$$M(X_1) = name(X_2, X_3)$$

$$M(X_2) = \beta$$

$$M(X_3) = tel(X_2, X_3)|X_0$$

The representation of t, after all transformations, is:

$$t = person(name(\beta, tel(\beta, *)|\epsilon), \epsilon)$$

We have seen how the type indexing problem maps to the multiterm indexing problem. Now, we will see how the supertype selection problem can be solved using a special case of the multiterm indexing problem (i.e., the one with unification as the retrieval condition). First, we define the supertype selection problem:

Definition 9 (Supertype Selection Problem) *Given a set S of types, and a query type t, identify the subset M of S consisting of all types l such that $t <: l$.*

Note that the supertype selection problem is a special case of the type indexing problem where the binary relation is the subtype relation. We will show that the supertype selection problem can be solved using unification as the retrieval condition of the multiterm indexing problem. In fact, we will see that unification will return a superset of set M. Then, to complete the supertype selection process, we need to eliminate false positives. Therefore, the solution we propose to solve the supertype selection problem consists of two passes: Pass 1 involves retrieving candidate types from the index using unification, and Pass 2 consists of applying the subtyping rules in a sequential manner on the filtered set, which, in general, is much smaller than the original set of types in the index.

Unification is concerned with mapping variables to terms. If no variables are present in the query or indexed terms, unification simply retrieves instances of the query term. However, in the presence of variables, unification will return both instances and generalizations of the query term. Note that unification for XML types in the absence of variables is simply the selection of the types equivalent to the query type. However, when recursive type definitions are present (e.g., type t defined above), the unification returns all supertypes of the query type in addition to some false positives. To better illustrate this idea, we consider the query type t defined above and the two types t_1 and t_2 defined as:

```
DEFINE TYPE t1 {
   ELEMENT person {
      ELEMENT name {xs:string}*,
      ELEMENT tel {xs:string}
   }
}

DEFINE TYPE t2 {
   ELEMENT person {
      ELEMENT name {xs:string}*,
      ELEMENT tel {xs:string}*
   }
}
```

The internal representation of type t_1 is:

$$person(X_4, X_0) \qquad \text{where:}$$

$$M(X_0) = \epsilon$$

$$M(X_2) = \beta$$

$$M(X_4) = name(X_2, X_4)|tel(X_2, X_0)$$

The internal representation of type t_2 is:

$$person(X_5, X_0) \qquad \text{where:}$$

$$M(X_5) = name(X_2, X_5)|X_6$$

$$M(X_6) = tel(X_2, X_6)|X_0$$

with X_2 and X_0 as defined above.

Note that the internal representation of t_1 is substantially different from that of t_2 even though the difference between their external representations is just one character (i.e., the Kleene star in element tel). The reason is that the Kleene star in element tel is translated into both a union and a recursion. The Kleene star states that element tel can be repeated zero or more times. The union $tel(...) \mid \epsilon$ models the case of no repetition of element tel, whereas the recursion $tel(X_2, X_6)$ models the case of one or more repetitions of element tel.

After replacing type states X_i with their corresponding type definitions, the type states that persist are considered variables, and are represented by a star (*):

$t = person(name(\beta, tel(\beta, *)|\epsilon), \epsilon)$

$t_1 = person(name(\beta, *)|tel(\beta, \epsilon), \epsilon)$

$t_2 = person(name(\beta, *)|tel(\beta, *)|\epsilon, \epsilon)$

Based on the idea of unification, both t_1 and t_2 will be selected as supertypes of type t. Type t_2 is indeed a supertype of t, however t_1 is a false positive. To understand why t_1 was selected, note that the first argument of the label *person* in type t_1 is the union of two types, the first element of that union is $name(\beta, *)$. Since * matches any type, it matches in this case $tel(\beta, *)|\epsilon$. So, the first argument of the label *person* in type t is subsumed by $name(\beta, *)$, which in turn is subsumed by the union $name(\beta, *)|tel(\beta, \epsilon)$. Therefore, t_1 generalizes t, and is selected as a potential supertype.

There are several implementations for building term indexes and several algorithms for performing different retrieval operations. We are not aware of any multiliteral indexing mechanism that will solve the supertype selection problem. However, Ramesh et al. [46] provide an indexing method for unit clauses, and a unification retrieval algorithm, that have improved the performance of the prolog compiler ALS. Moreover, they provide a formal framework for the indexing problem. So we decided to extend their work. In particular, we will extend their index construction technique to support disjunctive clauses, their retrieval algorithm based on unification to solve the supertype selection problem, and their formalism to prove the correctness of our retrieval algorithm.

To solve the supertype selection problem efficiently, M' needs to be minimized. Ideally, we would like M' to coincide exactly with M (i.e., the indexing performs perfect filtering). In fact, our indexing mechanism will provide perfect filtering in all cases, except in the presence of repetition (i.e., recursive variables) where it returns a superset of the ideal result, which means that some of the types returned are not supertypes of the query type. That is why we added Pass 2 in our supertype selection algorithm.

3.2.2 Tindex Construction

In constructing Tindex, we take advantage of the analogy between type indexing and string matching. First, we represent each type as a string. Then, we merge the string representations of all types into a single structure similar to the Fredkin trie structure

```
DEFINE TYPE t1 {
    ELEMENT author {
        ELEMENT name {xs:string},
        ELEMENT email {xs:string}*
    }
}
```

```
DEFINE TYPE t2 {
    ELEMENT author {
        ELEMENT name {xs:string},
        ELEMENT email {xs:string} |
        ELEMENT tel {xs:string}
    }
}
```

Figure 3.1: Types t_1 and t_2

used in string matching [28]. The key advantage of using a trie representation is that operations for matching against multiple strings are factored out. To produce the string representation of a type, we perform steps 1 and 2 on the tree representation of the type as detailed below. The set of root-to-leaf paths in the resulting structure, which we also call a trie, produces a string representation of the original type. Note that integers are used to preserve information about the position of each label making up the type.

The details of the construction of Tindex T are as follows:

1. Construct a type tree for each type in the environment. Nodes of the tree are labeled as : variables, elements, builtin types or as a union operator. Consider types t_1 and t_2 defined in Figure 3.1 as examples. Figure 3.2 shows their corresponding type trees.

2. Preprocess each type tree into a trie. The trie is constructed in three steps:

   (a) First, replace all nodes labeled by variables in the tree by their corresponding trees. If a variable is recursive, then replace it with a first-level unrolling of its tree representation, and mark it as unrolled. Repeat until all recursive variables in the tree are marked and non-recursive variables are replaced by their corresponding type expressions. Replace all marked variables with a star ($\star$). Figure 3.3 shows these transformations for type t_1.

   (b) Second, assign integer labels to every edge in the tree.

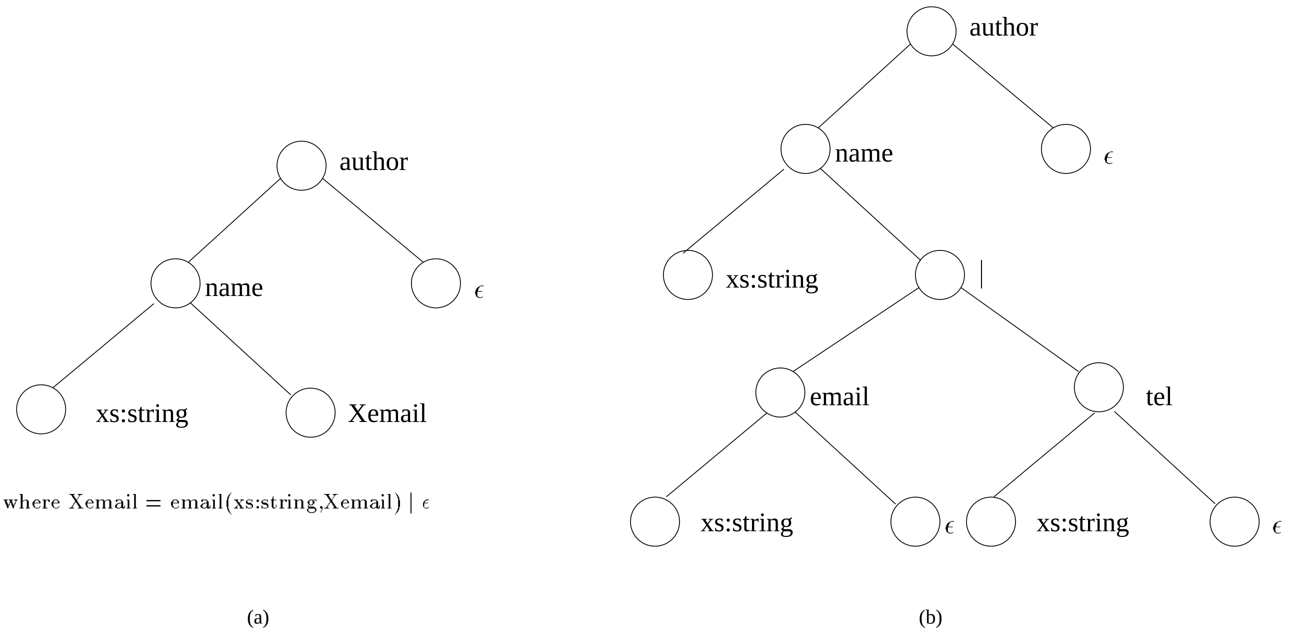


Figure 3.2: Tree Representation for Types t_1 (a) and t_2 (b)

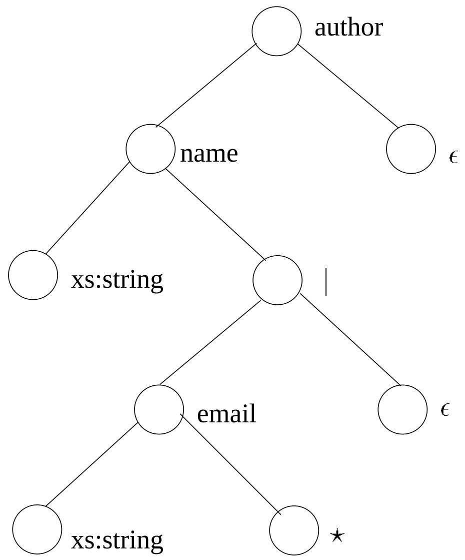


Figure 3.3: First Level Unrolling of Variable Xemail in Type t_1

(c) Third, split every node labeled by a functor or a star ($\star$), and whose parent is not a union operator, into two nodes connected by an edge labeled with the functor name. For each node that has union as its parent, replace the label of the edge connecting it to its parent with the label of the node. It is valid to have edges with the same labels. These edges will be factored into one edge when inserted into the index trie. Finally, remove the labels from all nodes. Figure 3.4 shows the trie of type t_1 and Figure 3.5 shows the trie of type t_2.

