A FORMAL DEFINITION OF SELECTION OPERATIONS THAT EXTEND XQUERY WITH INTERACTIVE QUERY CONSTRUCTION

Alda Lopes Gançarski
University of Minho
Departamento de Informática, Campus de Gualtar, 4710 Braga, Portugal
Member of LIP6, Paris, France

Pedro Rangel Henriques
University of Minho
Departamento de Informática, Campus de Gualtar, 4710 Braga, Portugal

Keywords: XML, XQuery, Information Retrieval, Interactive search.

Abstract: XQuery is the standard language for querying XML documents using structural and content restrictions. XQuery is being complemented with a Full-Text language to perform operations on text treating it as a sequence of words, units of punctuation, and spaces. Due to the complex nature of XQuery structured queries, an extension to XQuery was informally proposed to allow for the selection of the interesting subset of elements from each intermediate result of a query. Intermediate results are, thus, available during the construction of the query, which helps the user in building a query to retrieve the desired result. In this paper, we formally define selection operations by extending XQuery grammar and defining new functions. These definitions will be used to build a processing system. The system should be incremental such that, after changing a query, only the operations depending on the changes are computed.

1 INTRODUCTION[1]

Traditional IR consists of retrieving from a collection the relevant documents to a query, while returning as few as possible of non relevant documents. Moreover, the resulting documents should be ranked by their relevance to the query. A query is a natural language expression describing the desired subject. To take advantage from the structural information of XML documents, query formats for structured documents retrieval were enriched to access certain parts of documents. So, the user can access those parts based on content and structural restrictions. Examples of such queries are those defined by XPath language (Berglund et al., 2005) and XQuery (Boag et al., 2005), the proposition by the W3C to become the standard XML query language. To include similarity search operations of traditional IR in XPath, some works developed relevance computation methods, like the ones presented in (Fuhr et al., 2004). XQuery and XPath are being extended with the possibility of associating a score (or relevance measure) to an expression that verifies if some phrase exists in the content of some element or attribute. This functionality is included in a language that complements XPath and XQuery, the Full-Text language proposed by the W3C (Amer-Yahia et al., 2005). However, structured queries construction is not always an easy process because, among other reasons, the user may not have a deep knowledge of the query language, or may not know a priori exactly what to search. Moreover, after specifying a query, the user may get a final result that it is not what was expected. To solve this problem, IXDIRQL (Gançarski and Henriques, 2003) was defined as an extension to XPath, not only with textual similarity operations, but also with an interactive/iterative paradigm for building queries. With this paradigm, each operation specified by the user leads to an intermediate result which the user can access. This helps the user choosing the next operation, changing an operation already introduced in the query, or selecting, using selection operations, the interesting subsets of intermediate results, until reaching the adequate query and thus the desired result. If intermediate results are large, the user is able select a number of interesting elements that is sufficient to satisfy him. This avoids continuing the query with a large number of unnecessary elements to process and further results are easier to analyse.

[1]This research is done in the context of the RESPIRE project financed by the French ANR-ARA programm.

148 Lopes Gançarski A. and Rangel Henriques P. (2006).
A FORMAL DEFINITION OF SELECTION OPERATIONS THAT EXTEND XQUERY WITH INTERACTIVE QUERY CONSTRUCTION.
In *Proceedings of WEBIST 2006 - Second International Conference on Web Information Systems and Technologies - Internet Technology / Web Interface and Applications*, pages 148-155
DOI: 10.5220/0001251501480155
Copyright © SciTePress

A prototype to process IXDIRQL queries was created and used by real users (Gançarski and Henriques, 2005b) allowing to verify, not only its correct behavior, but also the correct understanding and use of selection operations with respect to some pre-defined information needs. In (Gançarski and Henriques, 2005a) the authors informally suggest to extend the interactive/iterative paradigm of query construction to XQuery. For that, XQuery is augmented with selection operations. The present paper formally defines these operations in order to: (1) include them in the XQuery W3C definition (Boag et al., 2005)(Amer-Yahia et al., 2005), thus following the same formalism for grammar and functions definition; (2) build an adequate processing system.

