A Typed Text Retrieval Query Language for XML Documents[†]

Dario Colazzo Carlo Sartiani Antonio Albano Paolo Manghi Giorgio Ghelli
Dipartimento di Informatica
Università di Pisa
Corso Italia 40, Pisa, ITALY
e-mail: {colazzo, sartiani, albano, manghi, ghelli}@di.unipi.it

Luca Lini Michele Paoli
Centro Ricerche Informatiche per i Beni Culturali
Scuola Normale Superiore
Via Della Faggiola 19, Pisa, ITALY
e-mail: {lini, zizzi}@cribecu.sns.it

[†] Dario Colazzo, Carlo Sartiani, Paolo Manghi, Antonio Albano, and Giorgio Ghelli were partially funded by the MURST DataX Project

ABSTRACT

XML is nowadays considered the standard meta-language for document markup and data representation. XML is widely employed in *Web-related* applications as well as in database applications, and there is also a growing interest for it by the literary community in order to develop tools for supporting document-oriented retrieval operations. The purpose of this paper is to show the basic new requirements of this kind of applications and to present the main features of a typed query language, called Tequyla-TX, designed to support them.

1. Introduction

During the last few years XML has rapidly emerged as a standard meta-language for document markup and data representation. XML was designed as a simplification and an evolution of SGML (ISO, 1986), with the purpose of broadening the language application field and of easing the development of efficient management tools.

XML has been widely employed in the document community, in many *Web-related* applications, as well as in many database applications. Its ubiquitous presence is due to the ability to represent nearly any kind of data sources, ranging from structured data (e.g., database records) to semistructured databases (e.g., collections of BibTeX references), and to unstructured data (e.g., DNA sequences).

To exploit this great flexibility of XML it is important to have managing tools (e.g., querying, indexing, and storing tools), which can efficiently manage structured, semistructured, as well as unstructured data. In particular, there is the need for query languages able to support both database-like queries and textual queries.

This paper describes Tequyla-TX, a *typed* text retrieval query language for XML documents. Tequyla-TX is an extension of Tequyla (Albano, Colazzo, Ghelli, Manghi, & Sartiani, 2000b) (Albano, Colazzo, Ghelli, Manghi, & Sartiani, 2000a), a typed query language for XML data, from which it inherits the ability to perform database-like queries; Tequyla-TX integrates these features with the support for both *word-based* and *char-based* searches to satisfy the basic requirements of text retrieval applications and, in particular, of literary applications.

The paper is organized as follows. Sections 2 and 3 present the motivations of the project as well as a brief overview of other query languages; Section 4 presents the Tequyla-TX language by means of examples. Sections 5 and 6 are focused on the type system and the query algebra of Tequyla-TX. In the conclusions we briefly comment the work in progress.

2. Motivations

Tequyla-TX is the result of a collaboration between the Database Research Group of the University of Pisa and the CRIBECU (Centro Ricerche Informatiche per i Beni Culturali) group of the Scuola Normale Superiore di Pisa. CRIBECU expertise is in the field of text retrieval for literary applications, such as philological analysis of literary texts, textual criticism, stylistic analysis, and linguistic research.

To support these kinds of applications CRIBECU has implemented a search engine TreSy (Text Retrieval System) (Corti, Lombardini, & Paoli, 2000) based on a particular text-indexing data structure, called *String B-tree* (Ferragina & Grossi, 1999). The goal of Tequyla-TX is to integrate a typed XML query language with the TReSy search engine to support the following basic functionality required by a wide range of literary applications: search on both content and structures; *word-based* and *char-based* searches; *tag-dependent* full text searches; match points; text normalization.

Search on both content and structure Traditional query languages for text and information retrieval allow one to query the content of documents only; this solution is not appropriate for XML documents, where data can be arbitrarily nested and mixed with structural information. In these cases, both content and structure of XML documents have to be queried.

Example 1. *Consider the following fragment of the Italian "Vocabolario della Crusca", where the first entry defines the meaning of the word 'ANITRACCIO' (Tuscan word for duckling), and the second one defines its synonym 'ANITROCCOLO':*

```
   ...
<entry ID="B6">
   <form type="lemma"> ANITRACCIO </form>
   <sense rend="definizione"
          value="semplice">
      ...
   </sense>
</entry>
...
<entry id="B10">
   <form type="lemma"> ANITROCCOLO </form>
   <xr> vedi
      <ptr target="B6">
         <hi>ANITRACCIO</hi>.
      </ptr>
   </xr>
</entry>
   ...
```

Assume that we want to find the definition of the word 'ANITRACCIO', without retrieving the definition of its synonym. We can express this query by combining a content constraint (look for 'ANITRACCIO') with a structural one (look inside form *elements contained into* entry *elements). Indeed, the specification of a content constraint only would return the definition of 'ANITRACCIO' as well as any other reference to it.*

Word-based and char-based searches XML documents are used for representing various kinds of information sources: texts, semistructured databases, scientific databases, and so

on. Given the wide application field of XML, the ability to perform both word-based and char-based searches (i.e., the search for patterns that cross word boundaries) is required; in particular, char-based searches are necessary in many literary applications, such as analyses of the use of prepositions in Latin texts.

Tag-dependent full text searches Literary applications frequently use in XML documents tags that do not show the same behavior w.r.t content searches. For example, it is common to find three different classes of tags. The first class contains tags (called *hard* tags), that describe the logical organization of documents, such as the sectioning in chapters, section, and so on; hence, they divide documents into distinct *search contexts*, whose borders should not be crossed by a search function (a search context is just a string).

The second class contains tags (called *soft* tags) that describe the typographical layout of documents. Even though one key idea behind XML is the separation between content (the XML document itself) and presentation (XSL stylesheets), still there is the need for encoding some layout information into documents themselves. For example, documents representing ancient manuscripts usually contain detailed layout descriptions, considered as important information by researchers; moreover, a proper rendering of such documents cannot be handled by XSL and XSLT engines (it is just too complex for XSL). These tags, describing typographical aspects, do not define new search contexts, and they should be transparent to a content search function

The third class contains tags (called *jump* tags) used for representing footnotes, endnotes, bibliography references, etc, which require a special treatment during retrieval operations. Consider, for example, a literature paper containing notes at the end of each page, explaining the meaning of particular statements, or giving the reader further information about a particular topic. These notes are not part of the main search context, but indeed they define a new one.

Example 2. *Consider the following fragment of XML:*

```
...
<chapter>
   <title> Anatomy of a conspiracy </title>
   ...
   <section id = 2>
      <hi> OAS </hi>
      <note> Organization de l'arm&eacute;e
             secr&egrave;te  </note>
      was …
   </section>
   …
   <section id = 10> ...  Paris </section>
   <section id = 11> The Jackal ...
   </section>
   ...
</chapter>
```

...

Element tags such as chapter*,* title*, and* section *describe the logical structure of the document; hence they are classified as hard tags. The tag* note*, instead, is used to describe the acronym 'OAS', and it is classified as a jump tag. The tag* hi*, finally, is used to describe that the word 'OAS' is capitalized, so it is classified as a soft tag.*

These tags define various search contexts, as shown in Figure 1.

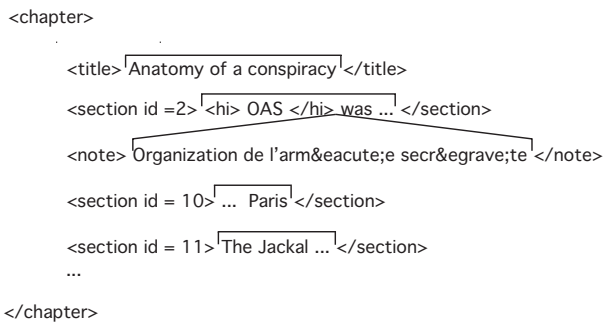

Figure1. Search contexts within an XML document

It should be noted that the note *element does not break the search context starting from OAS.*

Now, assume that we want to locate, in the previous fragment, the pattern 'Paris The Jackal'; since this pattern does not occur into a single search context, this query should fail on this fragment. A search for the pattern 'OAS was', instead, would succeed, since the tag hi *(surrounding 'OAS') is transparent to content searches, and the tag* note *does not break the current context.*

Match points Literary researchers are mainly interested in locating the portions of a document (or of a collection of documents) satisfying the search criteria, and in viewing them in the context of the original documents.

Thus, the result of a query should be a set of *match points*, i.e., a set of pointers to the original documents identifying the relevant portions of text, instead of a set of strings or XML elements (which could be very long and hard to read) as in traditional XML query languages (Deutsch, Fernandez, Florescu, Levy, & Suciu, 1998) (Robie, Lapp, Schach, Hyman, & Marsh, 1998).

Example 3. *Consider the following XML fragment:*

```
<paragraph> ... for two reasons ...
    ... set. For example
</paragraph>
```

The result of a query for finding any occurrence of the string 'for' inside the paragraph *element should contain two match points, as shown in Figure 2.*

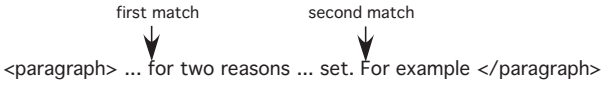

Figure 2. Match points

Text normalization White spaces, punctuation marks, and other symbols require a special treatment during query processing. For example, the search for the pattern 'the car' should locate also 'the__car', ignoring the extra white spaces in the text. In other words, the text should be normalized in order to be queried. This approach, coming from the PAT System for Oxford English Dictionary (OED) (Salminen & Tompa, 1994), allows the text to be transformed depending on the requirements of the specific application field. In particular, text normalization in the context of literary applications consists of three main transformations:

- resolution of XML entities representing accents and other special characters, e.g., città -> 'città';
- transformation of punctuation marks into word separators, e.g., 'It was a nice day. Nevertheless …' -> 'It was a nice day Nevertheless';
- fusion of contiguous word separators into a single one, e.g., 'it was \n a nice day' -> 'it was a nice day'.

In Section 4 it will be shown how text normalization can be confined into specific language functions, thus enhancing the flexibility of the query language.

3. Related Works

There exist many query languages for XML documents, coming from both the database community and the XML community; there also exist languages specifically designed for text-retrieval on structured documents (Navarro, 1995) (Neumann, 2000). None of them, including the original Tequyla, fully meets the above requirements.

XML-QL and Lorel (Deutsch, Fernandez, Florescu, Levy, & Suciu, 1998) (Abiteboul, Quass, McHugh, Widom, & Wiener, 1997), coming from the database community, are based on the SSD paradigm. They share many common features, such as the ability to perform querying operations as well as transformations, the use of graph data models (simple variants of OEM (Papakonstantinou, Widom, & Molina, 1996)), a copy semantics, etc.

Unfortunately they do not satisfy the requirements described in the previous section. In the first place, there is not a notion of match point, nor there are operators for binding variables to match points, and for manipulating them.

In the second place, XML-QL support to text retrieval is limited to a string containment predicate *like*, which compares the value of an element or attribute with a string regular expression. Lorel, instead, provides some textual operators, the same predicate *like*, as well as an operator for reconstructing the text contained in an XML tree, but they are still limited and do not properly manage white spaces.

