A Type System for Querying XML Documents*

A. Albano, D. Colazzo, G. Ghelli, P. Manghi, C. Sartiani

Dipartimento di Informatica

Università di Pisa

Corso Italia 40, Pisa, ITALY

e-mail: {albano,colazzo,ghelli,manghi,sartiani}@di.unipi.it

**This research was partially supported by the MURST DataX Project*

1. Introduction

In the last few years, the trend of publishing and sharing information on the World Wide Web caused much of the existing electronic data to lay outside of database management systems in the form of so-called Web documents. This process was further eased by the introduction of the *eXtensible Markup Language* (XML) by the *World Wide Web Consortium* (W3C) , which provided a standard format for Web-documents.

XML documents consist of textual information organized according to a tagged tree structure, which can be easily employed for representing both structured data and document-like information. Thus, XML represents the first concrete example of integration between the traditionally separated fields of document technology and databases. This unusual integration has generated an enormous demand for new technologies, capable of efficiently operate on both textual information and structured data.

Due to the close relationship between XML and semi-structured data models , most XML query languages have been conceived by extending semi-structured data technology. In these languages, queries consist of "path expressions" to be matched against the tree structure of a document. Unlike traditional query languages and databases, XML query languages are generally untyped, and documents are assumed to come along without a description of their structure. The total absence of meta-information deprived these languages of the benefits typically associated with static type information in DBMSs, i.e. the possibility of checking for query correctness, support for query optimization, storage optimization.

Recently, in order to support the increasing growth of XML-as-database documents, typed approaches to query and process XML data begun to appear (,). Our work follows the same line, trying to exploit type information with the main purpose of checking query correctness, of providing the user with information about the structure of data, and of performing some kinds of query optimization. Our type system is very similar to the one in [3], but we use it to support a language which is quite different: we study a query language, characterized by a bounded complexity and by the possibility of efficient execution, while [3] defined a Turing-complete programming language, where types are used as a matching tool; moreover, we use recursive types to represent cycling and sharing structures, i.e. combination of IDREF and ID attributes.

We focus on *XML-as-database* documents, thus issues such as ranking of documents, weighting of terms, etc. are out of our scope. An exception is made for *free text* searches, which are supported by a special basic type (text), exploited for marking free text sections.

A query language (TEQUYLA, which stands for TypEd QUerY LAnguage) has been defined to query XML documents conforming to a set of given types. However, the system supports mechanisms for inferring a type from an XML document, or for filtering a document with respect to a type, in order to extract the portion of the former that conforms to the latter. Furthermore, one of the main concern in the type system design has been the W3C specification for XML data model and languages, so as to define a type system and a notion of query correctness that could be adopted by any W3C-compliant language. In particular, we rely on a subset of XML Query Data Model , and on the XPath pattern language.

In the following, we shall briefly introduce the data model we refer to, the main concepts behind our type system, and the semantics underlying our notion of query.

2. Data model

The proposed type system is based on a subset of the XML Query Data Model, which represents an XML document as a pointed, node-labeled tree featuring four kinds of nodes: *element nodes*, *attribute nodes*, *value nodes*, and *infoitem nodes*. In Figure 1 an example is shown of an XML document *note.xml* containing both structured data and text.

Figure 1. Document *note.xml*: a software-testing note.

In Figure 2 is shown the representation of the previous document according to the data model, where circles represent element nodes, squares represent attribute nodes, triangles represent value nodes, and rhombuses infoitem nodes.

Figure 2. Representation of *note.xml*.

3. Type System

A type represents an infinite collection of XML documents sharing the same structural properties. For the sake of simplicity, we present the semantics of types by describing informally the kind of documents associated with them, and on the base of an example.

As we do not expect XML documents to be necessarily the representation of regularly structured data, our type language has been conceived so as to capture a wide range of possible structural irregularities:

T ::= ( ) | B | T,T | T + T | L(A)[T] | _ (A)[T] | rec X.T | X

A ::= ( ) | a : T, A

B ::= Int | String | Text | …

Recursive types, denoted by rec X.T, are essential to describe documents that feature cycles, i.e. combination of ID and IDREF attributes, and repeated patterns, i.e. sequences of elements with the same structure. We consider standard recursive types, according to which rec X.T ≡ T[rec X.T/X] (unfolding rule).