This article is organized as follows. Section 2 introduces XQuery and Full-Text languages. Then, Sections 3 and 4 define selection operations, namely *select* and *judgeRel*, respectively. Section 5 proposes an incremental processing for the extended XQuery. The article finishes with a conclusion, giving some directives for future work.

2 *XQUERY* AND *FULL-TEXT* LANGUAGES

XQuery is formed by several kinds of expressions, including XPath location paths and *for..let..where..order_by..return* (FLWOR) expressions based on typical database query languages, such as SQL. To pass information from one operator to another, variables are used. As an example, assume a document that stores information about articles, including title, author and publisher. Next query returns articles of author Kevin ordered by the respective title.

```
for $a in /articles/article
where $a/author = ”Kevin”
order by $a/title
return $a
```

XQuery operates in the abstract, logical structure of an XML document, rather than its surface syntax. The corresponding data model represents documents as trees where nodes can correspond to a document, an element, an attribute, a textual block, a namespace, a processing instruction or a comment. Each node has a unique identity.

Full-Text language extends XQuery with *ftcontains* expressions and the inclusion of *score* variables into the FLWOR expressions. The *ftcontains* function can be used anywhere a comparison can occur, like the equal operator. An *ftcontains* expression includes a location path to specify the nodes where the function is applied and the expression of the search strings to be found as matches. *ftcontains* returns a Boolean value true if there is some node in the path expression that matches the expression of the search strings. To show an example, the following query returns the author(s) of each article whose title contains ”XML”.

```
for $a in /articles/article
where $a/title ftcontains ”XML”
return $a/author
```

A *score* variable stores the relevance measure associated to an expression that verifies if some phrase exists in the content of some element or attribute. The expression is restricted to a Boolean combination of *ftcontains* expressions. The variable gets bound to a value of type *xs:float* (the *xs* namespace refers to *XML schema*) in the range [0, 1], a higher value implying a higher degree of relevance. The value reflects the relevance of the match criteria and the way it is calculated is left implementation-dependent. The following example query returns articles (stored in *$a*) ordered by the relevance (stored in *$s*) of their title with respect to ”XML”.

```
for $a score $s in
        /articles /article [title ftcontains ”XML”]
order by $s
return $a
```

3 *SELECT* FUNCTION

The interactive paradigm of query construction is based on selection operations which consist of restricting intermediate results to the subset of elements that satisfy the user. Selection is performed in location path expressions using the *mf:select* function. The namespace prefix *mf* (from my function) used in this paper is associated to new functions.

The *mf:select* function selects the subset of interesting elements based on some criteria. While in a filter the set of elements is selected by intention, in the *mf:select* it is by extension, ie explicitly referring to each element. This can be interesting when the specification of the criteria is too complicated (the user may even not know how to do it) or when it is more efficient/rapid to directly refer the desired elements.

Suppose each node is identified by a unique identifier and consider it as a string of characters. The input to *mf:select* is a node and a list of node identifiers. The output is the input node if it is selected (i.e., if its identifier belongs to the list of identifiers), or an empty sequence of nodes (denoted by “( )”). For example, suppose the user wants references made inside interesting articles of author “*Kevin*”. Here, *interesting* may refer, among other things, to the article's title, co-authors, publisher, date, size. The user can, then, make the following query:

```
for $a in /articles/article[author = "Kevin"]
    [mf:select(., ("a4", "a8"))]
return $a//references
```

In this query, function *mf:select* selects articles identified by *"a4"* and *"a8"*. Symbol "." refers to each context node, i.e., each resulting node of the precedent operation. Thus, *mf:select* takes each article being a context node and returns it if it corresponds to some of the selected items.

Due to the interactive nature of the *mf:select* function, this example query is written in three steps:

1. The user specifies the *for* clause with the path returning the list of articles of author *"Kevin"*:
   for $a in /articles/article[author = "Kevin"]

2. Analysing the list of articles given by the path, the user selects the interesting ones with the *mf:select*:
   for $a in /articles/article[author = "Kevin"]
   [mf:select(., ("a4", "a8"))]

3. The user completes the query with the *return* clause:
   for $a in /articles/article[author = "Kevin"]
   [mf:select(., ("a4", "a8"))]
   return $a//references

Despites *mf:select* receives a list of node identifiers, the user is not obliged to know them with a good intermediate results view. This view should allow the selection of interesting elements by using, for instance, a button associated with each element. The system should, then, automatically write the element identifiers in the query edition view.