Example 5. *Consider the following XML fragment:*

```
<chapter> <title> File Structures </title>
   <section> <title> Inverted File </title>
   ...
   </section>
</chapter>
```

We want to locate the character sequence 'tures invert' contained (directly or indirectly) into a chapter *element.*

This query cannot be expressed into XML-QL nor in Lorel, since they both lack support for match points. By relaxing this requirement, this query can be formulated in Lorel as follows.

```
select flatten(y)
from ...
where flatten(y) like "%tures invert%"
```

This query returns the text contained in the whole chapter, by means of the flatten *operator, which does not take care of tags with different behaviors.*

In (Florescu, Kossmann, & Manolescu, 2000) an extension of XML-QL that enables *keyword searches* is described. This extension consists of a boolean predicate *contains*, which tests whether a given word is contained into an XML tree at a specified depth; *contains* allows one to perform searches on tag names, attribute names, element content, and attribute values (or on any arbitrary combination of them).

While the proposed predicate is useful for expressing queries over documents, whose structure is unknown or heterogeneous, it does not suffice for text retrieval purpose, since it does not overcome the limitations of XML-QL.

XQL is a query language designed by Jonathan Robie (Robie, Lapp, Schach, Hyman, & Marsh, 1998) (Robie et al., 1999), and vastly adopted in the XML community. An XQL query has the form of an XPath pattern (Clark & DeRose, 1999), enriched with filtering conditions (*inline conditions*) and the ability to perform *join*-like operations on different fragments of a document.

XQL has a limited expressive power (Fernandez, Siméon, & Wadler, 1999), since it cannot express general transformations on query results. Moreover, XQL has not a well-defined semantics, which leaded to many inconsistent implementations (e.g., there is no agreement on whether XQL joins are *inner* joins or *outer* joins).

XQL offers many textual operators, which allow one to perform case sensitive or insensitive searches, to reconstruct the text contained into an XML tree, and to compare the position of different words (*before*, *after*), but still it does not satisfy our major requirements. First of all, XQL does not support match points; moreover, textual operators apply only to *word-based* searches (e.g., there is no support for *char-based* searches); finally, text normalization is not supported.

Example 6. *Assume that we want to locate any occurrence of the pattern 'ex a' inside a student translation from Italian to Latin (this control is useful since the preposition 'ex' applies only to words beginning with a vowel). Finding these occurrences requires to abstract from the word representation of the text, and to work on the char representation, which is not feasible in XQL.*

XIRQL (Fuhr, 2000) is an evolution of XQL for information retrieval applications. It extends XQL in three major ways. First of all, it contains operator for performing similarity searches, and for weighting and ranking query results. These similarity operators are defined not only on strings, but also on numbers, dates, and other relevant datatypes.

Second, XIRQL introduces a notion of *result contexts*, which is, to some extent, similar to the notion of match points. The key idea behind result contexts is to identify rooted, possibly nested, subtrees of an XML document, whose roots are then returned as results whenever the corresponding subtree matches the search criteria. Result contexts are statically defined by the database administrator, who enriches the document schema with such information.

Finally, XIRQL endows XQL path language with a keyword '-', used for matching both elements and attributes, e.g., `-author` matches both author elements and author attributes.

Despite these extensions, XIRQL still does not satisfy our requirements, since it does not support char-based searches nor tags with different behaviors.

XQuery is the standard XML query language being defined by the W3C (Chamberlin, Florescu, Robie, Siméon, & Stefanescu, 2001). XQuery core is based on Quilt, a Turing-complete query language for XML designed by Don Chamberlin, Daniela Florescu, and Jonathan Robie (Chamberlin, Robie, & Florescu, 2000). XQuery combines ideas borrowed from XML-QL, such as the ability to perform general transformations, from XQL, such as the use of XPath patterns instead of generalized path expressions, and from XSLT.

Given its Turing-completeness, XQuery proved to be the most expressive XML query language, meeting all W3C requirements (Chamberlin, Fankhauser, Marchiori, & Robie, 2000).

XQuery inherits the textual operators of XQL, which it combines with a clear support for external functions; this might be exploited for extending the language with functions that take care of tags with different behavior (see Section 4 for a similar solution in the context of Tequyla-TX).

Nevertheless, XQuery suffers the same difficulties in handling the previously described requirements as XML-QL and XQL, since it does not support match points, nor it is able to express char-based searches. We stress that XQuery definition is currently *in-progress*, hence its features are subject to change.

4. Tequyla-TX By Examples

Tequyla-TX is an extension of Tequyla, a typed query language for XML documents. Tequyla is based on the SSD paradigm, and even though it satisfies most of the technical requirements expressed in (Chamberlin, Fankhauser, Marchiori, & Robie, 2000)., it is quite different from other XML query languages. First of all, unlike XML-QL, Tequyla supports XPath patterns, in order to lower the evaluation cost of path expressions. Moreover, Tequyla is typed, which means that queries are statically checked against the database schema in order to determine their correctness, and that the type of the result of a correct query is statically computed by the system (further details on Tequyla can be found in (Albano, Colazzo, Ghelli, Manghi, & Sartiani, 2000b)).

In order to fulfill the requirements described in Section 2, Tequyla has been extended with *match point* variables, and with operators for binding and manipulating them.

In the following sections, after a brief overview, Tequyla-TX will be introduced by means of examples, some of which come from the W3C XML Query Requirement SGML Use Case (Chamberlin, Fankhauser, Marchiori, & Robie, 2000).

4.1 Language Overview

A Tequyla-TX query Q is written as a free nesting of from-select binders, path expressions, and forest construction operators, such as $l(a)[Q_1,...,Q_n]$, which builds an XML element tagged with l, whose attributes are specified in a, and whose content is the concatenation of the results of queries $Q_1, ...,Q_n$.

For example, the constructor

instrument(*type* = "*ER*")["*Foley catheter*"]

return the following XML element:

```
<instrument type = "ER"> Foley catheter
</instrument>
```

A query Q always denotes a forest, i.e., a sequence of trees, which may consist of one tree only.

from x in Q where W select Q' evaluates *Q'* once for each different binding $x = t$, where t ranges over the trees that compose the forest denoted by Q; all the forests f_i produced by these evaluations of Q are collected to obtain the query result $f_1,...., f_n$. The where clause cancels all those bindings that do not satisfy W. In general, the subquery Q is a pattern expression $Q''p$, which denotes the sequence of all subtrees that are reached by starting from a tree in Q'' by following the path p. Paths p are expressed through a simplified form of XPath patterns.

Example 7. *Consider the following Tequyla-TX query (applied to an XML document describing medical procedures (Chamberlin, Fankhauser, Marchiori, & Robie, 2000)):*

```
from x in instrument,
where x.value() = "Hasson trocar"
select strange_instrument[x.value()]
```

The from *clause binds x in turn to each instrument; the* where *clause, then, checks whether the content of x is equal to 'Hasson trocar', cutting off the other bindings.*

The select clause*, finally, builds a new XML element, whose content is the content of x.*

from x = Q select Q' evaluates *Q'* only once, and substitutes *x* with the whole forest denoted by *Q*.

Tequyla-TX *from-where-select* expressions are very similar to Quilt and XQuery FLWR expressions.

We abbreviate a *from select* pair with a comma, hence a nested query

from A select (from B select Q)

will be written as *from A,B select Q*.

4.2 W3C's SGML Use Case

The example document and queries in this Use Case are based on the W3C's SGML Use Case. A simplified DTD is given below.

4.2.1 DTD

This use case is based on an implicit (unnamed) input data set, using the DTD shown below.

```
<!NOTATION cgm PUBLIC "Computer Graphics
Metafile">
<!NOTATION ccitt PUBLIC "CCITT group 4
raster">
<!ENTITY % text "(#PCDATA | emph)*">
<!ENTITY infoflow SYSTEM "infoflow.ccitt"
NDATA ccitt>
<!ENTITY tagexamp SYSTEM "tagexamp.cgm" NDATA
cgm>
<!ELEMENT report (title, chapter+)>
<!ELEMENT title %text;>
<!ELEMENT chapter (title, intro?, section*)>
<!ATTLIST chapter shorttitle  CDATA #IMPLIED>
<!ELEMENT intro (para | graphic)+>
<!ELEMENT section (title, intro?, topic*)>
<!ATTLIST section  shorttitle CDATA #IMPLIED>
<!ELEMENT topic (title, (para | graphic)+)>
<!ATTLIST topic  shorttitle CDATA #IMPLIED>
<!ELEMENT para (#PCDATA | emph)*>
<!ATTLIST para security (u | c | s | ts) "u">
<!ELEMENT emph %text;>
<!ELEMENT graphic EMPTY>
<!ATTLIST graphic  graphname ENTITY #REQUIRED>
```

4.2.2 Queries

The following query shows the use of from and select clauses, the two kinds of variable binders of Tequyla-TX as well as the use of XML constructors.

Example 8. *"Locate all paragraph elements in an introduction (all* para *elements directly contained within an* intro *element)."*

```
result[
   from report = document("report.xml"),
        introduction in report//intro,
        paragraph in introduction/para
   select paragraph
]
```

`result[...]` *is an XML constructor, which builds an XML element tagged* result*, whose content is specified within the square brackets.*
The from *clause introduces variable bindings, which will be exploited in the* select *clause. In Tequyla-TX, a variable can be bound to a forest of XML trees as well as to a single tree. As in XML-QL, variable bindings are organized in tuples; hence, the result of the* from *clause is a set of bindings tuples.*

In this particular query, the from *clause first binds* report *to the root of the document, whose URI is specified in the* document *function; next, it builds the set of all* intro *elements descendant of* report*, and binds each of them in turn to the* introduction *variable.*

The following query shows the use of indexes in Tequyla-TX paths. Indexes, as well as inline conditions, are exploited to filter the nodes in the current evaluation node set (context in XPath jargon); indexes, in particular, select nodes by their position in the current evaluation context.

Example 9. *"Locate the second paragraph in the third section in the second chapter (the second* para *element occurring in the third* section *element occurring in the second* chapter *element occurring in the report)."*

```
result[
   from report = document("report.xml"),
        chapter = report/chapter[2],
        section = chapter/section[3],
        paragraph = (section//para)[2]
   select paragraph
]
```

Hence, `report/chapter[2]` *selects exactly the second* chapter *element in the* report *subtree.*

The following query shows the use of the where clause as well as how attributes can be examined.

The where clause is used, as in SQL and other languages, to filter the bindings built by the from clause; the where clause

evaluates its condition for each binding tuple in the environment, and discards every tuple not satisfying it.

Example 10. *"Locate all classified paragraphs (all* para *elements whose* security *attribute has the value C)."*

```
result[
   from report = document("report.xml"),
        paragraph in report//para
   where paragraph/@security = "C"
   select paragraph
]
```

In this particular example, the where *condition examines the* security *attribute for each paragraph element, and checks whether its value is C; this is done by translating* `@security = "C"` *into* `@security.value() = "C"`*, hence using the value extraction function* value().

