Projector – a partially typed language for querying XML

EXTENDED ABSTRACT

Richard Connor, David Lievens, Fabio Simeoni, Steve Neely and George Russell

Department of Computer and Information Sciences,
University of Strathclyde, Glasgow G1 1XH

Overview

Our context is standard programming languages used to query XML data. We are interested in "typeful" programming, by which we mean programming with regular structures used to model categories in the real world. Real-world entities are intuitively modelled in XML as labelled trees; however when these trees are presented to the programmer, as in various standards such as DOM, SAX, XLST and XQUERY, only the structure of the tree itself, rather than the real-world structure it is used to represent, is given. The entity modelling that has been used must be discovered through dynamic interpretation within the tree structure.

The XML namespace (xmlns) standard allows the effective introduction of some "type" knowledge for the programmer, albeit by convention rather than by guarantee. However this still leaves much to be desired for programmers trying to recreate the higher-level conceptual structures from labelled trees as presented. The namespace mechanism is relatively heavyweight for many purposes, and may require explicit structural checking within the program logic to ensure that conventions are obeyed. As well as these, it has the established disadvantages that "name equivalence" type systems suffer in comparison with "structural equivalence" over distributed programming systems, in particular with respect to evolution and version control.

We examine the addition of a structural equivalence mechanism into the framework. Through a scheme of type projection, we allow structural type assertions to be tested during program execution; this allows fragments of that code to be statically checked based on the type hypothesis coded in the projection. Our prototype language Projector is an extension of the ECMA-262 ("JavaScript") standard, and allows an interesting mix of typed and untyped code within a single context.

Introduction

We have previously worked on type projections over semistructured data, with the aim of allowing standard statically typed programming languages to bind, in a semantically intuitive way, to semistructured data emanating from outside their context. We now report a different application of this work, where the type projection algebra is embedded within a statically untyped programming language, giving a language which can query and manipulate its values in both typed and untyped algebras. When this is applied to the DOM standard as an XML query interface, we can use type projection to search via navigation for sub-trees that conform to a particular type's semantics, and thence provide a generic typed interface to a programmer.

The type projection mechanism gives several advantages over the basic DOM abstraction. As well as safety issues normally associated with type systems, the descriptive power of the type system can be used to handle cycles and shared subgraph components within the XML. While these are, by necessity, handled through interpretation, this happens at an earlier stage and is cleanly separated from the application logic rather than being intermingled.

While of course our language has no more descriptive power than the standard DOM interface, we believe that the effective encoding of a part of the language algebra within a type system framework gives clear succinctness to certain classes of computation. We use the type system framework in a very non-standard manner; rather than being primarily a static mechanism to ensure the partial safety of programs, we also use it to allow the programmer to express certain computation against the input data. We strongly believe that type system concepts are the best way to describe structural requirements that are normally expressed by fragments of computation.

Incorporation of static typing into ECMA-262

Our prototype language, Projector, is an extension of the language defined by the December 1999 ECMA-262 standard of JavaScript/JScript[1]. We chose this language for a number of important reasons: first, it is statically completely untyped, which suits our experimental purposes. Despite this, it is well-defined and in fact, being an evolved form of LISP, contains an elegant functional core, and a pure object model of prototype-based inheritance over aggregations of first-class functions. The language is actually defined on the basis of a type system, but this features only in the definition of semantics rather than in any static framework. Finally, it has a standard binding to XML via the DOM interface.

To this language we have added some standard syntactic forms for defining types, comprising object and array constructors over the scalar types *int*, *string* and *bool*, and a syntax for aliasing [SYN]. Type expressions are added to the standard syntax as an optional feature within the parameter list of functions. The language remains largely untyped: a single static restriction has been added, that where a formal parameter is typed in a function body, and that function is manifest[2], then the corresponding actual parameter at a call must be appropriately statically typed.

In itself, this partially typed version of JavaScript opens many questions about the integration of typed and untyped programming algebras; however many of these issues are parallel to the topic of this paper. Here we concentrate on the application of this paradigm to programming over XML data as presented to the system via the DOM.

Figure 1 shows the essential features of the extended language, and also hints at the motivation for adding a type system to a language which remains largely untyped.

[1] We will subsequently refer to this language as JavaScript, forsaking both political and technical correctness for the sake of readability!

[2] Functions are first class and we do not, at this point, type them.