Untagged union types, denoted by T1 + T2,represent documents conforming either to T1 or T2.

Labeled types and attributes types, denoted by L(A)[T], describe elements tagged as *L*, in turn associated to a given type T, possibly featuring a list of attributes typed as *A*.

The *underscore*, denoted by _ (A)[T], describes any possible element, despite of its tag, in turn associated to a given type T.

In addition, further classic database requirements, not described here, are fulfilled by the introduction of subtyping mechanisms.

For instance consider the following type, describing the structure of the document in Figure 1. Such type captures the structure of *note.xml*, as well as the structure of other notes, which might have a slightly different aspect: infoitem nodes may not appear at all, and the tag *currentVersion* may also appear nested within an arbitrary element.

```
CompanyType = company[name[string],
                      text + ( ),
                       headquarters[string]
                       ]
note[text + ( ),
      softwareVendors[ rec X.( )      + CompanyType,
text + ( ),
X
                       ],
      text + ( ),
        operatingSystems[ OS( releasedBy: CompanyType )
                       [name[ string ],
                       text + ( ),
currentVersion(lastRevisionDate:int)[string]       +
                       _[
currentVersion(lastRevisionDate:int)[string]]
                            ]
                       ],
          text + ( )
      ].
```

The type above points out that our type system is not exclusively concerned with representing the structure of XML data-oriented documents – documents corresponding to data stored within relational, object-oriented, semi-structured databases, etc. – but it aims also at characterizing documents with a considerable amount of text mixed up with tags identifying a specific underlying structure. In particular, the text type is used to represent sections of free text, over which a database engine can build standard free text indexes; this solution is similar to that adopted by some relational DBMS.

To complete the type system definition an algorithm is required to establish conformity of documents with respect to a type, such as *note.xml* and the type above. The system will be extended with type inference mechanisms so that external XML documents can be typed.

A further aspect of our research is concerned with filtering an XML document with respect to a type (cf.), in order to extract the portion of the document that conforms to that type. As an example, consider the portions (a) and (b) of our sample document as shown in Figure 3. These are the result of filtering the original document according to the following types T1 and T2:

```
CompanyType =company[name[string],
```

headquarters[string]

]

T1 = note[softwareVendors[rec X. ( ) + CompanyType,

X]]

T1 = note[operatingSystems[OS[name[string],

CurrentVersion(lastRevisionDate:int)[string]

]

]

]

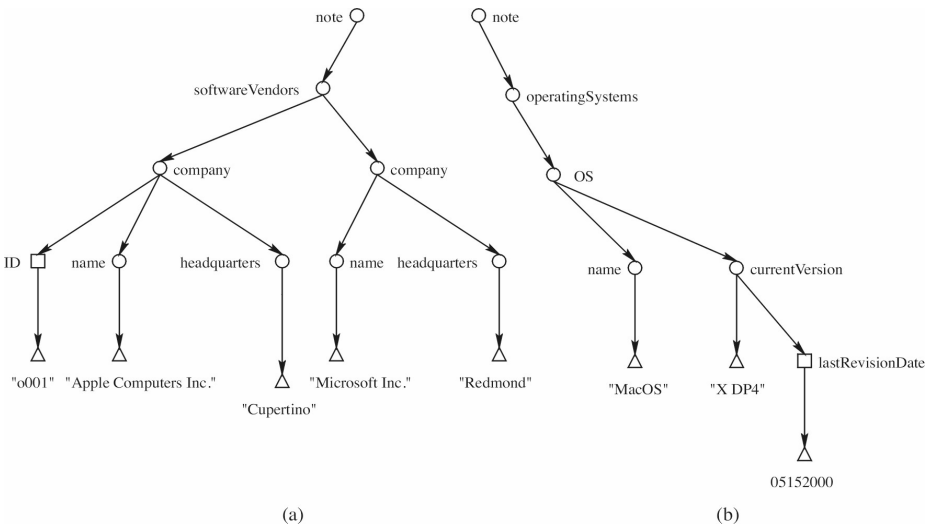

Figure 3. Filtering an XML file with respect to a type.

Once an XML document has been typed, appropriate optimization techniques can be used to retrieve efficiently data. This property, together with a notion of query correctness to be presented in the next section, gives rise to a safe and efficient query language for XML data.