The following query shows how the value of attributes and simple elements can be extracted and manipulated, as well as how to test the existence of optional attributes.

Example 11. *"List the short titles of all sections (the values of the* shorttitle *attributes of all* section *elements, expressing each short title as the value of a new element)."*

```
result[
from report = document("report.xml"),
   section in (report//section)[@shorttitle],
select shorttitle[section/@shorttitle.value()]
]
```

In the from *clause, the condition* `[@shorttitle]` *checks whether the shorttitle is present in the current* section *element; this control is required since the* shorttitle *attribute was declared as optional in the DTD (*`<!ATTLIST ... shorttitle CDATA #IMPLIED ...>`*).*

The value of the shorttitle *attribute can be extracted by using the* value() *function; this function retrieves the value of attributes as well as simple elements, i.e., elements without mixed or element content. The* value() *function does not make any normalization of text, nor it can be used on complex elements.*

4.2.3 Text Retrieval Use Case

The following query shows the use of the textual operator *CONTAINS* in the where as well as the function *textof()*.

The *textof()* function applies to a tree and returns a sequence of strings. This sequence is built by traversing the tree, and merging CDATA sections and the values of simple elements.

During the merging process, the behavior of tags is taken into account. As already stated, three kinds of tags are used: hard tags, soft tags, and jump tags. A different behavior is associated to each kind of tags: hard tags delimit strings that cannot be merged; soft tags, instead, are invisible to the merging function; jump tags, finally, delimit new strings, as hard tags, but do not halt the construction of the previous string.

The *textof()* function also accomplishes the normalization task, by applying transformations to white spaces, punctuation marks, etc.

It should be noted that the semantics of the *textof()* function is not fixed in Tequyla-TX, but it can be customized to be adapted to the user's needs, without the need to modify the core language: hence, it is possible to define specialized version of *textof()*, as long as typing constraints, described in Section 6, are satisfied. Moreover, confining tags behavior and normalization issues into the *textof()* function made the data model and semantics Tequyla-TX less complex.

Example 12. *"Find the entries referring to the definition of the word 'ANITRACCIO'."*

```
results[
from dict =
   document("www.cribecu.sns.it/crusca.xml"),
   en in dict//entry,
   xr in entry//xr
where xr.textof() contains $"ANITRACCIO"$
select entry
]
```

The CONTAINS *operator looks for the word 'ANITRACCIO' ($ is a word delimiter) in the string sequence returned by the* textof() *function. By default, the* CONTAINS *operator does not take care of the case, i.e., string matching is case-insensitive.*

The following query shows the use of wildcards in textual operators. The *?* symbol is a wildcard matching any character; hence, the expression *"Anner"?"re"* matches both 'ANNERARE' and 'ANNERIRE'.

Example 13. *"Find the definitions of the words 'ANNERIRE' or 'ANNERARE'."*

```
results[
from dict =
   document("www.cribecu.sns.it/crusca.xml"),
   en in dictionary//entry,
   form in en/form
where en.textof() contains $"Anner"?"re"$
select en
]
```

The following query introduces the use of match points. A match point is a pointer to an occurrence of a given string into a document. Match points are bound to variables by means of textual operators in the from clause.

A match point is represented as a rooted tree with the following structure, shown in Figure 3. The tree is labeled with tags coming from the Tequyla-TX namespace (http://tequyla.di.unipi.it).

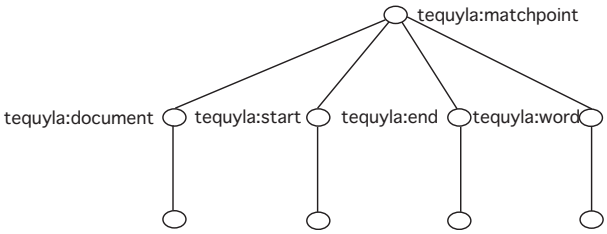

Figure 3. Structure of a match point

The document element is a pointer to the root of the XML document containing the match point; the start and end elements identify the position of the string into the document, by specifying the position of the first and last character of the string into the document; the word element, finally, identifies the position of the string into the word-based representation of the text, and it is meaningful for word-based searches only.

Example 14. *"Locate the occurrences of the word 'E'."*

```
results[
from dict =
   document("www.cribecu.sns.it/crusca.xml"),
   en in dict//entry,
   occur in en CONTAINS case-sensitive $"E"$
select occur
]
```

The CONTAINS operator looks for the word 'E' in the textual representation of en *(i.e., the result of the* textof() *function); for each occurrence of the word 'E' (case-sensitive), a match point describing it is created.*

Each match point is then bound in turn to the variable occur.

The result of this query is an XML element results *containing a sequence of match points.*
They are then used by a post-processing tool, which displays the related occurrences in the context of their original documents. Moreover, they are standard Tequyla-TX entities, which implies that they can be manipulated by Tequyla-TX operators, as it is shown in the following example.

The following query shows the use of the textual operator *before()*. It compares the position of the first operand (a match point) with the position of the match points that correspond to the second operand: it returns *true* if there exists at least one match point (described by the second operand) in the desired position w.r.r the first operand (existential semantics). The *before()* operator is a word based operator. The language also offers a char-oriented operator *beforechar()*, which compares operand positions by chars, as well as their complementary operators *after()* and *afterchar()*.

Example 15. *"Locate all the occurrences of the word 'signific.' preceding the word 'pass.' by one word."*

```
results[
from dict =

document("www.cribecu.sns.it/crusca.xml"),
     en in dict//entry,
     sen in en//sense,
     occur in sen contains $"signific."$
where before(occur, "pass.", 1)
select occur
]
```

The following query shows a way to avoid text normalization.

Example 16. *"Find all entries defining the word 'ANNEGAMENTO'."*

```
results[
from dict =
    document("www.cribecu.sns.it/crusca.xml"),
    en in dict//entry,
    form in en//form
where form.value() = "ANNEGAMENTO"
select en
]
```

The where *clause contains a call to the* value() *function, which extracts the* un-normalized *value of simple elements and attributes; this value is then compared with the string 'ANNEGAMENTO'.*

The expression `form.value() = "..."` *is not semantically equivalent to* `form = "..."`*, since Tequyla-TX by default applies the* textof() *function whenever the content of an element has to be compared with a string. Hence,* `form = "..."` *is equivalent to* `form.textof() = "..."`*, which normalizes the content of the* form *element.*

The following query is based on an XML fragment showed below.

```
…
<lb n="8"><name type="Fonti">AENEA</name>
<name type="Fonti">VICO</name> PARMENSI</lb>
<lb n="9">OLIM</lb>
<lb n="10">EDITA</lb>
<lb n="11">Noviter additis eorumdem
   <hi>C</hi>AESARUM  imaginibus  maiori
</lb>
<lb n="12">forma a praestantioribus
    Calchographis aeri incises </lb>
<lb n="13">Eminentissimo ac Reverendissimo
    Principi Domino</lb>
…
```

The query shows the use of the *contains-start* operator. This textual operator takes as argument an XML element and a string, and verifies that the string is contained in the textual representation of an ancestor of the element, and that it begins in such element.

Example 17. *"Locate all highlighted occurrences of the word 'CAESARUM' (all occurrences of 'CAESARUM' which begin inside a* hi *element)."*

```
results[
from bellorii =
  document("www.cribecu.sns.it/bellorii.xml"),
  div1 in bellorii/div1,
  caesar in div1//hi contains-start
                      $"CAESARUM"$
select caesar
]
```

The contains-start *operator goes back to the father of the selected* hi *element (which denotes an highlighting), and builds its textual representation by calling the* textof() *function, whose result is shown below:*
{"Noviter additis eorundem CAESARUM imaginibus maiori"}.

Then, it tests whether the string 'CAESARUM' is contained in this textual representation, and, finally, whether this string begins within the hi *element.*

5. Tequyla-TX Type System

The Tequyla-TX type system extends that of Tequyla with rules for typing textual operators.

A formal presentation of the type system is given in Appendix C. In this section we give an overview of its main features.

5.1 Motivations

Unlike traditional query languages and databases, most XML query languages are generally untyped, and documents are assumed without a description of their structure. The absence of meta-information deprives these languages of the benefits typically associated with static type information in DBMSs, i.e. the possibility of checking for query correctness and many query optimizations.

Tequyla-TX has been designed to exploit the benefits of type checking. The language type system is used to check whether XML documents conform to schema descriptions, and to check whether queries are valid with respect to the type of queried documents.

Tequyla-TX type system presents several differences w.r.t. recent typed approach to query XML data (Fankhauser et al., 2001) (Hosoya & Pierce, 1999) (Chamberlin, Fankhauser, Marchiori, & Robie, 2000) (Milo, Suciu, & Vianu, 2000).

Our type system is very similar to the one in Xduce (Hosoya & Pierce, 1999), 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 (Hosoya & Pierce, 1999) defines a Turing-complete programming language, where types are also used as a matching tool. Moreover, Tequyla-TX pattern language is based on XPath and inherits its semantics (*many-matches*), while (Hosoya & Pierce, 1999) is based on a ML-like pattern language whose semantics (*one-match*) is quite different from the one of XPath.

The main feature of our type system is the presence of mechanisms able to individuate incorrect queries, that is queries containing paths that cannot be present in the data being queried. This is the main difference w.r.t. (Fankhauser et al., 2001), where this kind of queries are considered as correct.

Moreover, (Fankhauser et al., 2001) is a Turing complete XML query algebra, and the type of functions has to be declared by the user. In particular, (Fankhauser et al., 2001) uses recursive functions to express the XPath deep path //. This means that, for (Fankhauser et al., 2001) queries which make use of //, the query type has to be declared by the user. Differently, Tequyla-TX is a pure query language (it is not Turing-complete) but, as we have seen in previous sections, a form of structural recursion is built-in in its pattern language. In particular, the type of queries, which make use of structural recursion, must not be declared by the user, since Tequyla-TX computes the query type.

We have to observe that (Fankhauser et al., 2001) is a W3C work in progress, hence some of the above observations may not hold for future versions of that work.

Regarding XQuery type system (Chamberlin, Florescu, Robie, Siméon, & Stefanescu, 2001), the same consideration made about (Fankhauser et al., 2001) hold, since typing for XQuery heavily relies on the typing system of its core language, which essentially consists of the system in (Fankhauser et al., 2001).

Finally, our approach is different from (Milo, Suciu, & Vianu, 2000) where the typechecking problem is studied for a particular class of XML transformation programs. In particular, given a transformation program and a DTD for the

input documents, it is considered the problem of checking whether every output of the program conforms to a given output DTD. To this end, the given output DTD plays a crucial role since typechecking is performed by a kind of backward type inference mechanism, which takes as argument the transformation program and the output DTD. Instead, in Tequyla-TX type system the focus is on input documents type, and outputs type are inferred.

5.2 Type System Overview

Tequyla-TX exploits typing for two different aims:

- to check query correctness;
- to compute the type of query results (the query type).