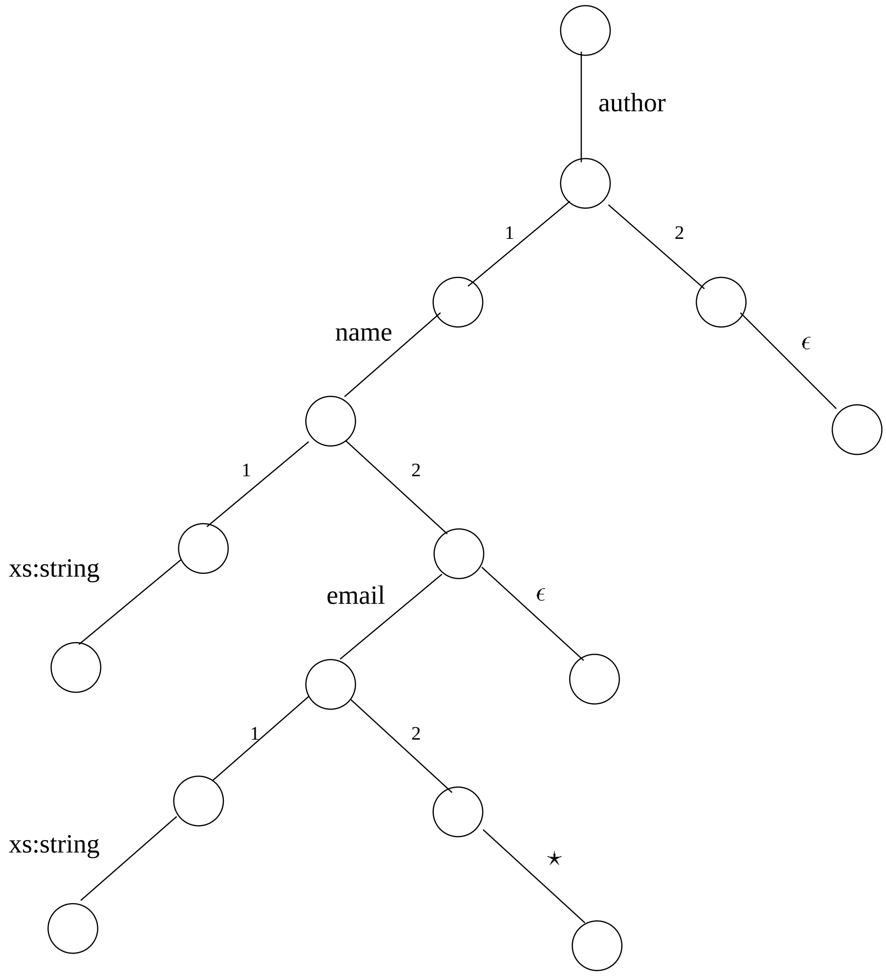


Figure 3.4: Trie Representation of Type t_1

3. Add the individual tries to the Tindex T one at a time. Let tr_i denote the trie of type t_i and T_{i-1} denote the trie that resulted from merging tries tr_1,...,tr_{i-1}. In the

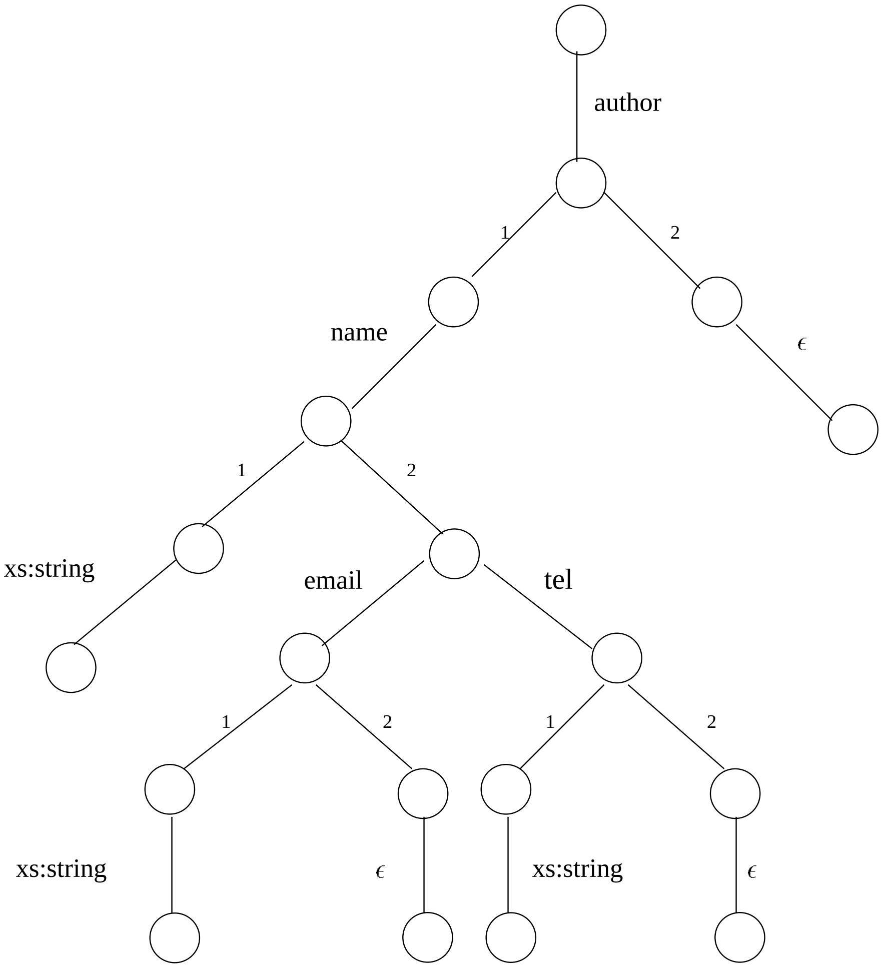


Figure 3.5: Trie Representation of Type t_2

ith step of the iteration, tr_i and T_{i-1} are merged to form T_i. Merging tr_i and T_{i-1} involves two steps:

(a) First obtain all the root-to-leaf paths in tr_i

(b) Second, insert each of these paths into T_{i-1} using the same technique used to build the Aho-Corasick automaton [5]. T_i is obtained after inserting the last path. Tindex T is obtained after inserting all types in the environment. Each node v in the trie T is associated with a set S_v. S_v is defined as follows:

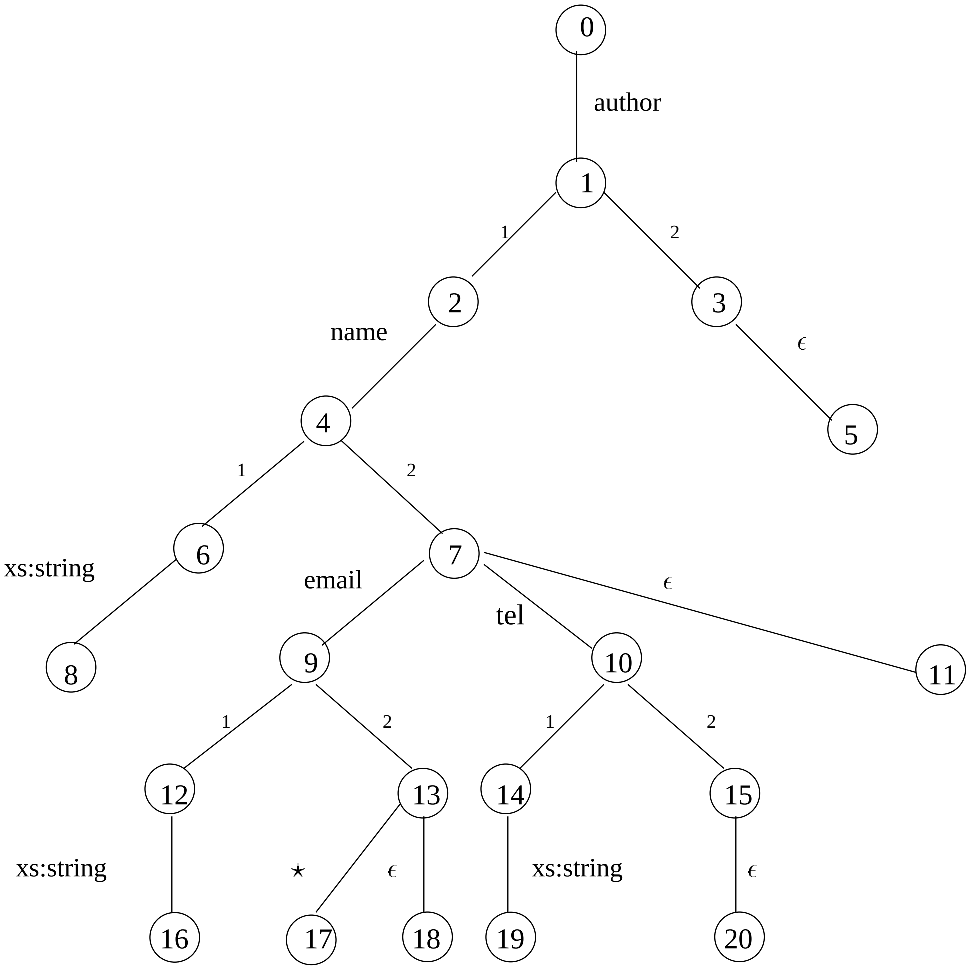
Definition 10 *The path to a node v, denoted by path(v), in a tree or a trie is the string obtained by concatenating the edge labels in the path from root to v.*

Definition 11 *S_v is the set of types t_i, such that the path from the root of T to v is a prefix of some root-to-leaf path in tr_i.*

Figure 3.6 shows Tindex T for the environment that consists of types t_1 and t_2.