The two functions *getName* and *getName2* both return the name of their *person* argument. The typed version is checked statically and the function body is therefore guaranteed to behave as expected. The untyped version instead has a dynamic check to ensure it has been called appropriately.

```
type person = { name : string; age : int }

var getName = function( p : person )
{
   return( p.name )
}

var getName2 = function( p )
{
   if( p.name != undefined && p.age != undefined &&
   typeof( p.name ) == "string" && typeof( p.age ) == "number" ){
      return( p.name ) }
   else{
      error( "getName2: p is not a person" ) }
}

var richard = { name : "Richard", age : 40 }
var fabio = { name : "Fabio" }

print( getName( richard ) )
print( getName2( fabio ) ) // causes a dynamic error
//print( getName( fabio ) ) // causes a static error
```

Figure 1 : structure checking by type and by algorithm

It is clear that the first function is the preferred form, as the error is caught statically instead of dynamically. However total static checking is not possible when some of the data is imported into the context during execution.

The other point to compare is the succinctness of code. If we were to define this language in an unorthodox manner, and use the type information to generate dynamic rather than static checks (effectively generating the second form from the syntax of the first) then there is still an advantage as the syntactic form of the first function directly reflects the same structural requirement in a much more succinct manner. The program is therefore more likely to reflect the programmer's intention, irrespective of any static consideration. This succinctness, albeit in combination with increased static safety, is the main claim we make of our projection mechanism over the DOM.

By adding explicit projection of types over the DOM structure, we believe we achieve both of these advantages in a single mixed-mode programming algebra.

Typed projection from DOM

We start by giving a motivating example drawn from an artificially simple XML data collection, valid with respect to the DTD[3] given in Figure 2.

```
<!ELEMENT people ( person+ ) >
<!ELEMENT person ( name, age ) >
<!ELEMENT name ( #PCDATA ) >
<!ELEMENT age ( #PCDATA ) >
```

Figure 2 : example trivial DTD

A Projector function to calculate the average age of this collection may be written as in figure 3. The *Collection* type describes a JavaScript model of the expected structure of the input data, modelled within the XML tree. The *project* keyword signifies a static assertion of the expected structure of the function parameter *XMLData*, which should be a DOM tree conforming to the DTD given in Figure 2. Its use conveys two effects: first, it tests the structure of the input, and will throw an exception if this is incorrect. Secondly, it results in a reference to a data structure conforming to the given type description, *Collection*, which in this example is assigned to the local variable *data*. From this point onwards, data is statically known to have the type *Collection*. Even in this example, where no mechanical use is made of this fact, the implication is that the code in the next line, which relies upon this type structure, is known to be type-safe by the programmer and allowance for failure need not be made.

```
type Collection = { people : { $array_person : [ { name : string, age : int } ] } }

var averageAge = function( XMLData )        // should be head of DOM tree
{
    var count = 0; var total = 0
    var data = project XMLData onto Collection
    var people = data.people.$array_person
    for( i in people ){ count++; total += people[ i ].age }
    return( total / count )
}
```

Figure 3 : succinct structure checking through projection

This motivating example looks convincing in terms of the succinctness of expression it achieves in comparison to code with an equivalent meaning written directly over the

[3] XMLSchema gives a tighter definition more fit for our purpose, but currently suffers less general support than DTD.

structure of the DOM tree. However when the use of the paradigm is extended to non-trivial examples, which by nature do not fit in academic papers, its use becomes more significant. Both DTD and XMLSchema tend to be used to describe general grammars, rather than tightly structured types, and typically allow great flexibility in conformance. In such cases structural projection looks even more convincing in terms of allowing code whose meaning is clear to programmers. This is of course true only in cases where a common structure, typically a subset of allowed structures, is the target of the computation.

The projection mechanism

We have described type projection schemes for semistructured data in detail elsewhere [SNAQUE], and give a simple overview here. It is worth noting that our motivation has previously been to integrate semistructured data with a statically typed context; however in the above example no static typechecking is performed, yet the value of the paradigm in terms of succinctness of expression alone is clear.

The basic model comprises a type grammar with two overlapping subsets. These subsets are termed the *SS* and *PL* type description frameworks; the *SS* system can be used to assign a type to any semistructured collection, and the *PL* is a standard programming language type system. A subtype relation is defined, based on a semantics of the type grammar in the domain of the semistructured data. If a *PL* type is a supertype of the *SS* type assigned to a data collection, then that collection may be viewed as the *PL* type. An underlying mechanism based on either indexing or extraction is then used to present the appropriately typed data in the context of the high-level language.