This two features are strictly related, since to check correctness of a query with subqueries, the types of subqueries are needed.

To introduce the notion of query correctness that is supported by Tequyla-TX type system, in the following we will refer to some examples of correct and incorrect queries over typed XML data. For simplicity, we will use DTDs as a formalism to describe data and query types. However, the actual system relies on a type language which strictly resembles the one of (Hosoya & Pierce, 1999) and is more expressive than DTDs.

The semantics of queries has already been explained in the previous section. To better understand the notion of correctness, recall that, in Tequyla-TX, the from clause specifies the structural requirements of the query, that is, the structure of the document tree to be searched, and the where clause specifies the logical properties to be satisfied by nodes bound to variables in the from clause.

Tequyla-TX type system considers a query as correct when there is a successful match between the query structural requirements and the type of data to be queried. In other words, the structure that the query expects to find in the data conforms to the actual structure of the queried data declared in its type (DTD).

As we will see, actually, our notion of query correctness is a notion of *partial* query correctness, in the sense that if alternative paths are present in a query, then the query is correct if at least one alternative is present in the data (see the query example Q7). Observe that this is an arbitrary choice, and the system can be easily modified in order to consider queries with alternative paths as correct only if all the alterative paths exist in the data.

Only correct queries will be executed over typed data. If a query is incorrect w.r.t. to a DTD, then it will be rejected and notified to the user. In particular, for incorrect queries, the system can be implemented so that incorrect paths can be indicated to the user in order to easily allow query corrections.

To give some example of correct queries, we consider documents typed by the following DTD.

DTD1=

```
<!DOCTYPE Person[
<!ELEMENT Person (firstname,secondname) >
<!ELEMENT firstname #PCDATA>
<!ELEMENT secondname #PCDATA >]>
```

This simple DTD describes documents containing one person element whose content is given by a firstname element followed by a secondname element.

Let d be an XML document satisfying the DTD above. An example of correct query over d is the following.

```
Q1 =    result[
        from x in d/firstname
        select name[x.value()]]
```

Indeed, according to the previous definition of correctness, the structural properties of Q1 require that the root of d contains some firstname element. These structural requirements successfully match with the type of d.

Tequyla-TX type system also computes the type of Q. In this case the computed type is

```
<!DOCTYPE result[
<!ELEMENT name #PCDATA>
]>
```

In the following case we consider a query Q2 over documents d typed by the following slightly different DTD.

DTD2=

```
<!DOCTYPE Person[
<!ELEMENT Person
        (firstname, secondname, tel_list)>
<!ELEMENT tel_list (tel_number)*>
<!ELEMENT firstname #PCDATA>
<!ELEMENT secondname #PCDATA >
<!ELEMENT tel_number #PCDATA >]>
```

Now, a person element also contains a sequence of zero or more tel_number elements.

The query Q2 is

```
Q2 =    result[
        from x  in d/tel_list/tel_number
        select number[x.value()]]
```

This query is correct, and now we have iteration over an arbitrary sequence of tel_number, as described by the DTD2. Hence, the system returns the following type for Q2:

```
<!DOCTYPE result[
<!ELEMENT result number*>
<!ELEMENT number #PCDATA >
]>
```

If in DTD2 *we* impose that at least one tel_element has to be present,

…

```
<!ELEMENT tel_list (tel_number)+>
```

…

then the type of Q2 changes accordingly:

```
<!DOCTYPE result[
<!ELEMENT result number+>
…]>
```

For documents d typed by DTD2, the following two queries are considered incorrect:

```
Q3=     result[
        from x in d/name
        select name[x.value()]]

Q4=     result[
        from x  in d/tel_number
        select number[x.value()]]
```

Q3 is incorrect since it searches for name elements inside a person element. As stated by the DTD, this requirement is not valid since a person element contains no name elements.

Q4 is incorrect because path for tel_number elements is wrong, since it assumes tel_number elements as children of person element. If we slightly modify Q4, we obtain the following correct query:

```
Q5=     result[
        from x  in d//tel_number
        select number[x.value()]]
```

Now, the query requires tel_number elements at an arbitrary depth from the root element, and this is valid for the DTD2.

Tequyla-TX type system always terminates in checking query correctness, even in the presence of recursive DTDs and of path expressions containing the deep path operator //. To guarantee termination, standard techniques used in type systems dealing with recursive types have been adopted (see (Amadio & Cardelli, 1993) (Colazzo & Ghelli, 1999)).

For incorrect queries, Tequyla-TX returns an output validity parameter with the value "no", to indicate that some query patterns are not valid w.r.t. the type of queried documents. Moreover, Tequyla-TX also returns the type of documents produced by the query; typically this is the empty sequence type, but it can also be a different type when query incorrectness is due to an incorrect subquery with empty sequence type.

The validity output parameter is needed since it may happen that a correct query has the empty type. Consider documents d typed by the following simple DTD:

DTD3=

```
<!DOCTYPE root[
<!ELEMENT root empty_element>
<!ELEMENT empty_element EMPTY>
```

and the query

```
Q6=     from x  in d/empty_element
        select x.value()
```

This query is correct. Indeed, the query path is valid with respect the type of d. However the type of its result is the empty type. In this case the type system will return the "yes" value for the valid parameter.

Due to the presence of union types in DTD, some further considerations about correctness have to be done.

Consider documents d typed by this DTD.

DTD4=

```
<!DOCTYPE root[
<!ELEMENT root (ele1 | ele2)>
<!ELEMENT ele1 #PCDATA >
<!ELEMENT ele2 #PCDATA >
]>
```

If we consider the query

```
Q6=     result[
        from x  in d/ele1
        select element[x.value()]]
```

even if a document d may not contain an ele1 element, the type system consider this query as correct, since the query path can be traversed on those valid documents that contain the required ele1 element. Of course, no values are bound to x in the case of absence of the ele1 element. Indeed, in this case Tequyla-TX returns the following type for the query:

```
<!DOCTYPE result[
<!ELEMENT result element?>
<!ELEMENT element #PCDATA >]>
```

that indicates that result content may be empty.

If we consider the query

```
Q7=     result[
        from x  in d/(ele1+ele3)
        select element[x.value()]]
```

the query is considered correct since in the alternative path there is at least one case (i.e., ele1) that matches the type of d. In this case, the type of Q6 remains the same as Q7.

Instead, the following query is not correct.

```
Q8=   result[
      from x  in d/(ele3+ele4)
      select element[x.value()]]
```

For this query, all the possible alternatives expressed in the path expression are not valid w.r.t. the document type.

Regarding the typing for Tequyla-TX textual operators, the most significant are queries returning sequences of match points, such as

```
result[Q contains $"word"$]
```

where Q is a generic query. The system consider this query as correct if Q is correct, and the result type is

```
<!DOCTYPE result[
<!ELEMENT result match_point *>
<!ELEMENT match_point (document, start,
            end, word)>
<!ELEMENT document #PCDATA>
<!ELEMENT start #PCDATA>
<!ELEMENT end #PCDATA>
<!ELEMENT word #PCDATA>]>
```

This result type can be useful to develop a visualization tool for the query result. Of course, the user can also navigate through the list of match points by means of a query, which includes the one above. Correctness of this query is checked w.r.t. the type above.

6. Tequyla-TX Query Algebra

Tequyla-TX queries are mapped into algebraic expressions before their execution. The starting point of the Tequyla-TX algebra (TTX algebra in the following) is the Xtasy query algebra, described in (Colazzo, Manghi, Sartiani, & Albano, 2001), which has been extended in order to support the textual operators of Tequyla-TX.

The key features of TTX algebra are the manipulation of *relational-like* intermediate structures (hence extending to XML common relational and OO optimization strategies), as well as the presence of *frontier* operators, which insulate the other ones from the technicalities of XML.

In the following subsections, the main features of TTX algebra will be described, as well as the translation of Tequyla-TX queries into algebraic expressions.

6.1 Data model and term language

XML documents are modeled as rooted, *node-labeled* trees. Internal nodes are labeled with constants, while leaves contain atomic values. Thus, XML documents are represented as terms conforming to the following grammar (very close to the term grammar of XDuce (Hosoya & Pierce, 1999)):

(1) t: := $t_1,\ldots,t_n$ | label[t] | @label[v_B] | v_B

(2) label ::= as defined by XML specifications

(3) v_B ::= Integer | String | Char | Boolean | …

Example 18. *Consider the XML fragment shown below:*

```
<book class = "OpSys">
   <author> Stuart Madnick </author>
   <author> John Donovan </author>
   <title> Operating Systems </title>
   <year> 1974 </year>
</book>
```

This fragment can be represented by the following term:

book[
@class["OpSys"],
author["Stuart Madnick"],
author["John Donovan"],
title["Operating Systems"],
year[1974]
]

6.2 Intermediate structure

Algebraic operators of TTX algebra manipulate relational-like structures, which contain the variable bindings collected during query evaluation: a variable can be bound to a tree, or to a whole forest. In order to apply useful optimization properties (e.g., vertical and horizontal decomposition of binding operators), these structures (called *Env* structures) are themselves represented as XML trees, as shown in Figure 4.

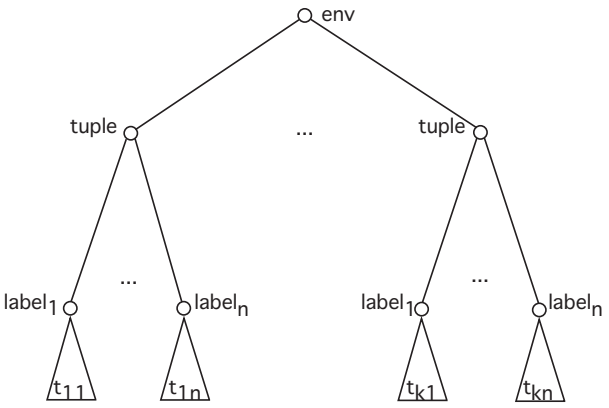

Figure 4. An intermediate structure

The *Env* structure is modeled as a rooted tree; each tuple element describes a binding tuple, where $label_i$ are variable names and t_{ji} the corresponding values.

The *Env* structure depicted above can also be represented by the following term:

e = env[
tuple[label$_1$[t$_{11}$],…,label$_n$[t$_{1n}$]],…,
tuple[label$_1$[t$_{k1}$],…,label$_n$[t$_{kn}$]]
]

Thus, an *Env* structure is basically an *environment* of variable bindings, which are organized into tuples. TTX algebra operators are defined on *unordered Env* structures; to preserve the ordering of XML trees, a *Sort* operation is applied during query results materialization. This allows the query optimizer to apply useful algebraic equivalences (e.g., reordering of join operations), which instead are not true on ordered structures. The same approach is employed in other XML query algebras, such as (Jagadish, Lakshmanan, Srivastava, & Thompson, 2001). For the sake of simplicity, unordered *Env* structures will be denoted by the following notation (similar to YAT (Cluet, Delobel, Siméon, & Smaga, 1998) *Tab* representation):

e = {[label$_1$:t$_{11}$, … , label$_n$:t$_{1n}$], … , [label$_1$:t$_{k1}$, … , label$_n$:t$_{kn}$]}

6.3 Algebra operators

TTX algebra operators can be divided into three classes: 'traditional' operators; *frontier* operators; textual operators.