3.2.3 Retrieval from Index

Figure 3.7 shows the algorithm *Retrieve* that returns all candidate supertypes using the index T. The algorithm is recursive. It takes three arguments: the current node u in the query type, a node v in the index T, and a set S of types that are still considered as potential supertypes. The query type is not transformed into a trie, rather it is preprocessed into a type tree (i.e., only steps 1 and 2.a from the trie construction algorithm are performed on it, which means that query type u has labeled nodes as opposed to index node v which has labeled edges). In the first call to *Retrieve*, u is the root of the query type tree, v is the root of T, and S is the set of all types that were used to form T. The algorithm first checks the label of u. If u is labeled with a variable or null (line 1), the call returns the set of types found so far. This is because a variable can be matched



$$S_1 = S_2 = S_3 = S_4 = S_5 = S_6 = S_7 = S_8 = S_9 = S_{12} = S_{13} = S_{16} = \{t_1, t_2\}$$
$$S_{10} = S_{14} = S_{15} = S_{18} = S_{19} = S_{20} = \{t_2\}$$
$$S_{11} = S_{17} = \{t_1\}$$

Figure 3.6: Trie Representation of Tindex T

to anything and a null is a subtype of all types. If u is labeled with a union (line 4), we first find the sets S_i of candidate supertypes for each child of u. These are found by calling the *Retrieve* function on each node that is a child of u (line 7). Then, we take the intersection of the sets S_i. The set resulting from the intersection represents the candidate supertypes of the union. The reason behind this is the subtyping rule SPLIT that states that:

$$T_1|T_2 <: R \iff T_1 <: R \quad and \quad T_2 <: R$$

If there is an edge leaving v towards w' having label $\star$ (line 12), then w' is a leaf, so $S \cap S_{w'}$ contains potential supertypes, and is returned by *Retrieve*. If u is labeled with a function name and there is no outgoing edge from node v labeled with the same function name, then all the types that are in S_v need to be eliminated from the set S of candidate supertypes (line 23). Remember that a type t belongs to S_v if the path from the root of T to v is a prefix of a root-to-leaf path in t. Therefore, if v does not have an outgoing edge having the same label as u, the types in S_v cannot be matched further, and can be eliminated. Therefore, the function *Retrieve* returns the intersection of the complement of S_v with S: $S \cap \overline{S_v}$. Now, suppose the node v has an outgoing edge to node w that is labeled with the same function name as the current node u in the query type (line 15). We can then move to w because the prefixes of some of the rules in S have been augmented. The set of all candidates that share the root-to-leaf path to w is S_w. However, there are some types in S_w that are not in S because they have been eliminated while processing the subtrees of w, thus the need to consider the set $S_1 = S \cap S_w$. If w is not a leaf, the retrieval of supertypes is not complete yet, and we still need to perform recursive calls on each child of w (lines 20 and 21). Note that the children of w are labeled with integers, so w must have two children. Let w_1, w_2 be the children of w and u_1,u_2 the children of u. We perform two recursive calls with u_i, v_i and S_1 as inputs to the ith call. After returning from the ith call, S_1 becomes $S_1 \cap S_i$. Then, after we return from the second recursive call, we return $S_1 \cup M_1$ (line 25). The algorithm terminates on the return of the initial recursive call at the root of the query type. It returns the set of candidate supertypes. Consider the type t_3 defined below as an example query type:

```
DEFINE TYPE t3 {
   ELEMENT author {
      ELEMENT name {xs:string},
      ELEMENT email {xs:string}
   }
}
```

If u is the root of t_3, v is the root of Tindex T in Figure 3.6 and S is the set of all types in the environment (i.e., $S = \{t_1, t_2\}$), then the call $Retrieve(u, v, S)$ returns $S_8 \cap S_{16} \cap (S_{17} \cup S_{18}) = \{t_1, t_2\}$, and indeed both t_1 and t_2 are supertypes of t_3.

3.2.4 Complexity

The complexity of the Tindex indexing mechanism is the same as the complexity of the trie version of the automata-driven indexing given by Ramesh et al. [46]. It consists of the complexity of constructing the index itself, plus the cost of retrieving the results. Inserting each type in the index takes time quadratic in the size of the type [32], and the time complexity of *Retrieve* is linear in the number of nodes in the goal, assuming set union and intersection take $O(1)$ time. In our implementation, sets of types are represented as sets of integers, where each integer is a unique id representing the type. Let n be the number of types in the index, $|t_i|$ the size of the type t_i, and $|t|$ the size of the query type. (The size of a type can be approximated by the number of nodes in its tree representation.) Therefore,

$$\text{Index Construction} : O(\sum_{i=0}^{n} |t_i^2|)$$

Index Retrieval : $O(|t|)$

3.2.5 Proofs

In this section, we extend the proofs of Ramesh et al. [46] to prove the correctness of the supertype selection algorithm *Retrieve*. First, we define some terminology:

Definition 12 *A node in t is said to be active if its outgoing edges are labeled with a function name.*

Definition 13 *$node_T(v, t_i)$ denotes the node in the tree representation of type t_i that produced node v in Tindex T during the construction of the index. (Note that v can correspond to nodes in several types.)*

Definition 14 *A node v is said to be a supertype of node u if the subtree rooted at v is a supertype of the subtree rooted at u.*

Proposition 1 *Let w_1, w_2,...,w_n be the children of an active node v. Then $r \in S_{w_i}$ if and only if $r \in S_v$ and $node_T(v, r)$ has the same label as the outgoing edge from v to w_i [46].*

Tindex T has the following properties:

Property 1 *α is a path in T if and only if it is a path in at least one of the type tries that was used to build the index, and if β is a root-to-leaf path in T, then it cannot be a path of a nonleaf node in any of these tries.*

Property 2 *Outgoing edges from any node are either all labeled with function names or all labeled with integers.*

Lemma 1 *Suppose S_{out} is the set of types returned from $Retrieve(u, v, S_{in})$. If $r \in S_{in}$ and $node_T(v, r)$ is a supertype of u, then $r \in S_{out}$.*

Proof. We will prove Lemma 1 by induction on the height of v. Note that *Retrieve* is called only from active nodes.

Base case: Height of v is 1. We will consider six cases depending on the label of u. So suppose $r \in S_{in}$.

Case 1: u **is labeled with a variable**

If u is labeled with a variable, then any type in S_{in} is a potential supertype of u. Note that by line 2, $S_{out} = S_{in}$. Therefore, $r \in S_{out}$.

Case 2: u **is labeled with** $\emptyset$

If u is labeled with $\emptyset$, then by the subtyping rule EMPTY, any type in S_{in} is a supertype of u. Note that by line 2, $S_{out} = S_{in}$. Therefore, $r \in S_{out}$.

Case 3: u **is labeled with** ϵ

If u is labeled with ϵ, then by the subtyping rule LEAF, $node_T(v, r)$ is a supertype of u if and only if $node_T(v, r)$ can be written as $node_T(v, r) \equiv x|\epsilon$. Consider the sets R, R_0, R_1:

$R = \{r \in S_{in} | node_T(v, r) \text{ is labeled with a } \star\}$

$R_0 = \{r \in S_{in} | node_T(v, r) \text{ is labeled with } \epsilon\}$

$R_1 = \{r \in S_{in} | node_T(v, r) \text{ is labeled with a union and } node_T(v, r) \text{ has an outgoing edge labeled with } \epsilon\}$

We clearly see that $node_T(v, r)$ is a potential supertype if and only if $r \in R \cup R_0 \cup R_1$. Therefore we need to prove that if $r \in R \cup R_0 \cup R_1$ then $r \in S_{out}$. Now, suppose that $r \in R \cup R_0 \cup R_1$. If $r \in R$, then there is an edge from v to w' with label $\star$, so by Proposition 1 $r \in S_{w'}$, and $r \in S_{in}$, so:

$$r \in R \Rightarrow r \in S_{in} \cap S_{w'} \tag{3.1}$$

On the other hand if $r \in R_0$ or $r \in R_1$, then by construction v must have an outgoing edge towards w labeled with ϵ, so $r \in S_w$ by Proposition 1 and $r \in S_{in}$. So:

$$r \in R_0 \cup R_1 \Rightarrow r \in S_{in} \cap S_w \tag{3.2}$$

From line 25, $S_{out} = S_1 \cup M_1$, and from lines 13 and 16, $M_1 = S_{in} \cap S_{w'}$ and $S_1 = S_{in} \cap S_w$, so $S_{out} = (S_{in} \cap S_{w'}) \cup (S_{in} \cap S_w)$. Therefore, using (3.1), and (3.2), we have:

$$r \in R \cup R_0 \cup R_1 \Rightarrow r \in S_{in} \cap (S_w \cup S_{w'}) \Rightarrow r \in S_{out}$$

Case 4: **u is labeled with β.** If u is a builtin type (or equivalently u is labeled with β), $node_T(v,r)$ is a supertype of u if and only if $node_T(v,r)$ is labeled by the same builtin name l. Therefore, $node_T(v,r)$ is a potential supertype of u if $node_T(v,r)$ is labeled with l or with a $\star$. Suppose that $node_T(v,r)$ is labeled with $\star$, then, using the same argument as in Case 3, we can show that $r \in S_{out}$. Now suppose that $node_T(v,r)$ is labeled with l. We will prove that $r \in S_{out}$. By construction, v has an outgoing edge to w labeled with l, and since the function with name l has arity 0, w is a leaf. Therefore, by Proposition 1, $r \in S_w$ and $r \in S_{in}$, so $r \in S_{in} \cap S_w$. From lines 25 and 16, we can see that *Retrieve* returns $S_{in} \cap S_w \cup S_{w'}$. Therefore, $r \in S_{out}$.