XQuery allows for user defined functions, such us *mf:select*. To formally define *mf:select*, let *IdNodeTab* be a table maintained by the system that makes each node to correspond to its own identifier:

IdNodeTab : xs:string $\times$ *node()*

The XQuery node test *node()* matches any node. The *mf:select* function can, then, be defined by:

```
declare function mf:select($contextNode as node( )?,
    $selectedIds as xs:string*) as node( )?
  {
      for $s in $selectedIds
      let $n := IdNodeTab[$s]
      if ($contextNode=$n)
          return $contextNode else return ( )
  }
```

Here, *$selectedIds* is a variable containing the list of selected node identifiers of type *xs:string*. Variable *$n* stores, for each selected identifier, the corresponding node given by table *IdNodeTab*. The function returns the context node if it is the same as some node in *$n*.

4 *JUDGEREL* OPERATOR

The *judgeRel* operator selects the subset of elements judged relevant by the user among the ones in the resulting ranked list returned by a *ftcontains* expression associated to a score variable. Let the following be an example query:

```
for $a score $s in
    /articles/article[title ftcontains "XML"]
order by $s
return $a/references
```

This query returns a list of references ranked by the relevance corresponding to the title of the article where they are cited. These references may or not correspond to effective relevant titles as they come from the ranked list of titles estimated by the processing system. Using the *judgeRel* operator, the user can judge and select relevant elements during query construction by analysing the resulting ranked list given by *ftcontains*. Consequently, the relevance associated to relevant elements becomes 1 and to non-relevant ones becomes 0. These new relevance values are taken into account in the final score computation. In the previous query, suppose title elements identified by *"t4"* and *"t8"* are judged relevant and selected when the user analyses the ranked list returned by the *ftcontains* clause. Then, the query becomes:

```
for $a score $s in /articles/article
    [title ftcontains "XML" judgeRel ("t4","t8")]
order by $s
return $a/references
```

Here, the list of references returned in the *return* clause is composed of references coming from articles where the title is for sure relevant (the user judge it relevant).

As with the *mf:select* function, due to the interactive nature of the *judgeRel* operator, the example query is written in three steps:

1. The user specifies the *for* clause and the *ftcontains* expression:
   for $a score $s
   in /articles/article[title ftcontains "XML"]

2. The resulting ranked list of the *ftcontains* clause gives the user a good starting point to search relevant titles. Analysing it, the user inserts the *judgeRel* operator with the found relevant elements:
   for $a score $s in /articles/article
   [title ftcontains "XML" judgeRel ("t4","t8")

3. Finally, the user writes the *order by* and the *return* clauses to have the final list of references:
   for $a score $s in /articles/article
   [title ftcontains "XML" judgeRel ("t4","t8")]
   order by $s
   return $a/references

As with the *mf:select* function, the view showing the ranked list should allow the user to directly choose the relevant elements, avoiding to know their internal node identifiers.

4.1 Syntax Definition

The *judgeRel* operator must be included in the XQuery grammar extended with Full-Text language grammar presented in (Amer-Yahia et al., 2005). In this grammar, productions number 35, 37, 38 and 51 derive the *for* clause, a score variable, the *let* clause and the *ftcontains* expression, respectively[2]:

[35] ForClause ::= "for" "$" VarName ...
FTScoreVar? "in" ExprSingle ...
[37] FTScoreVar ::= "score" "$" VarName
[38] LetClause ::=
(("let" "$" VarName ... FTScoreVar?) |
("let" "score" "$" VarName)) ":=" ExprSingle ...
[51] FTContainsExpr ::=
RangeExpr ("ftcontains" FTSelection ...)?