6.3.1 Traditional and frontier operators

Traditional operators manipulate *Env* structures only, and perform quite common operations. They resemble very closely their relational or object-oriented counterparts; this allows the query optimizer to employ usual algebraic optimization strategies. This class contains *Map*, *Join*, *TupJoin*, *DJoin*, *Selection*, *Projection*, *GroupBy*, *Sort*, as well as *Union*, *Intersection*, and *Difference*.

Frontier operators insulate the other ones from the technicalities of XML, in particular, from the (possibly) deeply nested structure of XML documents. This class consists of two operators only: *path* and *return* (which are very close to YAT *bind* and *tree* operators (Cluet, Delobel, Siméon, & Smaga, 1998)).

path takes as input an XML forest and an *input filter*, which is a tree representation of XPath-like expressions, and it returns a corresponding environment. This *Env* structure is built up by traversing the input forest according to the input filter, and by performing the required variable bindings. The *path* operator, hence, expresses horizontal navigation, vertical navigation, as well as variable binding.

Example 19. *Consider the following XML document:*

```
<book class = "OpSys">
   <author> Stuart Madnick </author>
   <author> John Donovan </author>
   <title> Operating System </title>
   <year> 1974 </year>
</book>
```

Assume that we want to bind the title of the book to a variable t, and each author in turn to a variable a. We can use the input filter depicted in Figure 5.

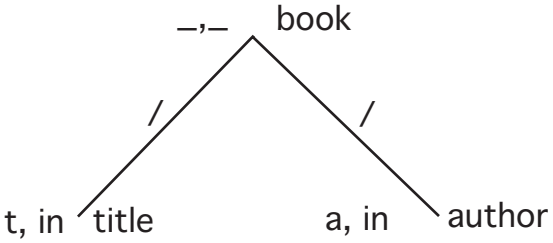

Figure 5. A simple input filter

This input filter is a tree, whose nodes are labeled with structural information, and whose edges are labeled with navigational operators (/ or //). Each node can also contain binding information (the variable to be bound and the binder).

The result of the application of this input filter is shown below.

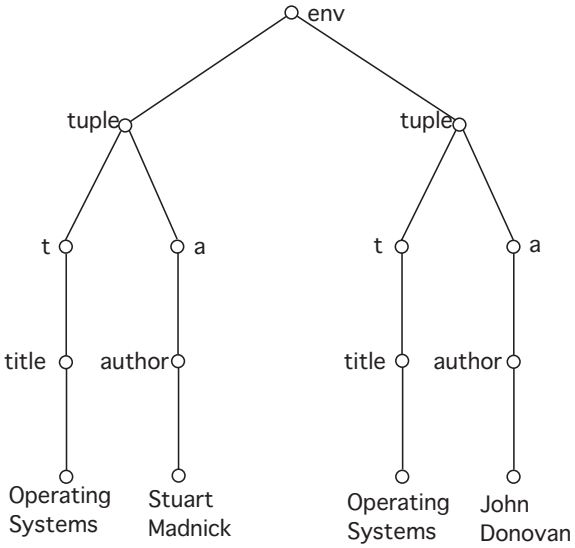

Figure 6. Application of an input filter

The *return* operator takes as input an *Env* structure and an *output filter*, and it builds a corresponding data model instance. This instance is build up by filling the XML skeleton, described by the output filter, with variable values taken from the *Env* structure: this substitution is performed once per each tuple contained in the *Env* structure, hence producing one skeleton instance per tuple. Output filters must satisfy the following grammar:

(1) OF ::= OF$_1$,…, OF$_n$ | label[OF] | @label[val] | val

(2) val ::= v$_B$ | var

Example 20. *Consider the Env structure built in the previous example, and assume that we want to produce a document containing the authors only. We can use the following return operator:*

return ($a) (e)

where the output filter is just the variable a.

The resulting document is shown below:

```
<author> Stuart Madnick </author>
<author> John Donovan </author>
```

6.3.2 Textual operators

The third class of algebraic operators contains the textual operators added to support text retrieval queries. The Xtasy query algebra has been extended with three main textual operators: *contains*, *containsword*, and *contains_start*. These operators, given a data model instance *t* and a string *s*, return the set of match points corresponding to the occurrences of *s* inside the text contained into *t*. They differ in the way occurrences are selected and search contexts are formed: *contains* searches for occurrences into a char-based representation of the text, while *containsword* performs this search on a word-based version of the text; *contains_start*, finally, mimics the behavior of its language counterpart, both on a char-based and on a word-based representation of the text, i.e., it searches for occurrences contained into an ancestor *t'* of *t* and whose starting point is contained into *t*.

contains: *DataModelInstance* ×
DataModelInstance ×
String -> Seq(Matchpoint)

contains(t)(t')(s) = $\{m_i\}^{i \in I}$, where *t* is the data model instance defining the search context, *t'* is a data model instance containing *t*, *s* is the string being searched, and m_i is a match point of *s* in *t*.

containsword: *DataModelInstance* ×
DataModelInstance ×
String -> Seq(Matchpoint)

containsword(t)(t')(w) = $\{m_i\}^{i \in I}$, where *t* is the data model instance defining the search context, *t'* is a data model instance containing *t*, *w* is the word being searched, and m_i is a match point of *w* in *t*.

contains_start: *(DataModelInstance* ×
DataModelInstance ×
String -> Seq(Matchpoint)) ×
DataModelInstance ×
DataModelInstance ×
String -> Seq(Matchpoint)

contains_start(f)(t)(t')(s) = $\{m_i\}^{i \in I}$, where *f* is the search function being used (*contains* or *containsword*), *t* is the data model instance containing the beginning of the string *s*, *t'* is the ancestor of *t* that should contain the occurrences of *s*, *s* is the string being searched, and m_i is a match point of *s* into *t'*.

The algebra also contains predicates for comparing match point positions: *before*, *after*, *beforechar*, and *afterchar*.

before takes as input a match point *m*, a word *w*, a data model instance *t*, and a positive integer *dist*; it generates the set of match points of the word *w* into *t*, and compares their positions with the one of *m*. This predicate returns true if there exists a match point of *w*, following *m* by *dist* words.

after takes the same input as *before*, and it returns true if there exists a match point of *w* preceding *m* by *dist* words.

beforechar and *afterchar* are defined in a similar way, with the only difference that the comparison is performed on a char basis.

before: *Matchpoint* ×
String ×
DataModelInstance ×
DataModelInstance ×
Integer -> Boolean

before(m)(w)(t)(t')(dist) = bool, where *m* is a match point, *w* is a word, *t* is the data model instance defining the search context, *t'* is the data model instance containing *t*, *dist* is the required distance between *m* and *w*, and *bool* is a Boolean value.

after: *Matchpoint* ×
String ×
DataModelInstance ×
DataModelInstance ×
Integer -> Boolean

after(m)(w)(t)(t')(dist) = bool, where *m* is a match point, *w* is a word, *t* is the data model instance defining the search context, *t'* is the data model instance containing *t*, *dist* is the required distance between *w* and *m*, and *bool* is a Boolean value.

beforechar: *Matchpoint* ×
String ×
DataModelInstance ×
DataModelInstance ×
Integer -> Boolean

beforechar(m)(s)(t)(t')(dist) = bool, where *m* is a match point, *s* is a string, *t* is the data model instance defining the search context, *t'* is the data model instance containing *t*, *dist* is the required char distance between *m* and *s*, and *bool* is a Boolean value.

afterchar: *Matchpoint* ×
String ×
DataModelInstance ×
DataModelInstance ×
Integer -> Boolean

afterchar(m)(s)(t)(t')(dist) = bool, where *m* is a match point, *s* is a string, *t* is the data model instance defining the search context, *t'* is the data model instance containing *t*, *dist* is the required char distance between *s* and *m*, and *bool* is a Boolean value.

The definition of these operators and predicates requires the introduction of some support functions, as well as functions and operators for string manipulation.

A string *s* is modeled as an ordered sequence of chars:

$S \equiv <char(s)(1), \ldots, char(s)(length(s))>$, where *char(s)(k)* extracts the k-th char from a string *s*, and *length(s)* returns the length of a string *s*.

To extract arbitrary sub-strings from strings the *substring* operator is provided:

$substring(s)(i)(j) = <char(s)(i), \ldots, char(s)(j)>$, *where* $i,j \leq length(s)$.

In order to perform textual searches, operators for building up search contexts are needed; the TTX algebra supports two context-forming operators: *textof* and *wordof*. The former, given a data model instance, returns a sequence of search contexts, endowed with their offset into the document; the latter returns a sequence of search contexts, each one in turn divided into words, in order to allow the evaluation of word-based operations. These operators are not part and parcel of the query algebra, and they can be adapted by the database administrator to the application requirements: the TTX algebra only imposes typing constraints on these operators.

textof: DataModelInstance -> Seq(Int × String)

$textof(t) = \{(i^j,s^j)\}^{j \in J}$, where i^j is the offset of the j-th context search w.r.t *t* and s^j is the j-th context search.

wordof: DataModelInstance ->
Seq(Integer × Integer ×
Seq(Integer × Integer × String))

$wordof(t) = \{(i_1^j,i_2^j,\{(i_3^k,i_4^k,s^k)\}^{k \in K_j})\}^{j \in J}$, where i_1^j is the offset of the j-th search context w.r.t to *t*, i_2^j is the word offset of the j-th search context w.r.t to *t*, i_3^k is the position of the k-th word inside the current search context, and i_4^k is the offset of the k-th word inside the current search context.

A formal definition of these textual operators can be found in Appendix B.

6.4 Mapping Tequyla-TX Queries

This section describes the mapping of Tequyla-TX queries into algebraic expressions by showing the translation of a Tequyla-TX query, containing both textual predicates and operators, as well as structural constraints.

Consider the following Tequyla-TX query:

```
from dictionary =
   document("www.cribecu.sns.it/crusca.xml"),
   e in dictionary//entry,
   s in e//sense,
   occur in s contains $"signific."$
where before(occur, "pass.", 1)
select occur
```

This query returns all the occurrences of the word 'signific.', inside sense elements, that precede 'pass' by one word.

The translation process consists of two phases. During the first phase, the query compiler applies to the parse tree of the query a syntactical preprocessing, consisting of three steps. First of all, path expressions, which occur free in any clause, are bound to variables, and the corresponding binders are introduced in the *from* clause. Second, nested queries occurring in any clause are bound to variables, and the corresponding binders are inserted into the *from* clause.

These transformations produce a query without nested queries in any clause but the *from* clause, and are required in order to exploit the algebraic rewritings described in (Cluet & Moerkotte, 1993). Since the sample query does not contain nested queries at all nor free XPath patterns, these two steps leave the query untouched.