Case 5: **u is labeled with a function name of arity 2**

If u is labeled with a function name with arity 2, then by the subtyping rule REC, if $node_T(v,r)$ is a supertype of u, $node_T(v,r)$ should be labeled with the same function name l of arity 2, or with a union, whose children are functions with arity 2. In either case, the children of $node_T(v,r)$ are not leaves, so by Property 1, the children of v are not leaves either. Therefore, v will have a height greater than 1. However, by the induction hypothesis, v has height 1, therefore the only potential supertypes of u are variables. So, $node_T(v,r)$ is a potential supertype if and only if $node_T(v,r)$ is labeled with a $\star$. Therefore $r \in S_{w'}$ by Proposition 1, and $r \in S_{in}$, so $r \in S_{in} \cap S_{w'}$. To prove the lemma in this case, we should check that S_{out} returns $S_{in} \cap S_{w'}$. By lines 13 and 25, $S_{out} = (S_{in} \cap S_w) \cup (S_{in} \cap S_{w'})$, so $r \in S_{out}$.

Case 6: **u is labeled with a union**

If u is labeled by a union, then the subtree rooted at u can be described as $q_1|...|q_n$, where

$q_1, ...q_n$ are types. So, by the subtyping rule SPLIT:

$$q_1|...|q_n <: node_T(v,r) \iff q_i <: node_T(v,r) \quad \forall i \in \{1,...,n\}$$

Therefore, finding supertypes of u is equivalent to finding the types t that are supertypes of all q_i.

Consider the following sets R_i:

$R_0 = \{r \in S_{in}\}$

$R_i = \{r \in R_{i-1} | node_T(v,r)$ is a supertype of $q_i\}$

Note that:

$$u <: node_T(v,r) \iff r \in R_n$$

Therefore, to prove that the *Retrieve* algorithm returns all potential supertypes of u in this case, we need to show that:

$$r \in R_n \Rightarrow r \in S_{out}$$

Let S_{1_i} be the values of S_1 at the end of the ith iteration of the "for" loop in lines 6-8. From line 7, we have $S_{1_n} = S_{1_{n-1}} \cap Retrieve(q_n, v, S_{1_{n-1}})$ and $S_{out} = S_{1_n}$. We show by induction on k that:

$$r \in R_n \Rightarrow r \in S_{1_n}$$

The hypothesis clearly holds for $k = 0$ since $r \in R_0 \Rightarrow r \in S_{in}$ and $S_{1_0} = S_{in}$. Suppose it holds for $k-1$, where $1 \leq k \leq n-1$, We prove that:

$$node_T(v,r) \in R_k \Rightarrow r \in S_{1_k}$$

Suppose $r \in R_k$, so $r \in R_{k-1}$ and $node_T(v,r)$ is a supertype of q_k.

By definition, if u is labeled with a union, $\forall i \in \{1,...n\}$, q_i is labeled with either a variable, $\emptyset$, ϵ, β, or a function name of arity 2. According to cases 1 to 5:

$\forall i \in \{1,...n\}, node_T(v,r)$ is a supertype of q_i implies that r is returned by the Retrieve algorithm.

In particular, $node_T(v, r)$ is a supertype of q_k means that r is returned by $Retrieve(q_k, v, S_{1_{k-1}})$. On the other hand, by the induction hypothesis on k, $r \in S_{1_{k-1}}$, and we have:

$$S_{1_k} = S_{1_{k-1}} \cap Retrieve(q_k, v, S_{1_{k-1}})$$

Therefore, $r \in S_{1_k}$. So we can conclude that

$$r \in R_k \Rightarrow r \in S_{1_k}, \text{ for } 0 \leq k \leq n$$

In particular,

$$r \in R_n \Rightarrow r \in S_{1_n}$$

and $S_{1_n} = S_{out}$, so

$$r \in R_n \Rightarrow r \in S_{out}$$

This proves the lemma for this case.

Induction step: We assume that the lemma holds for heights of v that are less than k, and prove that the lemma will hold for height of v equal to k. We will consider the same cases as in the base condition, and we will only treat case 5 since the proof for cases 1, 2, 3, 4, and 6 is the same as in the base condition.

Case 5: u is labeled with a function name of arity 2

Suppose $r \in S_{in}$. If u is labeled by a function name l, let X and Y represent the subtrees rooted at the children of u. By the subtyping rule REC, if $node_T(v, r)$ is a supertype of u, the subtree rooted at $node_T(v, r)$ should be of the form:

$$l(X'_1, Y'_1)|...|l(X'_n, Y'_n)$$

Since we consider only first-level unrolling in the construction of Tindex, we can rewrite $node_T(v, r)$ as:

$$l(X'_1|...|X'_n, Y'_1|...|Y'_n) \tag{3.3}$$

Therefore, $node_T(v,r)$ is a potential supertype of u if and only if $node_T(v,r)$ is a variable (condition 1) or the subtree rooted at $node_T(v,r)$ is of the form (3.3) and $X <: X'_1|...|X'_n$ and $Y <: Y'_1|...|Y'_n$ (condition 2). We will prove that if either condition (1) or condition (2) is satisfied, r will be returned by the *Retrieve* algorithm.

If $node_T(v,r)$ is a variable, then there is an edge from v to w' with label $\star$, and w' is a leaf, so $r \in S_{in} \cap S_{w'}$ according to Proposition 1. We should prove that $S_{in} \cap S_{w'} \subset S_{out}$. From lines 25 and 13, we can see that $S_{out} = S_1 \cup M_1$ and $M_1 = S_{in} \cap S_{w'}$ therefore $S_{in} \cap S_{w'} \subset S_{out}$, so r is returned by the *Retrieve* algorithm.

Now suppose that condition 2 is satisfied. The subtree rooted at $node_T(v,r)$ is of the form (3.3). Therefore, by construction, v has an outgoing edge towards w labeled with l. Let w_1 and w_2 be the children of w. w_1 will have at least n outgoing edges labeled with X'_1 to X'_n, and w_2 will have at least n outgoing edges labeled with Y'_1 to Y'_n. Therefore, the subtree rooted at $node_T(w_1,r)$ is of the form $X'_1|...|X'_n$ and the subtree rooted at $node_T(w_2,r)$ is of the form $Y'_1|...|Y'_n$. From lines 25, 20 and 21, $S_{out} = S_1 \cup M_1$ and $S_1 = S_{in} \cap Retrieve(X, w_1, S_{in}) \cap Retrieve(Y, w_2, S_{in})$. Note that v has height k, so w_1 and w_2 have height less that k, so according to the induction hypothesis on v,

$$r \in S_{in}, X <: node_T(w_1,r) \Rightarrow r \in S_{out_1}$$

and

$$r \in S_{in}, Y <: node_T(w_2,r) \Rightarrow r \in S_{out_2}$$

where S_{out_1} and S_{out_2} are the results of $Retrieve(X, w_1, S_{in})$ and $Retrieve(Y, w_2, S_{in})$, respectively. Since condition 2 is satisfied, $r \in S_{out_1} \cap S_{out_2}$, thus $r \in S_{out}$.

Theorem 1 *The Retrieve algorithm selects all supertypes of a query type.*

Proof. *Retrieve* is called initially with S being the set containing all the types in the environment, v the root of Tindex, and u the root of the query type. Therefore, according to Lemma 1, *Retrieve* will return all supertypes of u.

3.2.6 DTD Indexing

In this section, we will prove that the retrieve algorithm in Figure 3.7 also retrieves all candidate supertypes for DTDs. First, we define subtyping for DTDs:

Definition 15 (DTD Subtyping) *Let dtd_1 and dtd_2 be two DTDs. DTD dtd_1 is said to be a supertype of DTD dtd_2 if and only if the set of all documents conforming to dtd_2 is included in the set of all documents conforming to dtd_1. It is written as $dtd_1 <: dtd_2$.*

Lemma 2 *Let dtd_1 and dtd_2 be two DTDs containing m and n element declarations respectively. Let $Type_{1_i}$ be the type denoted by the ith element declaration in dtd_1 for $1 \leq i \leq m$. Let $Type_{2_j}$ be the type denoted by the jth element declaration in dtd_2 for $1 \leq j \leq n$.*

$$dtd_1 <: dtd_2 \quad \Longleftrightarrow \quad \forall \quad 1 \leq i \leq m, \quad \exists \quad 1 \leq j \leq n \quad | \quad Type_{1_i} <: Type_{2_j}$$

Proof.

sufficiency

Suppose that $\forall \quad 1 \leq i \leq m, \quad \exists \quad 1 \leq j \leq n \quad | \quad Type_{1_i} <: Type_{2_j}$. Then it is clear that all documents that conform to dtd_1 also conform to dtd_2. From Definition 15, dtd_1 is a subtype of dtd_2.

necessity

Suppose that $dtd_1 <: dtd2$, we prove that all types of dtd_1 have supertypes in dtd_2 by contradiction.