In production number 51: *RangeExpr* derives the expression that yields the list of nodes where the *ftcontains* is applied, also called the search context (list of context nodes); *FTSelection* derives Boolean combinations of phrases to search and match options, such as case sensitivity. In productions 35 and 38, the score variable stores the score associated to the expression derived by *ExprSingle*. This last symbol derives any kind of XQuery expression, such as FLWOR expressions and *ftcontains* expressions. However, the expression associated to score variables is restricted to a Boolean combination of *ftcontains* expressions, involving only "and" and "or" operators. Consequently, we propose to substitute *ExprSingle* in productions number 35 and 38 by *ScoreExpr*, yielding:

[35] ForClause ::= "for" "$" VarName ...
(FTScoreVar "in" ScoreExpr |
"in" ExprSingle) ...
[38] LetClause ::= ("let" "$" VarName ...
(FTScoreVar ":=" ScoreExpr |
"=:" ExprSingle) |
"let" "score" "$" VarName ScoreExpr) ...

Symbol *ScoreExpr* derives the Boolean combination of *ftcontains* expressions in the following productions:

ScoreExpr ::= ScoreOrExpr
ScoreOrExpr ::= ScoreAndExpr |
ScoreAndExpr "or" ScoreOrExpr
ScoreAndExpr ::= ScoreExprUnit |
ScoreExprUnit "and" ScoreAndExpr
ScoreExprUnit ::= "(" ScoreExpr ")" |
RangeExpr ("ftcontains" FTSelection ...
JudgeRelExpr?)?

As in XQuery grammar specified in (Amer-Yahia et al., 2005), productions reflect operator precedence. Higher precedence operators appear more deeply nested. Symbols *ScoreOrExpr* and *ScoreAndExpr* derives an "or" and an "and" Boolean operation, respectively. The symbol *ScoreExprUnit* derives a *ScoreExpr* expression between parenthesis or derives *ftcontains* expressions associated to score variables. These expressions are similar to those derived by production number 51 augmented with the optional *judgeRel* operator. The symbol *JudgeRelExpr* derives the *judgeRel* operator by the following production:

JudgeRelExpr ::= "judgeRel" "(" StringLiteral ")"*

The *StringLiteral* symbol defined in the XQuery grammar allows to derive a node identifier. *judgeRel* is, thus, associated to the set of node identifiers judged relevant by the user.

4.2 Semantics Definition

The *judgeRel* operator is included in expressions that compute score variables. Thus, its semantic definition is given together. However, the definition of those expressions cannot be expressed in terms of XQuery, because they require the presence of second-order functions (i.e. functions that do not evaluate their argument(s) as regular XQuery expression(s) but use them interpreted). It is assumed in (Amer-Yahia et al., 2005) that there is a semantic second-order function *fts:score* that takes one argument (a *ScoreExpr* expression) and returns the score of this expression. Given this function, the generic expression

[2]For simplicity, some optional symbols are substituted by "...".

score $var as ScoreExpr is evaluated as though it is replaced with *$var:=fts:score(ScoreExpr)*, where *fts* namespace refers to Full-Text semantics. We propose to define the *fts:score* function as follows:

```
declare function fts:score($e as xs:string) as xs:float
{
1. if (mf:operatorScore($e) = ”or”) then
2.     mf:scoreOr(fts:score(mf:operandLeftScore($e)),
3.                fts:score(mf:operandRightScore($e)))
4. else if (mf:operatorScore($e) = ”and”) then
5.     mf:scoreAnd(fts:score(mf:operandLeftScore($e)),
6.                fts:score(mf:operandRightScore($e)))
7.  else
8.     let $s := mf:searchContext($e)
9.     return
10.        if (mf:includesJudgeRel($e)) then
11.           let $j := mf:judgeRelIds($e)
12.           let $i := for $a in $j return IdNodeTab[$a]
13.           return mf:scoreJudgeRel($s, $i)
14.        else
15.           let $m := mf:matchExpr($e)
16.           return mf:scoreFTContains($s, $m)
}
```

The argument of the function is a string corresponding to the score expression derived by the *ScoreExpr* symbol defined in Section 4.1. The function returns a float value *xs:float*.

Due to the recursive calls to the *fts:score* function (lines 2, 3, 5, 6), the score is computed, first, for each *ftcontains* expression, and then for each Boolean operator of the *ScoreExpr* expression, respecting operator precedence, until a final result.