The third transformation applied during this phase is the construction of input and output filters. While the construction of output filters is straightforward, building input filters requires to transform each path into a *filter-like* expression, and then to merge together paths referring to the same persistence root. This transformation is not applied to textual operators, which are handled at the algebraic level only.

Referring to our sample query, the result of this preprocessing phase is the following abstract query:

```
from (input_filter, document("…")),
    occur in s contains $"signific."$
where before(occur, "pass.", 1)
select output_filter,
```

where *output_filter = $occur* and
input_filter =
(_,_,_)dictionary[(//,e,in)entry[(//,s,in)sense[ϕ]]].

The second phase of the translation process takes as input an abstract query parse tree, and it returns an algebraic expression. This expression is built up by translating *where* clauses into *Selection* operators, *select* clauses into *return* operators, and *from* clauses into *path* operators.

These operators are then combined in a compositional fashion, or by using *DJoin*, *Join*, and *Union* operators.

Referring to the sample query, the query compiler maps the first *from* clause into a *path* operator *path(input_filter)(db)*, where *db* denotes the document being queried. Then, the query compiler translates the second *from* clause (*occur in s contains $"signific."$*) into an applications of the *containsword* operator to each instance of the variable *s*; the binding of the variable *occur* is performed by means of a *Map* operation: *Map([occur:x])(containsword($s)(db)("signific."))*.

These two subexpressions are then combined by using *DJoin* operation, as shown in Figure 7.

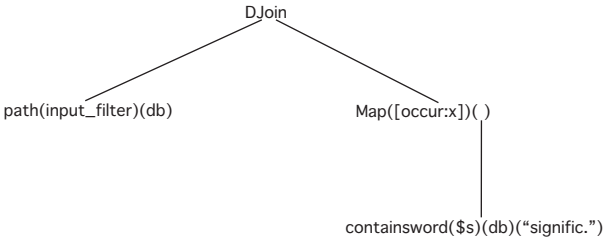

Figure 7. Partial query plan

The result of the *DJoin* operation is an *Env* structure, containing bindings for variables *dictionary*, *e*, *s*, and *occur*.

The query compiler, then, applies to this *Env* structure a *Selection* operation, in order to discard the tuples that do not satisfy the predicate *before(occur,"pass.",1)*: *σ(before($occur)("pass.")($s)(db)(1))(...)*.

Finally, the query compiler builds a *return* operation, which, given the previously built output filter and the current *Env* structure, produces an XML document. The resulting algebraic expression is shown in Figure 8.

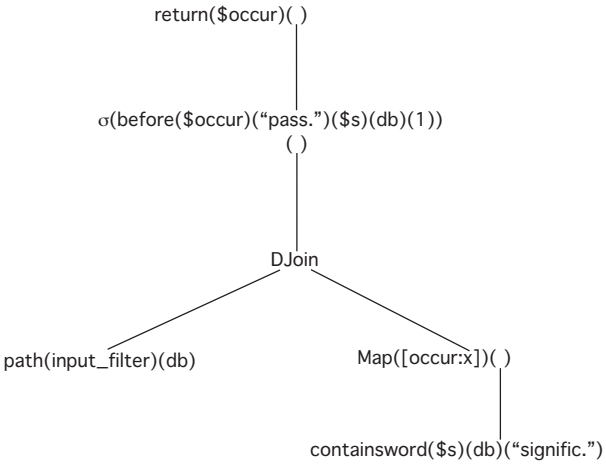

Figure 8. Query plan

7. Conclusions

A statically typed query language for XML documents has been presented. This language is able to perform both *word-based* and *char-based* text searches, in order to support the requirements of literary applications.

A prototype implementation of Tequyla-TX is *in-progress* to experiment with the integration of database access structures and those provided by the underlying search engine, and with optimization techniques for XML data.

8. REFERENCES

Abiteboul, S., Quass, D., McHugh, J., Widom, J., & Wiener, J. L. (1997). The Lorel Query Language for Semistuctured Data. *Journal of Digital Libraries, 1*(1), 68-88.

Albano, A., Colazzo, D., Ghelli, G., Manghi, P., & Sartiani, C. (2000a, July). *A Type System for Querying XML Documents.* Paper presented at the ACM SIGIR 2000 Workshop On XML and Information Retrieval, Athens, Greece.

Albano, A., Colazzo, D., Ghelli, G., Manghi, P., & Sartiani, C. (2000b). *A Typed Query Language for XML Documents*.Unpublished manuscript.

Amadio, R. M., & Cardelli, L. (1993). Subtyping recursive types. *ACM Transactions on Programming Languages and Systems, 15*(4), 575-631.

Chamberlin, D., Fankhauser, P., Marchiori, M., & Robie, J. (2000). *XML Query Requirements*.: World Wide Web Consortium.

Chamberlin, D., Florescu, D., Robie, J., Siméon, J., & Stefanescu, M. (2001). *XQuery: An XML Query Language*.: World Wide Web Consortium.

Chamberlin, D., Robie, J., & Florescu, D. (2000). *Quilt: An XML query language for heterogeneous data sources.* Paper presented at the WebDB'2000, Dallas, Texas.

Clark, J., & DeRose, S. (1999). *XML Path Language (XPath) Version 1.0*.: World Wide Web Consortium.

Cluet, S., Delobel, C., Siméon, J., & Smaga, K. (1998, June). *Your mediators need data conversion!* Paper presented at the ACM SIGMOD International Conference on Management of Data (SIGMOD-98), New York.

Cluet, S., & Moerkotte, G. (1993, 30 August - 1 September 1993). *Nested Queries in Onject Bases.* Paper presented at the Fourth International Workshop on Database Programming Languages – Object Models and Languages, Manhattan, New York City, USA.

Colazzo, D., & Ghelli, G. (1999). *Subtyping recursive types in Kernel Fun.* Paper presented at the 14th Annual IEEE Symposium on Logic in Computer Science (LICS), Trento, Italy.

Colazzo, D., Manghi, P., Sartiani, C., & Albano, A. (2001). *Yet Another Query Algebra for XML Data*. Submitted Paper. Available at http://www.di.unipi.it/~sartiani/papers/algebra.pdf.

Corti, F., Lombardini, D., & Paoli, M. (2000). *TResy: A Text Retrieval System for SGML/XML documents*. Pisa: Centro Ricerche Infomatiche Beni Culturali - Scuola Normale Superiore.

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., & Suciu, D. (1998). *XMLQL: A Query Language for XML*.: World Wide Web Consortium.

Fankhauser, P., Fernandez, M., Malhotra, A., Rys, M., Siméon, J., & Wadler, P. (2001). *The XML Query Algebra*.: World Wide Web Consortium.

Fernandez, M., & Robie, J. (2000). *XML Query Data Model*.: World Wide Web Consortium.

Fernandez, M., Siméon, J., & Wadler, P. (1999). *XML Query Languages: Experiences and Exemplars*.Unpublished manuscript.

Ferragina, P., & Grossi, R. (1999). The String B-tree: A New Data Structure for String Search in External Memory and Its Applications. *Journal of the ACM, 46*(2), 236-280.

Florescu, D., Kossmann, D., & Manolescu, I. (2000). *Integrating keyword search into XML query processing*. Paper presented at the WWW9, Amsterdam.

Fuhr, N. (2000, July). *XIRQL – An Extension of XQL for Information Retrieval.* Paper presented at the ACM SIGIR 2000 Workshop On XML and Information Retrieval, Athens, Greece.

Hosoya, H., & Pierce, B. C. (1999). *XDuce: An XML Processing Language* (Preliminary Report). Philadelphia: Department of CIS, University of Pennsylvania.

ISO. (1986). ISO 8879. Information Processing – Text and Office Systems - Standard Generalized Markup Language (SGML).

Jagadish, H. V., Lakshmanan, L. V. S., Srivastava, D., & Thompson, K. (2001). *TAX: A Tree Algebra for XML*.Unpublished manuscript. Available at http://www.eecs.umich.edu/db/timber/tax_full.ps

Milo, T., Suciu, D., & Vianu, V. (2000, May 15-17, 2000). *Typechecking for XML Transformers.* Paper presented at the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Dallas, Texas, USA.

Navarro, G. (1995). *A language for queries on structure and contents of textual database.* Unpublished Master's Thesis, University of Chile.

Neumann, A. (2000). *Fxgrep: the Functional XML Querying Tool*.: Universität Trier.

Papakonstantinou, Y., Widom, J., & Molina, H. G. (1996). *Object exchange across heterogeneous information sources.* Paper presented at the IEEE International Conference on Data Engineering, Birmingham, England.

Robie, J., Derksen, E., Fankhauser, P., Howland, E., Huck, G., Macherius, I., Murata, M., Resnick, M., & Schning, H. (1999). *XQL (XML Query Language)*.

Robie, J., Lapp, J., Schach, D., Hyman, M., & Marsh, J. (1998). *XML Query Language (XQL)*.

Salminen, A., & Tompa, F. (1994). PAT expressions: an algebra for text search. *Acta Linguistica Hungarica*, 277-306.

TEI. (1998). TEI: Text Encoding Initiative.

A. Tequyla-TX Grammar

Queries:

Q ::=

()	empty query
| *m*(*a*) [*Q*]	element constructor
| Q_1, Q_2	forest constructor
| *value*	base value of type B
| **from x** =Q_1 **where** *W* **select** Q_2	binding
| **from** *x* **in** Q_1 **where** *W* **select** Q_2	iteration
| *x*	variables
| *QP*	path selection
| *Q optext union occpos*	textual operator application

Textual operator

optext ::=

contains *cs start cstring end*	contains the string
| **contains-start** *cs start cstring end*	contains the beginning of the string

Union of textual operator

union ::= ε

\| ∪ *optext union*

Occurrences position

occpos ::=

ε	all occurences
| [*positions*]	occurrences in *positions*

Case sensitive operator

cs ::= ε | **case - sensitive**

Word beginning operator

start ::= ε | $

Word ending operator

end ::= ε | $

String sequences

cstring ::=

v_B

\| *value* ? *cstring*

\| *value* * *cstring*

Element positions

positions ::=

integer, *positions*

\| **integer** - **integer** , *positions*

Attributes list

a ::=

( )

| $m = v_B$, *a*	base value

Conditions

W ::=

W_1 **and** W_2	logical conjuction
| W_1 **or** W_2	disjunction
| **not** *W*	negation
| *value compop value*	value comparision
| *value optext value*	textual comparison
| **exists** (*Q*)	emptiness condition
| $\mathbf{f^n}$ (Q_1, ...,Q_n)	general boolean function

	with n parameters

Value extraction

value ::=

Q.**value** ()	simple element and attribute value extraction
| *Q*.**textof** ()	string conversion and normalization
| *Q*.**prefix** ()	namespace component
| *Q*.**local** ()	local label component
| v_B	base value

Comparison operator

compop ::= < | <= | > | >= | = | ! =

Path

P ::=

ε	empty path
| / *ls*	direct descendants
| //*ls*	descendants
| $P_1 P_2$	concatenation
| $P_1 + P_2$	alternative

Labels selector

ls ::=

m	elements labelled *m*
| *m*[*positions*]	elements labelled *m* in *positions*
| @*m*	attributes labelled *m*
| *	all elements

Labels

m ::=

p : **l**	namespace and label
| **l**	label

A.1 Some external operators

fun ::=

before
| **after**
| **both**
| **beforchar**
| **afterchar**
| **distance**
| **times**

B. TTX Algebra Operators

B.1 String manipulation operators

S ≡ <*char(s)(1), ... , char(s)(length(s))*>, where *char(s)(k)* extracts the k-th char from a string *s*, and *length(s)* returns the length of a string *s*.

substring(s)(i)(j) = <char(s)(i), ... , char(s)(j)>, where i,j ≤ length(s).