In the context of JavaScript, an appropriate system is shown in Figure 4. The curly and square brackets are syntactic forms representing JavaScript object and array type constructors. One unusual aspect of the general type grammar *T* is that repeated label names are allowed within object types; when the normal restriction of non-repetition is imposed, the *PL* subset is derived. The *SS* subset comprises the scalar types with the object type constructor, and allows label repetition.

$$T ::= \text{int} \mid \text{string} \mid \text{bool} \mid \text{null} \mid \{\, l_1 : T, \ldots , l_n : T \,\} \mid [\, T \,]$$

$$SS ::= \text{int} \mid \text{string} \mid \text{bool} \mid \text{null} \mid \{\, l_1 : SS, \ldots , l_n : SS \,\}$$

$$PL ::= \text{int} \mid \text{string} \mid \text{bool} \mid \text{null} \mid \{\, l_1 : PL, \ldots , l_n : PL \,\} \mid [\, PL \,] \qquad l_i \neq l_j$$

Figure 4 : The three type grammars for JavaScript/XML projection

The type assignment from any simple XML document onto *SS* is straightforward: the XML tree is simply typed as a collection of nested objects. Scalar content is typed as *int*, *bool*, *null* or *string* according to its structure, and structured content is typed as an object, each tag name being represented by a label. The lack of repetition restriction in the definition of object typing deals with the case of repeated tags within a single nesting context. Type assignment is extended to cover attributes and mixed content also, by use of the label prefix *$attribute_* and the label *$mixedContent* as used in Figure 5[4].

```
<person xmlns="person.richard.cis.strath.ac.uk">
   <name>Richard</name>
   <age>40</age>
   <motto>XML doesn’t care.</motto>
   <motto><emphatic>Never</emphatic> buy a horse from a bishop.</motto>
</person>

:

{ person : {    $attribute_xlmns : string, name : string, age : int, motto : string,
                motto : { emphatic : string, $mixedcontent : string } } }
```

Figure 5 : an example type assignment

The subtype relation is based on the following semantic interpretation of the type grammar within the XML context:

1. objects are represented by a set of elements at the same level, where the tag names represent the object field names.

2. arrays, which must be contained within objects and labelled with an identifier of the form $array_*X*, are represented by a set of elements at the same level which share a common tag name *X*.

3. scalar types are represented by text conforming to the structural rules of that type, as defined in the microsyntax of the language.

The subtype relation is informally defined by the rewrite rules given in Figure 6, which specify a mechanism for rewriting a given type as a supertype. The reason for expressing the relation in this unusual form is that this represents exactly the process required when a projection is applied: if the type assigned to the XML can be rewritten as the goal type, then the projection is valid. Furthermore, the structure of the rewrite rules gives a basis for performing the required extraction or building of indexing structures.

$$\{ l_1 : T_1, \dots , l_m : T_m , \dots , l_n : T_n \} \Rightarrow \{ l_1 : T_1, \dots , l_n : T_n \} \tag{1}$$

$$\{ l_1 : T_1, \dots , l_m : T_m , \dots , l_n : T_n \} \Rightarrow \{ l_1 : T_1, \dots , \$array\_l_m : [T_m] , \dots , l_n : T_n \} \tag{2}$$

[4] Variable names starting with $ are legal JavaScript; these are normally reserved for system use and are not recommended for use by applications programmers.

$\{ l_1 : T_1, \ldots , \$array\_l_m : [T_m] , \ldots , l_m : T_m , \ldots, l_n : T_n \} \Rightarrow$
$\{ l_1 : T_1, \ldots , \$array\_l_m : [T_m] , \ldots , l_n : T_n \}$ (3)

$int \Rightarrow string \quad bool \Rightarrow string \quad null \Rightarrow string$ (4-6)

Figure 6 : The subtype relation expressed by rewrite rules

The rules given are not formally exhaustive but give the four main axioms of subtyping in a readable manner. Rule (1) is standard record subtyping; from any object type, a supertype can be obtained by dropping any *label* : *type* pair from the structure. Rule (2) is an array introduction, which states that any single tag *X* in an object can be viewed as a array, labelled *$array_X*, with a single element. Rule (3) is an array assimilation, which allows other fields with the same label and type to be assimilated into such an array once formed. Rules (4-6) are just an admission that the eager typing of scalar values according to their structure does not necessarily reflect their intended meaning.