4. Querying XML documents

TEQUYLA is introduced informally by an example. Queries are expressed by means of path expressions mirroring W3C XPath patterns, which can be combined in several ways to express the structure to be searched within the target document. Consider the following query, to be run over the document shown in Figure 1:

"Find headquarters of all software vendors that have released those *chosen* software products".

Let x = document("note.xml") **in**

From

x // InfoItem[text( ) **contains** "We chose"] -> u,

u.next() / OS / @releasedBy / company -> v

Select

Go-to[v / headquarters]

The user knows the document type, and that selected operating systems are distinguished from those which have been discarded by the fact that such notes contain the string "We chose".

4.1 Query semantics

The *let* clause binds the variable *x* to the root node of *note.xml*. Within the *from* clause, variables, such as *u* and *v*, are bound to those nodes that may be reached traversing the tree structure of the document according to the *paths* associated to the variables. For instance the path associated to *u* is *x//InfoItem[condition]*, which identifies all the infoitem nodes which are reachable from *x* at arbitrary depth (//) in the tree, and such that the specified condition holds. The condition states that the text must contain the string "*We chose*". The variable *v* is associated to the path *u.next()/OS/@releasedBy/company*, which identifies the nodes that are reachable from siblings of the nodes associated to *u* (*.next*) by following elements *OS* (/), their attributes *releasedBy* (@), and then elements *company* (/).

The *select* clause specifies how to build query results by using the bound variables introduced in the previous clauses. The query results is an XML document whose root tag is *Go-to* containing a list of all headquarters of the company nodes bound to *v*.

4.2 Query correctness

In general, queries over XML typed documents specify a set of structural properties and a set of logical properties, the former specifying the structure of the document tree to be searched, and the latter expliciting conditions to be satisfied by the nodes. The query shown above, specifies both structural and logical properties within the *from* clause, in the form of paths and conditions ([…]), even though further conditions on the variables might have been specified in a *where* clause. A query is correct when there is a successful match between query structural properties and the document type; correct queries only are executed, since they are the only one that can have a non-empty result. When a system does not support a notion of document type, any query is correct and it is always executed.

The query above is correct since the paths *x//InfoItem->u* and *u.next()/OS/@releasedBy/company->v* do match with the type specified for *note.xml*. Indeed, the type states that there exists at least one infoitem element within the type, and there exists a sibling of such element that is structured as requested by the second path. As a further example of correctness, it is interesting to observe that replacing the second path with the similar one

u.next()/OS+operatingSystems/@releasedBy/company->v,

i.e. a multiple option path, would be correct too, and return the same result as before: the user might be informedthat the tag *operatingSystems* is not present in the type, i.e. in any document conforming to the type.

Query correctness has been defined by a set of rules to typecheck queries; in particular, these rules establish when the XPath patterns specified in a query *respect* the document type.

5. Conclusion

A system for querying XML documents has been presented, which exploits type information for checking query correctness. Given an untyped XML document, a type can be inferred from it. Otherwise a user can choose to filter the document according to a type, in order to query the portion of the document that conforms to the selected type. Once a document has been assigned a type, a query can be formulated on the base of the information conveyed by the type, and be checked for correctness.

Algorithms for checking conformity, and query correctness have been defined and are under development. Type inference as a general typing approach, and type filtering for XML data, have been already discussed in the literature, and will be adapted to the type system.

References

1. Bray, T., J. Paoli, and C. Sperberg-McQueen. *Extendible Markup Language (XML) 1.0*. In *W3C recommend*. 1998.

2. Suciu, D., *Semistructured Data and XML*, 1998.

3. Hosoya, H. and B.C. Pierce, *XDuce: A Typed XML Processing Language*, 2000, Department of CIS - University of Pennsylvania.

4. Buneman, P. and B. Pierce. *Union Types for Semistructured Data.* In *Proceedings of the International Database Programming Languages Workshop*. 1999.

5. Fernandez, M. and J. Robie, *XML Query Data Model*, 2000, W3C World Wide Web Consortium.

6. Connor, R.H., et al., *SNAQue: a Mechanism for Programming over XML Data*, 1999, University of Strathclyde, Glasgow.

7. Connor, R.H., P. Manghi, and F. Simeoni, *Extracting Typed Values from Semistructured Collections*, 1999, University of Strathclyde, Glasgow.