B.2 Support operators

offset: DataModelInstance × DataModelInstance -> Integer

offset(t)(t') = offset of t in t'

wordoffset: DataModelInstance × DataModelInstance -> Integer

wordoffset(t)(t') = word offset of t w.r.t t'

B.3 Search context forming operators

textof: DataModelInstance -> Seq(Int × String)

textof(t) = <(int,string)>, where *int* is the offset of each context search w.r.t *t* and *string* is the context search.

wordof: DataModelInstance -> Seq(Int × Int × Seq(Int × Int × String))

wordof(t) = <(int_1*,*int_2*,<(*int_3*,*int_4*,string)>)>*, where int_1 is the offset of the search context w.r.t to *t*, int_2 is the word offset of the search context w.r.t to *t*, int_3 is the position of the word inside the current search context, and int_4 is the char offset of the word inside the current search context.

B.4 String and word comparison operators

like: Seq(Integer× String) × String -> Seq(Integer × Integer)

like(ts)(s) = {(i+first(ts),j+first(ts)) | i ≤ length(snd(ts)) and j ≤ length(snd(ts)) and j-i+1 = length(s) and substring(snd(ts))(i)(j) = s}

likeword: Seq(Integert × Integer × Seq(Integer × Integer × String)) × String -> Seq(Integer × Integer × Integer)

likeword(ws)(s) = { (i+first(ws[z]), j+first(ws[z]), k +snd(ws[z]) | (((ws[z])[3])[z'])[3] = s and i = (((ws[z])[3])[z'])[2] and j =i + length(s) and k =(((ws[z])[3])[z'])[1]}

B.5 Textual operators and predicates

contains: *DataModelInstance* ×
DataModelInstance ×
String -> Seq(Matchpoint)

contains (t) (t')(s) = Map(matchpoint[document[uri(t')], start[first(x)+ offset(t)(t')],end[second(x) + offset(t)(t')], word[0]]) ($\bigcup_{ts_i \in textof(t)}$ *like*(ts_i)(s))

containsword: *DataModelInstance* ×
DataModelInstance ×
String -> Seq(Matchpoint)

containsword (t) (t')(s) = Map(matchpoint[document[uri(t')], start[first(x)+ offset(t)(t')], end[second(x) + offset(t)(t')], word[x[3] + wordoffset(t)(t')]]) ($\bigcup_{ws_i \in wordof(t)}$ *likeword*(ws_i)(s))

contains_start: *(DataModelInstance* ×
DataModelInstance ×
String -> Seq(Matchpoint)) ×
DataModelInstance ×
DataModelInstance ×
String -> Seq(Matchpoint)

contains_start(f)(t)(t')(s) = {o | o in f(t')(t')(s) and Pred(t)(t')(o)}, where:

Pred(o) = o.matchpoint.start >= offset(t)(t') and o.matchpoint.start <= offset(t)(t') + $\sum_{ts_i \in textof(t)}$ *length*(ts_i[2])

and

f ∈ *{contains, containsword}*

before: *Matchpoint* ×
String ×
DataModelInstance ×
DataModelInstance ×
Integer -> Boolean

before(m)(s)(t)(t')(dist) = ∃*occ*∈*containsword(t)(t')(s): occ.word – m.word >= dist*

after: *Matchpoint* ×
String ×
DataModelInstance ×
DataModelInstance ×
Integer -> Boolean

after(m)(s)(t)(t')(dist) = ∃*occ*∈*containsword(t)(t')(s): m.word – occ.word >= dist*

beforechar: *Matchpoint* ×
String ×
DataModelInstance ×
DataModelInstance ×
Integer -> Boolean

beforechar(m)(s)(t)(t')(dist) = ∃*occ*∈*contains(t)(t')(s): occ.end – m.end >= dist*

afterchar: *Matchpoint* ×
String ×
DataModelInstance ×
DataModelInstance ×
Integer -> Boolean

afterchar(m)(s)(t)(t')(dist) = ∃*occ*∈*contains(t)(t')(s): m.end – occ.end >= dist*

C. Tequyla-TX Type System

The formalization of the type system we present refers to a core part of Tequyla-TX. We consider only the main mechanisms of the language, as the extension to the full language does not present significant technical problems.

In particular, in this version of the type system we concentrate on checking that path specifications in the *from* clauses of queries are valid with respect to the type of queried data and types of nested queries. Accordingly, we consider queries without the *where* clause. Moreover, we consider a restricted path language, where only the main constructs are present, namely single step */ls*, structural recursion //ls, and alternative (union) paths P1 + P2.

In the following, we present the type language, data language, data typing rules, and query typing rules. For the syntax of queries and patterns see Appendix A.

C.1 Types

T ::=	()	empty sequence type
	B	base type
	T, T	sequence type
	T + T	union type
	m(A)[T]	element type
	X	type variable

B :: =	Char	
	Integer	
	…..	

C.2 Data

D ::=	()	empty document
	m(a) [D]	element constructor
	D, D	sequence constructor
	v_B	base value of type B

C.3 Attributes list

a ::=	( )	
	m = v_B , a	

C.4 Type Environment

E ::=	()	
	x: T, E	variable typing
	X=T, E	type definition

C.5 Data Typing

E |-- *D*: T, data *D* has type T according to the type environment E.

(Empty)

$$E \vdash () : ()$$

(BaseValue)

$$E \vdash v_B : B$$

(ValueElement)

$$\frac{E \vdash a : A \quad E \vdash D : T}{E \vdash m(a)\,[D] : m(A)[T]}$$

(Seq)

$$\frac{E \vdash D_1 : T1 \quad E \vdash D_2 : T2}{E \vdash D_1, D_2 : T_1, T_2}$$

(UnionType)

$$\frac{E \vdash D : T_1 \quad \text{or} \quad E \vdash D : T_2}{E \vdash D : T_1 + T_2}$$

(AttrType)

$$\frac{E \vdash a : A \quad E \vdash v_B : B}{E \vdash m = v_B\,,\, a : m{:}B, A}$$

C.6 Query Typing

C.6.1 Type Judgements

E |-- *Q* : (T, yes), with respect to the type declarations in E for *Q*, *Q* has type T and patterns in *Q* are valid (Q is correct).

Observe that correct queries may have *empty type*, which we shall define in the following.

E |-- *Q* : (T, no), with respect to the type declaration in E for *Q*, patterns in *Q* are not valid (Q is incorrect). In this case T is an empty type.

C.6.2 Types Of Empty Documents

A type T is a type of empty documents (TOED in the following) if all documents D such that E |--*D* : T are empty documents, such as D=() or D=(),(),(). By definition, the type () is a TOED. However, due to the presence of union, sequence and recursive types, other more complex types may respect this definition: for example, the type $X = () + (), X$ is a TOED too.

TOEDs could be defined as all types equivalent to () according to a given set of equivalence rules. For brevity, however, we only give a characterization of TOEDs.

The definition of TOEDs requires the operator $Reach_E(S)$ which, given a set of type definitions E and a set of types S, returns the set of types *reachable* from types in S by following the definitions in E (we assume that E defines all variables occurring in types in E and S):

$Reach_E(\{ () \}) = \{ () \}$

$Reach_E(Ø) = Ø$

$Reach_E(\{ X \}) = Reach_E(\{ E(X) \}) \cup \{ X \}$

$Reach_E(\{ m(a)[T] \}) = Reach_E(\{ T \}) \cup \{ m(a)[T] \}$

$Reach_E(\{ T,U \}) = Reach_E(\{ T \}) \cup Reach_E(\{ U \}) \cup \{ T,U \}$

$Reach_E(\{ T+U \}) = Reach_E(\{ T \}) \cup Reach_E(\{ U \}) \cup \{ T+U \}$

$Reach_E(\{ T \} \cup S) = Reach_E(S) \cup Reach_E(T)$

$Reach_E$ is defined as the minimum funtion (w.r.t. the point wise ordering on functions) that satisfies the above equations.

We say that, according to E, T is a TOED if and only if each type in $Reach_E(\{ T \})$ is a type defined by the following grammar:

Empty ::= ()
| Empty + Empty
| Empty, Empty
| X.

Our notion of query correctness guarantees that, whenever

E |-- Q : (T, yes)

and T is not a TOED, the set of documents returned by Q is not empty, and each data *D* returned by *Q* is of type T, i.e. E |-- D : T.

C.6.3 Trails

To type a query

from x **in** Q_1 **select** Q_2

our system proceed as follows. It first finds the type of Q_1, say T_1, and then it makes the following case distinction on T_1.

If T_1 is an element, base or empty sequence type then this type is assigned to x and this information is used to determine the type of Q.

If T_1 is a union T_2+T_3 or a product T_2,T_3 then the systems computes the types of the two queries

from x **in** y_1 **select** Q_2

from x **in** y_2 **select** Q_2

where y_1 and y_2 are respectively assigned to have types T_2 and T_3. The two computed types are then opportunely recombined to have the type of the initial query.

The remaining case is when T_1 is a recursion variable X that is defined in the current type environment E. In this case, the type of the query is given by the type of

from x **in** *y* **select** Q_2

where y is assigned to have type E(X), the type that defines X. Due to the presence of recursive types, this process may not terminate. To avoid this, for *from-in* queries we consider a particular judgment

E; Σ |-- **from** x **in** Q_1 **select** Q_2 : (T, valid)

Note that a second environment Σ is present. We call it *trail* environment and its task is to keep track of information that can be useful to avoid loops. More precisely, if Q_1 has type X, a triple (x, X, Z) is added to the trail. Such triple keeps track of the fact that the variable x has iterated on the type X and assumes that the iteration query has type Z. The actual definition of Z will be computed by the subsequent typing steps, where, as already stated, the iteration according to x continues on the definition of X in the current type environment. If the type X is encountered once again in the same iteration on x, the iteration stops and returns Z, associated to x and X in the triple (x, X, Z) contained in the trail environment. In this case, the value returned for the *valid* parameter is "no" so that this typing cannot determine the entire typing of the initial query. Observe that, had iteration continued on the definition of X, it would have infinitely repeated the same steps following the first iteration of x on X.

Regarding queries Q P, the system behaves in a similar way, that is by case distinction on the type of Q. Essentially, the type of QP is computed by traversing the type of Q according to the path P. In particular, for queries QP, the system may loop for two reasons. The first one is essentially the same of the previous case: iteration over a recursive type. The second reason is that P may contain structural recursion and, if the type of Q is a recursive type then the system may loop in traversing this type according to P. Think, for example, of a query

Q//a

where Q has type X= () + a[X].