In line 1, the function *mf:operatorScore* takes the *ScoreExpr* expression and gives the first operator to evaluate: an ”and”, an ”or” or none. For that, operator precedence is taken into consideration. Depending on the operator, different actions are taken. If the operator is an ”or” (line 1), the score is computed by function *mf:scoreOr* applied to the score of both left and right operands of the ”or” (lines 2 and 3, respectively). Those operands are given by functions *mf:operandLeftScore* and *mf:operandRightScore*, respectively. If the operator is an ”and” (line 4), a similar action is taken, being now the score computed by the function *mf:scoreAnd* (lines 5 and 6).

If no operator is found, the score of a *ftcontains* expression derived by symbol *ScoreExprUnit* (defined in Section 4.1) is computed by the actions between lines 7 and 16. Variable *$s* stores the search context derived by symbol *RangeExpr* (presented in Section 4.1) (line 8). This is done by function *mf:searchContext*. The existence of a *judgeRel* operator is, then, verified by function *mf:includesJudgeRel* analyzing the *ScoreExpr* expression. If there is such operator (line 10), the following actions are done. Variable *$j* stores the node identifiers that are judged relevant by the user (line 11). These are given by function *mf:judgeRelIds* which receives the *ScoreExpr* expression. Another variable, *$i*, stores the nodes corresponding to the identifiers judged relevant by the user (line 12). These nodes are given by table *IdNodeTab* (presented in Section 3). Function *mf:scoreJudgeRel* takes the list of search context nodes (stored in *$s*) and the list of nodes judged relevant (stored in *$i*) and gives the resulting score of the score clause (line 13).

If there is no *judgeRel* operator in the *ScoreExpr* expression (line 14), variable *$m* stores the Boolean combinations of phrases to search and match options derived by symbol *FTSelection* (presented in Section 4.1). This is done by function *mf:matchExpr* (line 15). Then, taking variable *$m*, function *mf:scoreFTContains* computes the score associated to the search context nodes stored in variable *$s* (line 16).

The new functions used inside *fts:score* are not defined here more in detail. Most of them give the result based in a simple lexical/syntactic analysis of the *ScoreExpr* expression to find specific subexpressions (*mf:operatorScore*, *mf:operandLeftScore*, *mf:operandRightScore*, *mf:includesJudgeRel*, *mf:judgeRelIds*, *mf:matchExpr* and *mf:ignoreOption*). The function *mf:searchContext* finds a sub-expression to compute a list of corresponding nodes. The remaining functions (*mf:scoreOr*, *mf:scoreAnd*, *mf:scoreJudgeRel* and *mf:scoreFTContains*) are dedicated to score computation. The Full-Text language and the extensions made here are independent of the score computation method. So, each application can choose its own method for *ftcontains* expressions and their Boolean combinations. For example, in (Gançarski and Henriques, 2005a), a method is proposed for the XQuery extended with selection operations.

4.3 An Example of *fts:score* Processing

As an example of executing the *fts:score* function, consider the following query:

```
for $a score $s in
    /articles/article[reference ftcontains ”XML”
    and section ftcontains ”XML” judgeRel (”s1”)]
order by $s
return $a/title
```

In what follows, for simplicity, the definition of *fts:score* given in Section 4.2 is referred by the lines of the actions to execute. Also, element nodes are referred by their identifiers.

The previous query returns titles of articles where references are about *"XML"* and sections are about *"XML application"*. Resulting titles are ordered by their score. Assume that article *a1* was found in the for clause. Assume also that it has sections *s1* and *s2* and references *r1* and *r2*. When function *fts:score* is executed, the Boolean operator "and" is detected by function *mf:operatorScore* in line 4. Thus, function *fts:score* is recursively called for both operands of the "and", as indicated in lines 5 and 6. Those operands are subexpressions of the *ScoreExpr* expression in the score clause given by functions *mf:operandLeftScore* and *mf:operandRightScore*. The corresponding results are used as arguments to the *mf:scoreAnd* function to compute the final result of the score clause for article *a1* (line 5). If there are more articles, a score is computed for each one using again the function *fts:score*.