Suppose that $\exists k \quad 1 \leq k \leq m \quad | \quad \forall \quad 1 \leq j \leq n$, $Type_{1_k}$ is not a subtype of $Type_{2_j}$.

Let d be an XML document that conforms to dtd_1. We choose d such that it contains XML data of type $Type_{1_k}$. Since there is no j, $1 \leq j \leq n \mid Type_{1_k} <: Type_{2_j}$, there is no element declaration in dtd_2 that validates document d. Therefore, document d does not conform to $dtd2$. However, we know that d conforms to dtd_1 and $dtd_1 <: dtd_2$, so this contradicts Definition 15, which states that all documents conforming to dtd_1 should conform to dtd_2.

Therefore,

$$dtd_1 <: dtd_2 \iff \forall \;\; 1 \leq i \leq m, \;\; \exists \;\; 1 \leq j \leq n \;\; | \;\; Type_{1_i} <: Type_{2_j}$$

Theorem 2 *The Retrieve algorithm selects all supertypes of a DTD*

Proof.

Let d be a DTD containing m element declarations. Let $Type_i$ be the type denoted by the ith element declaration in d for $1 \leq i \leq m$. By Theorem 1, the *Retrieve* algorithm will return all supertypes of $Type_i$ for $1 \leq i \leq m$. Moreover, by Lemma 2, selecting supertypes of the individual element declarations within a DTD is equivalent to selecting the supertypes of a DTD. Therefore, *Retrieve* selects all supertypes of DTD d.

Retrieve(u, v, S)
/*u = query type node, v = index node, S = set of selected types*/
/*Returns updated set of selected types*/
BEGIN
IF u is null OR u is labeled by a variable THEN
    RETURN(S)
FI
IF u is labeled by a union operator | THEN
    $S_1 = S$
    FOR each child node q_i of u DO
        $S_1 = S_1 \cap Retrieve(q_i, v, S_1)$
    END
    RETURN(S_1)
FI
$M_1 = \emptyset$
IF there is an edge e in T leaving v towards w' having label $\star$ THEN
    $M_1 = S \cap S_{w'}$
FI
IF there is an edge e in T leaving v towards w having label of u THEN
    $S_1 = S \cap S_w$
    IF w is not a leaf THEN
        LET w_1, w_2 be the children of w
        LET u_1, u_2 be the children of u
        $S_1 = S_1 \cap Retrieve(u_1, w_1, S_1)$
        $S_1 = S_1 \cap Retrieve(u_2, w_2, S_1)$
    FI
ELSE $S_1 = S \cap \overline{S_v}$
FI
RETURN($S_1 \cup M_1$)
END

Figure 3.7: Selection of Candidate Supertypes Using Tindex T

Chapter 4

Experiments

In this chapter, we first describe the implementation of both the sequential subtyping algorithm and our indexing algorithm Tindex. Then, we discuss the experimental methodology of the experiments that we performed to study the performance of Tindex, and we analyze the experimental results.

4.1 Implementation

We implemented two algorithms: a sequential subtyping algorithm, and the Tindex algorithm. The sequential algorithm that we implemented is a JAVA version of the XDuce system, which was originally implemented by Hosoya et. al [35] in the ML programming language. The indexing algorithm of Tindex consists of two main modules: an index construction algorithm, and a retrieval algorithm. Both were described in detail in the previous chapter. All algorithms were developed in JAVA, and all experiments were run under Solaris 8.0 on a Sun SPARC station with 32MB of Memory.

4.2 Methodology

In order to evaluate the performance of our indexing mechanism, we performed four different experiments. We used both synthetic types and real types in our experiments. All of the types used in the experiments were extracted from DTDs. The real types were extracted from a real DTD, and the synthetic types were extracted from synthetic DTDs. The real DTD that we used in our study is the DTD IndexTermsPage that we showed in Figure 1.2 in Chapter 1. From this real DTD, we generate synthetic DTDs by performing the following changes to the element declarations in the real DTD:

1. Removing subelements from sequence. For example, the following element: <!ELEMENT author (name,email,tel)> is changed to <!ELEMENT author (name,email)>.

2. Adding dummy subelements.

3. Adding/Removing "*", "+", "?". For example, the following element: <!ELEMENT author (name,email,tel)> is changed to <!ELEMENT author (name*,email,tel)> .

4. Including unions with dummy variables. For example, the following element: <!ELEMENT author (name,email,tel)> is changed to <!ELEMENT author (name | name | email | tel, tel)>.

Once the synthetic DTDs are generated, we extract their element declarations and use them to populate the index. Then we use two types from the real DTD as input to the indexing and sequential algorithms. One of the types has a low complexity, and the other has a high complexity as described in Table 4.1.

The four experiments that we performed are summarized in Table 4.2.

Data Set	Description
S	The input type is element *author*. The input has low complexity: small size, little nesting of subelements, and no presence of Kleene operators "*", "?" or "+".
L	The input type is element *IndexTermsPage*. The input has high complexity: large size, deep nesting of subelements, and contains Kleene operators "*" or "+" and/or sequences of these Kleene operators.

Table 4.1: Data Sets

Data Set	Description
Experiment 1	Compare the supertype selection time using Tindex for both data sets L and S to the sequential algorithm for a growing number of types in the index.
Experiment 2	Measure the contributions of both the retrieval time and the elimination of false positives to the total subtyping time of Tindex for a growing number of types in the index.
Experiment 3	Measure the construction time for Tindex for a growing number of types in the index.
Experiment 4	Measure the size of Tindex for a growing number of types in the index.

Table 4.2: Experiments

4.3 Results

This section shows the results of the experiments that we performed. In our experiments, we tried to study the limitations of our indexing algorithm by studying its behavior in the presence of a large database of types. We also populate the index with types very similar to the inputs, which means a large number of supertypes (including some false positives) is returned. We will see that, even in this extreme situation, our indexing mechanism always outperforms the sequential algorithm.

4.3.1 Experiment 1

Figure 4.1 shows the supertype selection time using Tindex for data set S compared to the supertype selection using the sequential algorithm. We can see, that as the size of the index grows, supertype selection using the index is much faster than by using the sequential algorithm. For instance, when our database contains almost 10,000 types, the sequential algorithm takes 16.5 seconds, whereas the selection of supertypes using the index takes 0.33 seconds.

We ran the same experiment on a more complex input, which is element $IndexTermsPage$. Figure 4.2 shows the results of this experiment. We can see that, for this complex type too, our indexing mechanism outperforms the sequential algorithm. For instance, when the database contains almost 10,000 types, the sequential algorithm takes 62.1 seconds, whereas the selection of supertypes using the index takes 11.2 seconds.

4.3.2 Experiment 2

In this experiment, we want to study in more detail the distribution of the subtyping time of our indexing mechanism Tindex. Recall that our indexing mechanism might return false positives in the presence of Kleene operators "*" and "+". Therefore, to guarantee that only actual supertypes in the database are selected, our algorithm runs

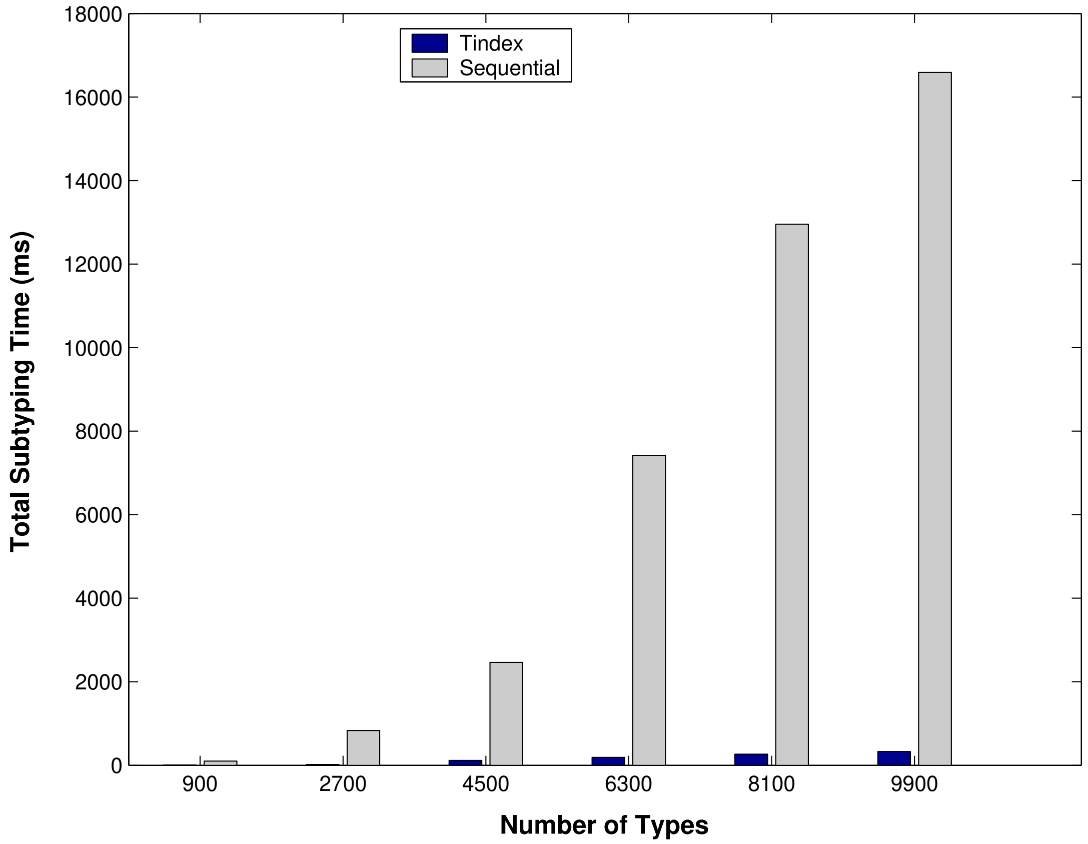


Figure 4.1: Supertype Selection Time for Type *author* Using Tindex vs. the Sequential Algorithm