To avoid looping, we consider trails also for typing queries QP, with the only distinction that in this case triples (P,X,Z) are added to the trail. A triple (P,X,Z) indicates that the type of Q is X and that the type of QP is assumed to be Z.

Trails are defined as follows:

Σ ::= () | (x, X, Z), Σ | (P,X, Z), Σ

C.6.4 Query Typing Rules

The following set of rules defines our query typing system. In the following, we consider *yes* and *no* as boolean values *true* and *false*.

(TypeEmpty)

E; Σ |-- () : ((); yes)

(TypeValueElementMatch)

$$\frac{E \text{ |-- } a : A \qquad E \text{|-- } Q : (T; yes)}{E \text{|-- } m(a)\,[Q] \; : (\, m(A)[T]; yes)}$$

(TypeValueElementNotMatch)

$$\frac{E \vdash a : A \qquad E \vdash Q : (T; no)}{E \vdash m(a)\,[Q] : (m(A)\,[T]\ ; no)}$$

(TypeValueValExtr)

$$\frac{E \vdash Q : (m(A)[B]; yes)}{E \vdash Q.\textbf{value}\ () : (B; yes)}$$

(TypeValueTextOf)

$$\frac{E \vdash Q : (T;\ yes)}{E \vdash Q.\textbf{textof}\ () : (String; yes)}$$

(TypeValuePrefix)

$$\frac{E \vdash Q : (m(A)[B]; yes)}{E \vdash Q.\textbf{prefix}\ () : (String; yes)}$$

(TypeValueLocal)

$$\frac{E \vdash Q : (m(A)[B]; yes)}{E \vdash Q.\textbf{local}\ () : (String;\ yes)}$$

(TypeValueBaseValue)

$$E \vdash v_B : (B;\ yes)$$

(TypeVar)

$$E, x:T, E' \vdash x : (T; yes)$$

(TypeSeqMatch)

$$\frac{E \vdash Q_1 : (T1;\ valid_1) \qquad E \vdash Q_2 : (T_2; valid_2)}{E \vdash Q_1, Q_2 : (T_1,T_2; valid_1\ \text{And}\ valid_2)}$$

(TypeFromEq)

$$\frac{E \vdash Q_1: (T_1; valid_1) \qquad E, x:T \vdash Q_2: (T_2; valid_2)}{E \vdash \textbf{from}\ x = Q_1\ \textbf{select}\ Q_2 : (T_2; valid_1\ \text{And}\ valid_2)}$$

For iteration performed by queries of the form

from x in Q_1 **select** Q_2

we consider the following rules, which distinguish on the type of Q_1 and make use of trails to guarantee finite proofs in the cases that the type of Q_1 is recursive.

(TypeFromIn)

$$\frac{E \vdash Q_1: (T_1; valid_1) \qquad E, y:T_1; () \vdash \textbf{from}\ x\ \text{in}\ y\ \textbf{select}\ Q_2 : (T_2; valid_2)}{E \vdash \textbf{from}\ x\ \text{in}\ Q_1\ \textbf{select}\ Q_2 : (T_2; valid_1\ \text{And}\ valid_2)}$$

(TypeFromInUnitTOED)

$$\frac{E \vdash Q_1: ((); valid)}{E; \Sigma \vdash \textbf{from}\ x\ \text{in}\ Q_1\ \textbf{select}\ Q_2 : ((); valid)}$$

(TypeFromInUnit)

$$\frac{E \vdash Q_1: (T; valid_1) \qquad T = B \text{ or } T = m(A)[T'] \qquad E, x:T \vdash Q_2: (T_1; valid_2)}{E; \Sigma \vdash \textbf{from}\ x\ \text{in}\ Q_1\ \textbf{select}\ Q_2 : (T_1; valid_1\ \text{And}\ valid_2)}$$

(TypeFromInSeq)

$$\frac{E \vdash Q_1: (T,U; valid_0) \qquad E, y:T; \Sigma \vdash \textbf{from}\ x\ \text{in}\ y\ \textbf{select}\ Q_2: (T_1; valid_1) \qquad E, y:U; \Sigma \vdash \textbf{from}\ x\ \text{in}\ y\ \textbf{select}\ Q_2: (T_2; valid_2) \qquad valid = valid_0\ \text{And}\ (valid_1\ \text{Or}\ valid_2)}{}$$

E ; Σ|-- **from x** in Q_1 **select** Q_2 : (T_1,T_2; valid)

(TypeFromInUnion)

E|-- Q_1: (T +U; $valid_0$)
E, y:T; Σ |-- **from x** in y **select** Q_2: (T_1; $valid_1$)
E, y:U; Σ |-- **from x** in y **select** Q_2: (T_2; $valid_2$)
valid = $valid_0$ And ($valid_1$ Or $valid_2$)

E; Σ |-- **from x** in Q_1 **select** Q_2 : (T_1+T_2; valid)

(TypeFromInVarUnfMatch)

E |-- Q_1: (X; $valid_1$)
(x, X, _) ∉Σ
E, y: E(X); Σ, (x, X, Z) |--
from x in y **select** Q_2: (T_1; $valid_2$)

E , Z=T_1; Σ |-- **from x** in Q_1 **select** Q_2 : (Z; $valid_1$ And $valid_2$)

(TypeFromInVarEnd)

E |-- Q_1: (X; valid)
(x, X, Z) ∈Σ

E; Σ |-- **from** x **in** Q_1 **select** Q_2 : (Z; no)

C.6.5 Path Rules

In the following rules, the auxiliary path syntax |*ls p* is used, for typing *Q /ls p*: given a sequence of elements, |*m* returns all elements tagged as *m*, while |*@a* returns all element attributes named as *a*.

(Path)

E|-- Q: (T_1; $valid_1$)
E, y: T; () |-- y P: (T_2; $valid_2$)

E |-- Q P : (T_2; $valid_1$ And $valid_2$)

(PathEmpty)

E |-- Q: (T; valid)

E; Σ |-- Q ε : (T; valid)

(PathVarUnf)

E |-- Q : (X; $valid_1$)
(P,Z,X) ∉Σ
E, y: E(X); Σ, (P,X,Z) |-- yP : (T_1; $valid_2$)

E, Z=T1; Σ |-- Q P : (Z; $valid_1$ And $valid_2$)

(PathVarEnd)

E|-- Q : (X; valid) (P, X,Z) ∈Σ

E; Σ |-- Q P : (Z; no)

(PathDeepMatch)

E|-- Q : (m(A)[T]; $valid_0$)
E; () |-- Q|lsP : (T_1; $valid_1$)
E, z: T; Σ |-- z//ls P : (T_2; $valid_2$)
valid = $valid_0$ And ($valid_1$ Or $valid_2$)

E; Σ |-- Q//ls P : (T_1,T_2; valid)

(PathDeepNotMatch)

E |-- Q : (T; valid)
T = B or T = ()

E; Σ |-- Q //ls P : ((); no)

(PathSingleMatchElAux)

$$\frac{E \vdash Q : (m(A)[T]; valid_1) \quad ls = m \text{ or } ls = * \quad E, z: T; () \vdash z \mid ls\ P : (T_1; valid_2)}{E; \Sigma \vdash Q \mid ls\ P : (T_1; valid_1 \text{ And } valid_2)}$$

(PathSingleMatchAttrAux)

$$\frac{E \vdash Q : (T; valid_1) \quad T = m'(A)[T'] \text{ or } T = B \text{ or } T = () \quad E, z: T; () \vdash z \mid @m\ P : (T_1; valid_2)}{E; \Sigma \vdash Q \mid @m\ P : (T_1; valid_1 \text{ And } valid_2)}$$

(PathSingleMatchEnd)

$$\frac{E \vdash Q : (m(A)[T]; valid_1) \quad ls = m \text{ or } ls = *}{E; \Sigma \vdash Q \mid ls : (m(A)[T];; valid_1 \text{ And } valid_2)}$$

(PathSingleMatch)

$$\frac{E \vdash Q : (m'(A)[T]; valid_1) \quad ls = m \text{ or } ls = * \quad E, z: T; () \vdash z\ P : (T_1; valid_2)}{E; \Sigma \vdash Q \mid ls\ P : (T_1; valid_1 \text{ And } valid_2)}$$

(PathSingleLableNotMatch)

$$\frac{E \vdash Q : (T; valid) \quad T = m'(A)[T'] \text{ and } m \neq m' \text{ or } T = B \text{ or } T = | ()}{E; \Sigma \vdash Q \mid m\ P : ((); no)}$$

(PathSingleStarNotMatch)

$$\frac{E \vdash Q : (T; valid) \quad T = B \text{ or } T = ()}{E; \Sigma \vdash Q \mid *\ P : ((); no)}$$

(PathSingleAttrMatch)

$$\frac{E \vdash Q : (m'(A)[T]; valid_1) \quad A = A', (m: B), A'' \quad E, z: B; \Sigma \vdash z\ P : (T; valid_2)}{E; \Sigma \vdash Q \mid @m\ P : (T; valid_1 \text{ And } valid_2)}$$

(PathSingleAttrNotMatch)

$$\frac{E \vdash Q : (T; valid) \quad T = m'(A)[T'] \text{ and } (m: B) \text{ not in } A \text{ or } T = B \text{ or } T = ()}{E; \Sigma \vdash Q \mid @m\ P : ((); no)}$$

(PathTypeUnion)

$$\frac{E \vdash Q : (T + U; valid_0) \quad E, z: T; \Sigma \vdash z\ P : (T_1; valid_1) \quad E, z: U; \Sigma \vdash z\ P : (U_1; valid_2) \quad valid = valid_0 \text{ And } (valid_1 \text{ Or } valid_2)}{E; \Sigma \vdash Q\ P : (T_1 + U_1; valid)}$$

(PathProductType)

$$\frac{E \vdash Q : (T, U; valid_0) \quad E, z: T; \Sigma \vdash z\ P : (T_1; valid_1) \quad E, z: U; \Sigma \vdash z\ P : (U_1; valid_2) \quad valid = valid_0 \text{ And } (valid_1 \text{ Or } valid_2)}{E; \Sigma \vdash Q\ P : (T_1, U_1; valid)}$$

(PathUnion)

$$\frac{E \vdash Q : (T; valid_0) \quad E, z: T; \Sigma \vdash z\ P_1\ P : (T_1; valid_1) \quad E, z: T; \Sigma \vdash z\ P_2\ P : (T_2; valid_2) \quad valid = valid_0 \text{ And } (valid_1 \text{ Or } valid_2)}{}$$

E; Σ |-- Q (P_1+P_2) P : (T_1 , T_2; valid)

(OptextType)

E |-- Q : (T ; yes)

E, (X=()+MPType, X) |-- Q *optext union occpos* : (X; yes)

where

MPType = matchpoint[document[String],
start[Integer],
end[Integer],
word[Integer]]