For both arguments of the "and" operator, the *fts:score* function is executed after line 7 because there are no more Boolean operators. Concerning the first argument, the search context is computed by function *mf:searchContext*, returning the reference list of nodes *("r1", "r2")* stored in variable *$s*. The function *mf:includesJudgeRel* verifies that there is no *judgeRel* operator (line 14) and the execution continues in line 15. Here, *mf:matchExpr* function returns the phrase to search "XML" stored in variable *$m* (there are no match options). This phrase, together with the search context, is given to function *mf:scoreFTContains* to compute the resulting score of the first argument of the "and" operator.

Concerning the second argument, line 8 is also executed to compute the search context, in this case the list of section nodes *("s1", "s2")*. This argument has a *judgeRel* operator. Consequently, actions of lines 11 to 13 are executed. The user judged relevant section *s1*. This node identifier is given by function *mf:judgeRelIds* (line 11). The corresponding node is, then, given by table *IdNodeTab* (line 12). The resulting score is, finally, given by function *mf:scoreJudgeRel* which takes the search context *("s1", "s2")* and the list *("s1")* of sections judged relevant in this context (line 13).

5 INCREMENTAL QUERY PROCESSING

The editing environment for the extended XQuery must allow the user to access intermediate results of query operations. Besides, it should be associated with an incremental processing of query operations. This means that, each time a new operation is inserted or an existing one is changed, the system does not calculate all the query operations. Instead, it first calculates the intermediate results of the new or changed operation; then, it recalculates the intermediate results that are dependent on the previous ones and the final result of the query.

5.1 *fts:score* Incremental Processing

A particular case of incremental operation evaluation is for the *fts:score* function defined in Section 4.2 because it includes many operations. Suppose, for instance, that the user is specifying a query with the following *for* clause:

for $a score $s in
/articles/article[title ftcontains "XML"]

The value of the *score* variable is given by *fts:score*. As there is not yet a *judgeRel* operator, the else condition in line 14 is executed. For a correct access to intermediate results, the resulting list of titles of the search context stored in variable *$s* (line 8) should be presented to the user, together with the respective scores computed in line 16 by function *mf:scoreFTContains*. Facing this list, if the user judges relevant title identified by *"t1"*, the query becomes:

for $a score $s in articles/article
[title ftcontains "XML" judgeRel ("t1")]

The *fts:score* function is executed again, now executing lines 11 to 13 because there is the *judgeRel* operator. The incremental query processing must assure that all the computations executed before these lines are not done again.

5.2 Automatic Generation of an Incremental Processing Prototype

We propose to build an incremental editor/processor using LRC (Kuiper and Saraiva, 1998), as done for IXDIRQL (Gançarski and Henriques, 2005a). LRC is a generator of incremental environments based on formal definition of languages. Language definition is made through an attribute grammar (AG) which consists of a context free grammar extended with a set of attributes (and semantic rules for their evaluation) to specify the semantics of the analyzed texts. If necessary, it also allows imposing contextual conditions to productions of the grammar, based on attribute values. Contextual conditions correspond to the static semantics, in opposition to dynamic semantics, which consist of computing the meaning of a text of the language. Editors generated by LRC are syntax-directed.

This helps the user to write his texts by making explicit the syntax of the language and also its static semantics.

If LRC generates an environment for XQuery, we have: (1) A text is a query. (2) Language syntax is given by the XQuery grammar defined in (Amer-Yahia et al., 2005). (3) Dynamic semantics corresponds to the evaluation of query results. It is based on the semantic definition of the XQuery and Full-Text, including the new productions and functions defined in this paper. (4) Static semantics verifies, among other things, which elements are valid operands for each location path operation. Elements validation is based on the documents DTD or Schema.

To exemplify the XQuery language definition by an AG to give to LRC, suppose that attribute *aScore* stores the score associated to symbols *ScoreOrExpr* and *ScoreAndExpr* defined in Section 4.1. Then, the production where *ScoreOrExpr* is derived and the rule to compute the value of *aScore* are, respectively:

ScoreOrExpr ::= ScoreAndExpr "or" ScoreOrExpr
ScoreOrExpr$1.aScore =
mf:scoreOr(ScoreAndExpr.aScore,
ScoreOrExpr$2.aScore)

Here, the two occurrences of *ScoreOrExpr* are distinguished by suffixes *$1* and *$2*, representing the position of the symbol in the production. Attribute *aScore* of symbol *ScoreOrExpr$1* is denoted by *ScoreOrExpr$1.aScore* (the same for *ScoreOrExpr$2* and *ScoreAndExpr*).