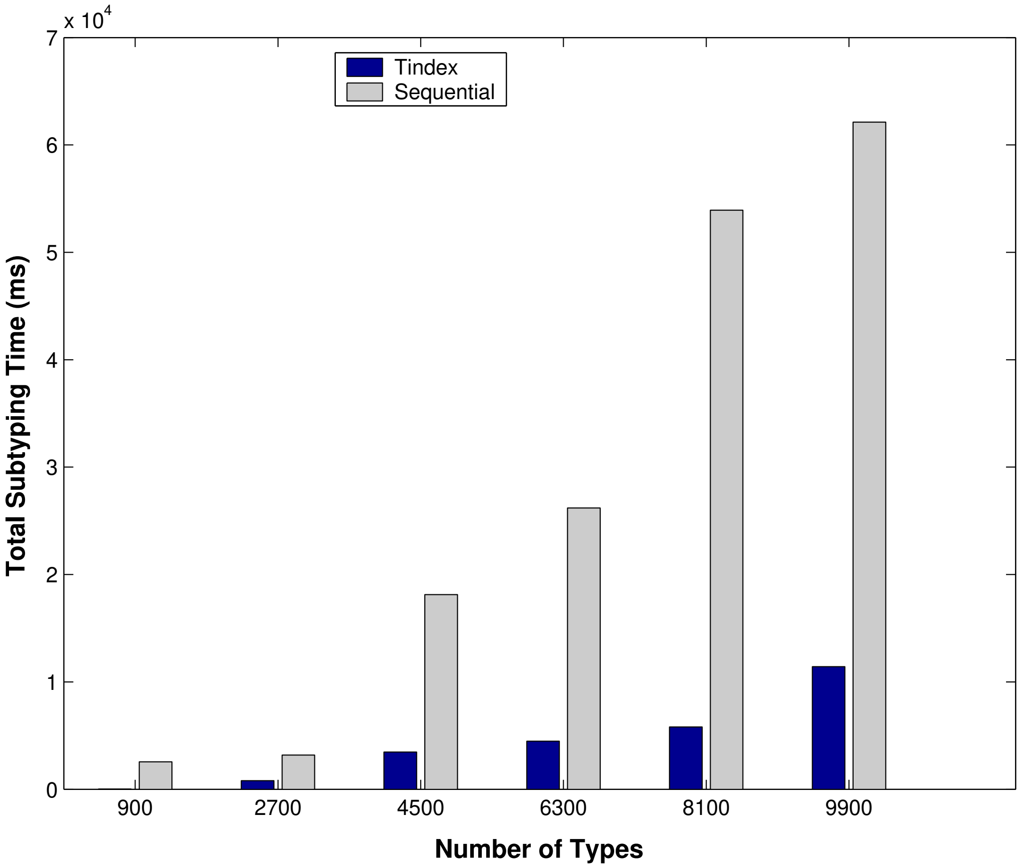


Figure 4.2: Supertype Selection Time for Type $IndexTermsPage$ Using Tindex vs. the Sequential Algorithm

in two passes: first, we select candidate supertypes from the index (Pass 1), then we run the sequential algorithm on the filtered set (Pass 2).

Figure 4.3 shows the contribution of each of the retrieval time from the index and the elimination of false positives to the total subtyping time for a growing number of types in the index for the input *author*. We can see that it takes longer to eliminate false positives than to retrieve types from the index. The reason is that element *author* had many supertypes in the database. For instance, when there are almost 10,000 types in the database, element *author* has 550 supertypes. Also note that because this element has low complexity, all these 550 supertypes were actual supertypes and no false positives were returned.

Figure 4.4 shows the contribution of each of the retrieval time from the index and the elimination of false positives to the total subtyping time for a growing number of types in the index for the input *IndexTermsPage*. In this experiment too, we can see that it takes longer to eliminate false positives than to retrieve candidate supertypes from the index. Also note that, in this experiment, false positives were returned because the element *IndexTermsPage* has a Kleene star and a sequence of Kleene stars. The maximum number of false positives returned in all experiments was 80 false positives. Recall that in our experiments, we want to push our indexing machanism to the limits, so we used synthetic data, and we made sure to include false positives in our data sets. However, in practice, using Tindex for real data sets will probably not produce as many false positives. The assumption is that there is only a small probablity that an input type has many types that share its structure in the database.

Now we will analyze why the indexing mechanism behaves differently for data sets S and L. First, it takes longer to retrieve supertypes of data set S than data set L, as was expected from our complexity analysis, since the size of data set S is smaller than the size of data set L. Second, even if the set of supertypes of data set S is larger than the set of supertypes plus false positives of data set L, the sequential algorithm takes longer in

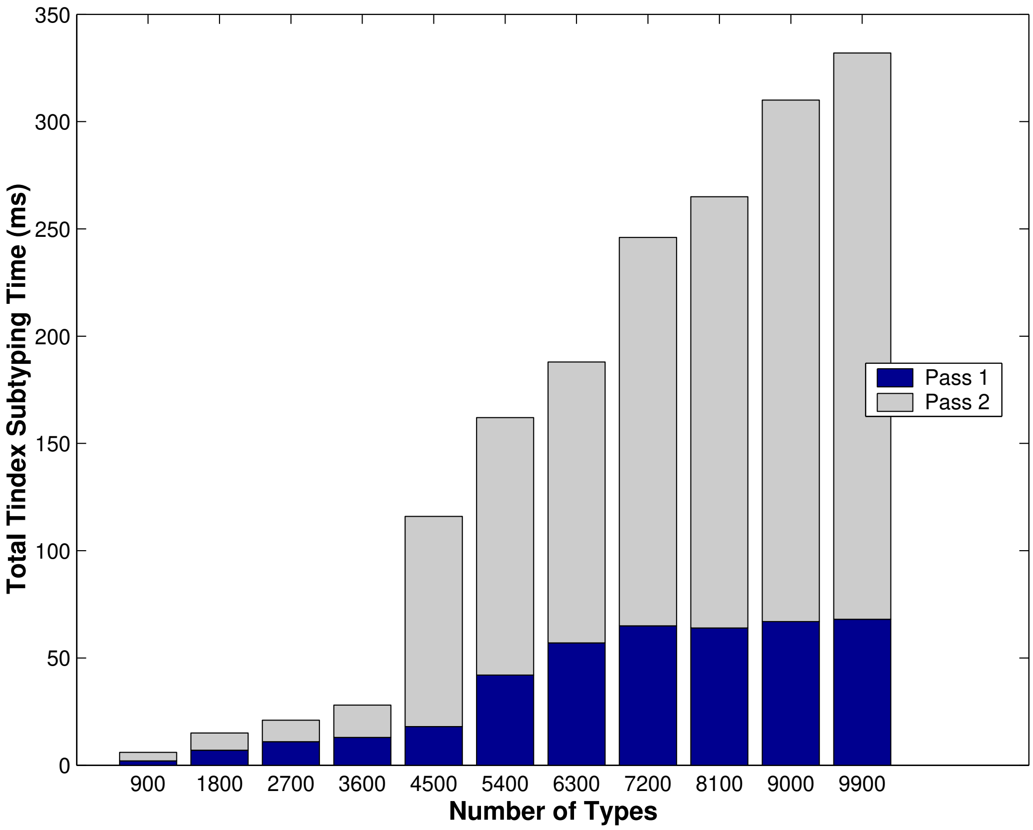


Figure 4.3: The Contribution of the Retrieval Time and the Elimination of False Positives to the total Supertype Selection Time for Type *author*

the case of data set L because input element $IndexTermsPage$ is complex and requires the subtyping rule REC to be executed. (Rule REC is depicted in 2.4 and was explained in detail in Chapter 2.)

4.3.3 Experiment 3

Figure 4.5 illustrates the total time to construct the index consisting of the preprocessing of all the types into tries and the insertion of the all the types to the index. We can see that the more types in the index, the longer it takes to construct the index.

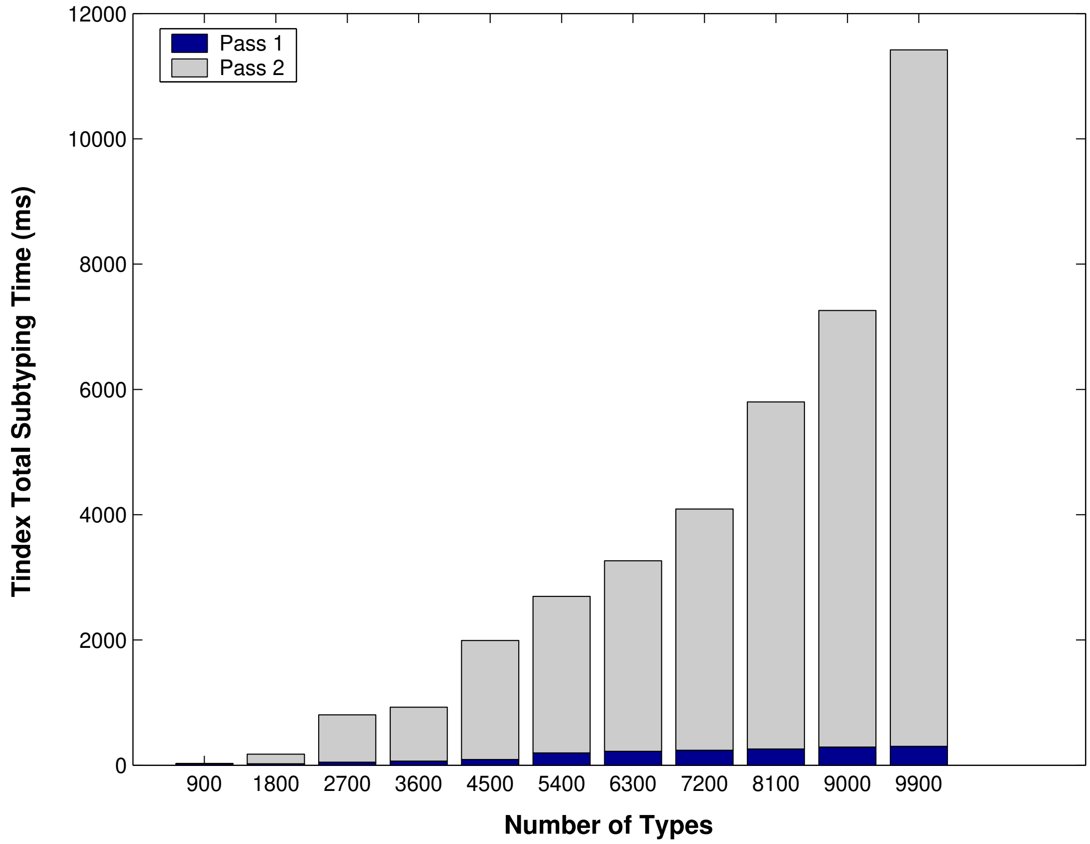


Figure 4.4: The Contribution of the Retrieval Time and the Elimination of False Positives to the total Supertype Selection Time for Type *IndexTermsPage*