The score is calculated by function *mf:scoreOr* introduced in Section 4.2. It take as arguments attributes *aScore* of both symbols on the right hand side of the production.

To compute attribute values, a derivation tree of queries is first created. Then, each node in the tree is decorated with its attributes and attribute values which are computed accordingly to the corresponding rules. These rules define a computation order in the attributes because they can be dependent on each other, yielding a dependencies graph.

Each time a text (a query in our case) is changed, the dependencies graph changes. Then, the incremental attribute evaluator computes the values of the new attributes in the graph and the values of existing attributes that depend on the new ones. The incremental evaluation is obtained via standard function memoization. It is out the scope of the paper the presentation of this method, the interested reader being able to find details in (Saraiva et al., 2000).

6 CONCLUSION AND FUTURE WORK

This paper formally defines an extension to XQuery with selection operations for the interactive/iterative query construction. This helps the user, not only in choosing the operations that yield the desired answer, but also in restricting each intermediate result to the subset of nodes that pleases the user. The proposed formal definition can be used to build a processing system for the interactive edition and processing of XQuery. As future work, a prototype of a processing system will be built using LRC, as explained in Section 5.2. For score computations, the method proposed in (Gançarski and Henriques, 2005a) can be used. Once created, the prototype will be used by real users to verify the correct understanding and use of selection operations, as well as the interest of accessing intermediate results during query construction.

ACKNOWLEDGEMENTS

The authors are grateful to the Portuguese Fundação para a Ciência e a Tecnologia for the financial support.

REFERENCES

Amer-Yahia, S., Botev, C., Buxton, S., Case, P., Doerre, J., McBeath, D., Rys, M., and Shanmugasundaram, J. (2005). XQuery 1.0 and XPath 2.0 Full-Text Working Draft. http://www.w3.org/TR/2004/WD-xquery-full-text-20040709/.

Berglund, A., Boag, S., Chamberlin, D., Fernandez, M., Kay, M., Robie, J., and Siméon, J. (2005). XML Path Language (XPath) 2.0 W3C Working Draft. http//www.w3c.org/xpath20/.

Boag, S., Chamberlin, D., Fernandez, M., Florescu, D., Robie, J., and Siméon, J. (2005). XQuery 1.0: An XML Query Language. W3C Working Draft. http://www.w3.org/TR/xquery/.

Fuhr, N., Lalmas, M., Malik, S., and Szlávik, Z., editors (2004). *INEX: Initiative for the Evaluation of XML Retrieval Workshop Proceedings*. DELOS Network of Excellence in Digital Libraries, Schloss Dagstuhl, Germany.

Gançarski, A. and Henriques, P. (2003). IXDIRQL: an Interactive XML Data and Information Retrieval Query Language. In *Proceedings of the 7th ICCC/IFIP International Conference on Electronic Publishing*, Guimarães, Portugal.

Gançarski, A. and Henriques, P. (2005a). A processing environement for the IXDIRQL XML query language. In *Proceedings of the IADIS Virtual Multi Conference on Computer Science and Information Systems (MCCSIS05)*.

Gançarski, A. and Henriques, P. (2005b). Extending XQuery with selection operations to allow for interactive construction of queries. In *Proceedings of the 9th ICCC International Conference on Electronic Publishing*, Leuven, Belgium.

Kuiper, M. and Saraiva, J. (1998). LRC: A Generator for Incremental Language-Oriented Tools. In *7th International Conference on Compiler Construction*, volume 1383, pages 298—301. LNCS.

Saraiva, J., Swierstra, D., and Kuiper, M. (2000). Functional Incremental Attribute Evaluation. In *9th International Conference on Compiler Construction (CC/ETAPS'00)*, volume 1781. LNCS.