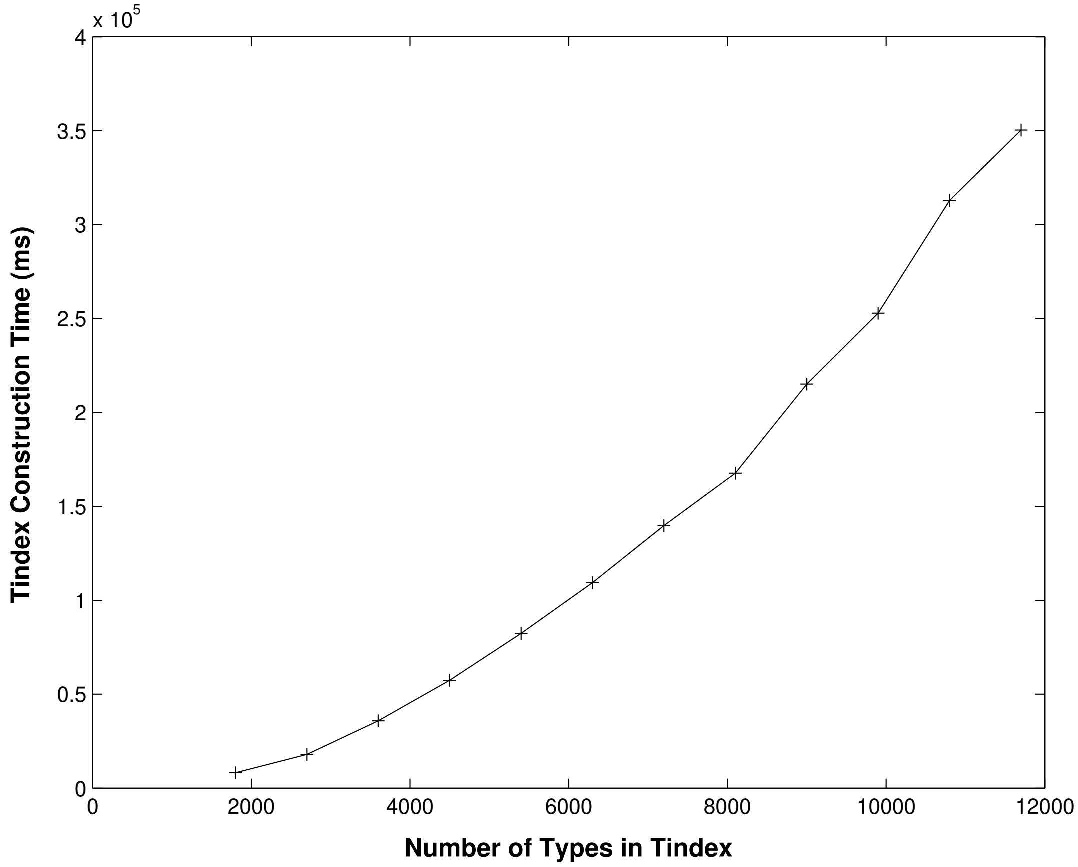


Figure 4.5: Tindex Construction Time vs. Number of Types in Tindex

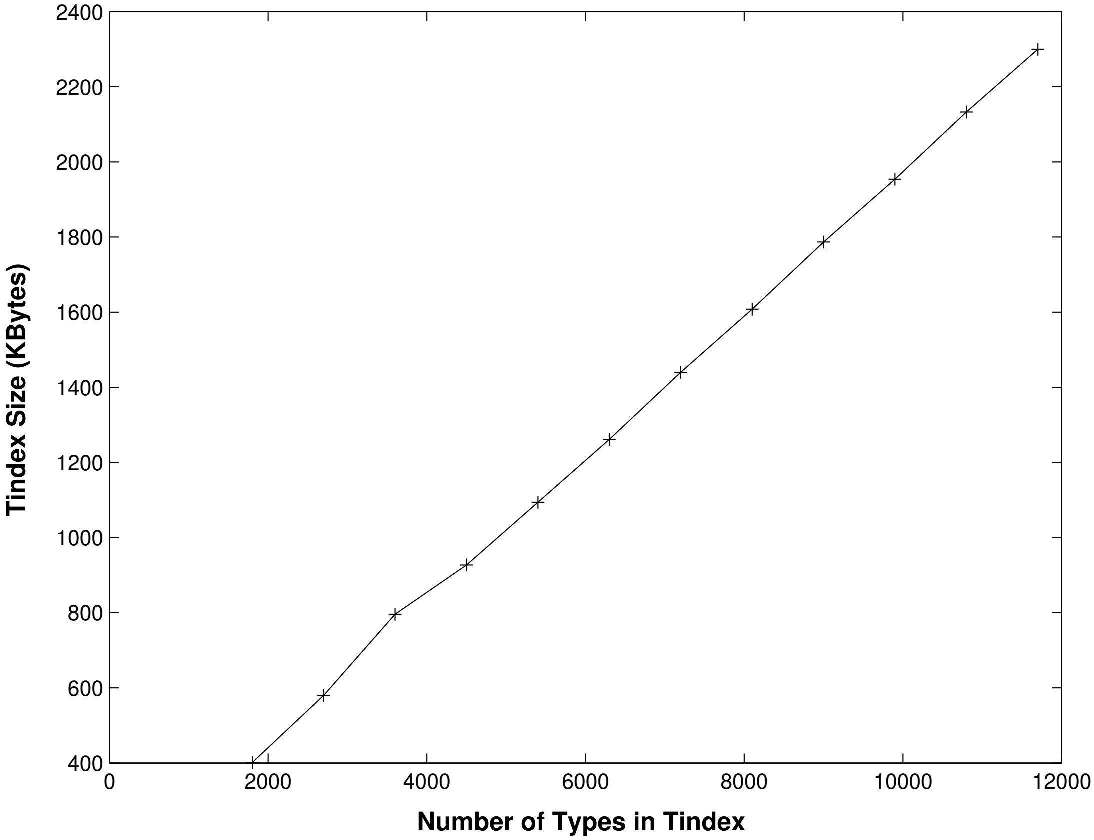


Figure 4.6: Tindex Size vs. Number of Types in Tindex

4.3.4 Experiment 4

Figure 4.6 shows the size of the index in Kbytes with a growing number of types in the database. The size of the index grows almost linearly with the number of types. Each additional type adds about 200 bytes to the size of the index.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis we developed an indexing mechanism called Tindex that efficiently retrieves supertypes of an XML type. We presented an overview of subtyping proposals for XML. We adopted the subtyping rules developed in the XDuce system [35] because XDuce types map naturally to XQuery types. We surveyed literature related to the type indexing problem, and we extended the ALS prolog automata-driven indexing mechanism to index XML types [46]. We discussed the index construction and supertype selection algorithms, and we formally proved that our indexing mechanism returns all supertypes of an input type. We implemented a sequential subtyping algorithm based on the XDuce system, and we compared the performance of our indexing mechanism, Tindex, to the sequential algorithm. We presented detailed performance results that show that our indexing mechanism, Tindex, always outperforms the sequential algorithm.

5.2 Future Work

Although the Tindex indexing mechanism is useful in its current state, there are several extensions that could improve its applicability. We will describe three such extensions:

support for XML Schema, parameterized unrolling for recursive types, and alternate criteria for equivalence and subtyping.

Support for XML Schema

Currently, Tindex handles XQuery types (except the interleave operator and attributes) and DTDs. However, because XQuery was still work in progress while we were developing Tindex, the final W3C recommendation might contain features that are not currently handled by Tindex. In fact, the XQuery team states that the final proposal for XQuery types will be equivalent to XML Schema [25]. There are three particular features of interest that we would like Tindex to handle in the future: named typing, namespaces, and facet constraints. Recall that in the current XQuery proposal only the structural information about types is preserved, and type names are ignored, whereas XML Schema supports both structural and named typing. Also, remember that namespaces are a mechanism used to differentiate among types written by different authors. Therefore, we would like Tindex to take into consideration namespace information while selecting relevant supertypes of a given type. Finally, as we have seen in chapter 2, XML Schema supports constraint operators not only at the level of elements but also at the level of facets. For instance, the XML Schema "$xs : string$" atomic type can be constrained to represent strings of fixed length. We would like to extend Tindex to support this feature as well.

Parameterized Unrolling

Currently, Tindex handles Kleene operators "*" and "+" by representing them as recursive variables in XDuce internal representation, then performing a first level unrolling of the recursive variable. We would like to extend Tindex with a parameterized unrolling, so that the user can define how many levels of unrolling should be applied to recursive variables. Adding more levels of unrolling will improve the accuracy of Tindex by decreasing the number of false positives returned, but it would increase the size and construction time of the index. For instance, recall types t and t_1 from section 3.2.1. Type t_1 was selected as a supertype of type t by the Retrieve algorithm although it was not a real supertype. The first-level unrolling of types t and t_1 is as follows:

$$t = person(name(\beta, tel(\beta, *)|\epsilon), \epsilon)$$

$$t_1 = person(name(\beta, *)|tel(\beta, \epsilon), \epsilon)$$

We will see that a second level unrolling of type t_1 will eliminate this false positive. The second level unrolling representation of t_1 is as follows:

$$t_1 = person(name(\beta, \underbrace{name(\beta, *)|tel(\beta, \epsilon)})|tel(\beta, \epsilon), \epsilon)$$

The underbraced portion of the equation above replaces the star * in the first level unrolling of t_1. Using this second level unrolling, type t_1 will not be returned as a candidate supertype for t by the Retrieve algorithm because the following subtyping relation does not hold:

$$tel(\beta, *) \quad | \quad \epsilon \quad <: \quad name(\beta, *) \quad | \quad tel(\beta, \epsilon)$$

Because ϵ can only be a subtype of types of the form $\quad T \equiv T|\epsilon \quad$ according to the subtyping rule LEAF in Figure 2.4.

Therefore, we see that second level unrolling will decrease the number of false positives returned, which in turn will decrease the total subtyping time in Pass 2 of the supertype selection algorithm. It would be interesting to study the trade off between the level of unrolling of types, the number of false positives returned, the supertype retrieval time using Tindex, and the size of the index.

Alternate Definitions of Subtyping

Another possible extension to Tindex would be to allow different criteria for equivalence and subtyping between types. For instance, we could define new subtyping relations called *nested subtyping*, *approximate subtyping*, and *subset subtyping*.

Nested subtyping can be useful if we are interested mostly in the level of nesting of types as opposed to the names of the subelements that make up the type. It is defined as follows:

Definition 16 (Nested Subtyping) *A type t_1 is a subtype of type t_2 by nested subtyping if, after replacing all the type names in t_1 and t_2 by the same dummy type name, t_1 is a subtype of t_2 by inclusion subtyping where inclusion subtyping is as used throughout this thesis.*

Approximate subtyping could be useful if we are interested in checking subtyping between types only up to a level of nesting that could be determined by the user. It is defined as follows:

Definition 17 (Approximate Subtyping) *A type t_1 is a subtype of type t_2 by approximate subtyping if t_1 is a subtype of t_2 by inclusion subtyping up to a level of nesting.*

Subset subtyping could be useful in query optimization. If we know the domain of a query, we can select supertypes of the domain from the index. Since the query does not necessarily explicitly state the full path of the domain, selecting supertypes by subset subtyping could be helpful. It is defined as follows:

Definition 18 (Subset Subtyping) *A type t_1 is a subtype of type t_2 if t_1 is a subtype of a subtree of t_2 by inclusion subtyping.*

Bibliography

[1] ToX. *http://cs.toronto.edu/tox.*

[2] ISO 8879. International Standard ISO 8879: Standard Generalized Markup Language (SGML). *ISO*, 8879, 1986.

[3] A. Deutsch and V. Tannen. Optimization properties for classes of conjunctive regular path queries. In *Proceedings of The International Conference on Database Programming Languages (DBPL)*, 2001.

[4] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[5] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[6] D. Beech, A. Malhotra, and M. Rys. A formal data model and algebra for XML. *Communication to the W3C*, 1999.

[7] C. Beeri and Y. Tzaban. SAL: An algebra for semistructured data and XML. In *Int'l Workshop on the Web and Databases (WebDB)*, pages 37–42, 1999.

[8] A. Broder. Some applications of Rabin's fingerprinting method. In *R. Capocelli, A. De Santis, and U. Vaccaro, editors, Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[9] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Database Theory—ICDT'97, 6th International Conference*, volume 1186, pages 336–350. Springer, 1997.

[10] P. Buneman and B. Pierce. Union types for semistructured data. In *Proceedings of the International Database Programming Languages Workshop (DBPL)*, 1999.

[11] D. Calvanese, G. De Giacomo, and M. Lenzerini. Representing and reasoning on XML documents: A description logic approach. *Journal of Logic and Computation*, 9(3):295–318, 1999.

[12] L. Cardelli. Type Systems. In *The Computer Science and Engineering Handbook, CRC Press*. 1997.

[13] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: A graphical language for querying and restructuring XML documents. In *Sistemi Evoluti per Basi di Dati*, pages 151–165, 1999.

[14] T. Chen, I. V. Ramakrishnan, and R. Ramesh. Multistage indexing algorithms for speeding prolog execution. *Software - Practice and Experience*, 24(12):1097–1119, 1994.

[15] J. Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10(1):95–113, 1993.

[16] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *Proceedings of The ACM SIGMOD International Conference on Management of Data*, volume 27, pages 177–188, 1998.

[17] G. Conforti, O. Ferrara, and G. Ghelli. TQL algebra and its implementation (extended abstract). *To appear in Proc. of IFIP International Conference on Theoretical Computer Science (IFIP TCS), Montreal,Canada, August 2002.*

[18] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proceedings of The International Conference on Very Large Databases (VLDB)*, pages 341–350, 2001.

[19] B. Cooper and M. Shadmon. The Index Fabric. *Technical Report. Available at http://www.rightorder.com/technology/overview.pdf.*

[20] SIGMOD IndexTermsPage database. *http://www.acm.org/sigs/sigmod/record/xml/Record/IndexTermsPage/.*

[21] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of The ACM SIGMOD International Conference on Management of Data*, pages 431–442, 1999.

[22] A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. "XML-QL: A Query Language for XML". In *WWW The Query Language Workshop (QL)*, Cambridge, MA, 1998.

[23] W3C Working Draft. XQuery 1.0: An XML Query Language. *http://www.w3.org /TR/xquery/*, 2002.

[24] W3C Working Draft. XQuery 1.0 and XPath 2.0 Functions and Operators. *http://www.w3.org/TR/xquery-operators/*, 2002.

[25] W3C Working Draft. XQuery 1.0 Formal Semantics. *http://www.w3.org/TR/2002 /WD-query-semantics-20020326/*, 2002.

[26] M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of The International Conference on Data Engineering (ICDE)*, pages 14–23, 1998.

[27] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. In *IEEE Data Engineering Bulletin*, pages 27–34, 1999.

[28] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.

[29] R. Goldman and J. Widom. Approximate Dataguides. In *Proceedings of the Workshop on Semi-Structured Data and Non-Standard Data Formats*, pages 436–445, 1999.

[30] P. Graf. Path indexing for term retrieval. *Technical Report MPI-I-92-237. Max-Planck-Institut fur Informatik, Saarbrucken, Germany*, 1992.

[31] P. Graf. Term Indexing. In *Lecture Notes in Computer Science*, volume 1053. Springer-Verlag, 1996.

[32] C. M. Hoffman and M. J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, 1982.

[33] H. Hosoya and B. C. Pierce. XDuce: A Typed XML Processing Language. In *Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, 2000.

[34] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 67–80, 2001.

[35] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *In Proceedings of the International Conference on Functional Programming (ICFP)*, pages 11–22, 2000.

[36] DTD IndexTermsPage. *http://www.acm.org/sigs/sigmod/record/xml/Record/DTD/IndexTermsPage.dtd.*

[37] H. Jagadish, L. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *Proceedings of The International Conference on Database Programming Languages (DBPL)*, 2001.

[38] C. Kanne and G. Moerkotte. Efficient storage of XML data. In *Proceedings of The International Conference on Data Engineering (ICDE)*, 2000.

[39] G. M. Kuper and J. Siméon. Subsumption for XML Types. *In Proceedings of The International Conference on Database Theory (ICDT)*, 2001.

[40] W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[41] W. McCune and L. Wos. Otter-the CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997.

[42] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of The International Conference on Very Large Databases (VLDB)*, pages 315–326, 1999.

[43] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, 2000.

[44] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Symposium on Principles of Database Systems*, pages 35–46, 2000.

[45] I.V. Ramakrishnan, R.Sekar, and A. Voronkov. Term Indexing. *Handbook of Automated Reasoning*, pages 1853–1964, 2001.

[46] R. Ramesh, I. V. Ramakrishnan, and D.S. Warren. Automata-driven indexing of Prolog clauses. *Journal of Logic Programming*, 23(2):151–202, 1995.

[47] W3C Recommendation. XML Path Language (XPath) Version 1.0. *http://www.w3.org/TR/xpath*, 1999.

[48] W3C Recommendation. Extensible Markup Language (XML) 1.0 (Second Edition). *http://www.w3.org/TR/REC-xml*, 2000.

[49] W3C Recommendation. XML Schema Part 0: Primer. *http://www.w3.org/TR/xmlschema-0/*, 2001.

[50] W3C Recommendation. XML Schema Part 1: Structures. *http://www.w3.org/TR/xmlschema-1/*, 2001.

[51] W3C Recommendation. XML Schema Part 2: Datatypes. *http://www.w3.org/TR/xmlschema-2*, 2001.

[52] F. Rizzolo and A. Mendelzon. Indexing XML Data with ToXin. In *Proceedings of the Fourth International Workshop on the Web and Databases (WebDB'2001)*, 2001.

[53] J. Robie, J. Lapp, and D. Schach. XML query language (xql). In *QL'98-W3C Workshop on Query Languages*, Cambridge, MA, Dec. 1998.

[54] H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19(3):424–437, 1990.

[55] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, and D. De Witt. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of The International Conference on Very Large Databases (VLDB)*, pages 302–314, 1999.