A variant of rule (2), which will be used later in the paper, is given in Figure 7. This version seems less justifiable, but is useful in conditions where it is sensible for an object abstraction to be typed as containing an array of some type even when the current manifestation does not do so; logically this assumes the presence of an empty array. Whether this is desirable or not depends on the nature of the application; how to handle this elegantly is an open issue.

$\{ l_1 : T_1, \ldots , l_n : T_n \} \Rightarrow \{ l_1 : T_1, \ldots , l_n : T_n, \$array\_l_p : [T_p]\}$ (2a)

Figure 7 : introduction of “empty” arrays

Fragment Example

We now give some more sophisticated examples of the use of the paradigm: XML fragments; recursive types, and interpreted references within XML denoting shared subgraphs or cycles. For these purposes we modify our earlier DTD to that of Figure 8.

```
<!ELEMENT person ( name, age, child* ) >
<!ATTLIST person myid ID #REQUIRED >
<!ATTLIST person xmlns CDATA #FIXED “person.richard.cis.strath.ac.uk” >
<!ELEMENT name ( #PCDATA ) >
<!ELEMENT age ( #PCDATA ) >
<!ELEMENT child EMPTY >
<! ATTLIST child childId IDREF #REQUIRED >
```

Figure 8 : a more realistic DTD

An increasing use of XML conforms to the general principle of using the *xmlns* standard as a mechanism akin to name type equivalence matching, therefore allowing generic code to be written independently of its context of use. The mandatory embedding of a URI in the namespace attribute of person in Figure 8 means that general traversal code can be written to locate instances of valid data within arbitrary XML collections.

Projection was originally envisaged as a binding mechanism to allow the incorporation of semistructured data into a statically typed programming algebra. In this context however it can be useful for the different (navigational versus structured) views over the DOM trees to coexist. In the mixed paradigm, for instance, it is possible to write an unstructured, navigation-based traversal over the tree and apply projections wherever the structured view is more appropriate. This style of programming is particularly well-suited to tasks where only partial knowledge of the data is available. One common case of this is where fragments of the data are governed by a global XML namespace, and code is written over those fragments independent of the context in which they occur.

Code to calculate the average age of all people records embedded in any collection can be written using JavaScript first class functions against the standard DOM model as in Figure 9; this code abstracts away the test for whether a particular DOM node represents a person or not.

```
var applyToDOMTree = function( node, f )
{
    if( node != undefined )
    {
        f( node )
        applyToDOMTree ( node.firstChild, f )
        applyToDOMTree ( node.nextSibling, f )
    }
}

var averageAge = function( XMLData )
{
    var count = 0; var total = 0
    var accumulate = function( n )
    {
        if( isPerson( n ) ){ count++; total += getAge( n ) }
    }
    applyToDOMTree( XMLData, accumulate )
    return( total / count )
}
```

Figure 9 : generic traveral of the DOM

Minimal JavaScript code for the two functions *isPerson* and *getAge* is shown in Figure 10. Notice that this code is not guaranteed to succeed structurally, and will only do so if the namespace convention is correctly enforced throughout the use of the URI; otherwise the *getAge* function may fail dynamically or, worse, succeed mechanically but result in an incorrect answer. Full code for *isPerson*, which guarantees the correct meaning for *getAge*, must incorporate many more structural checks; even so, the onus is still on the programmer to ensure that the data extraction expressed within *getAge* corresponds correctly to the DOM structure corresponding to the schema description.

```
var isPerson = function( node ) // node is a DOM tree node
{
    try{
        var isInNamespace = node.attributes[ 1 ].nodeName = "xmlns" &&
            node.attributes[ 1 ].nodeValue == "person.richard.cis.strath.ac.uk"
        return( isInNamespace )
    }
    catch( e ){ return( false ) }
}

var getAge = function( p ) // p is a DOM tree node
{
    return( p.firstChild.nextSibling.nodeValue )
}
```

Figure 10 : “dynamically typed” DOM code

Figure 11 shows the equivalent Projector code for the two functions. Once again two significant advantages are highlighted. First is the succinctness of expression of the specification of the dynamic structural test, as seen in the *isPerson* function. The *isPerson* function in Projector is no harder to read than that of Figure 10, even although the latter does not perform any structural checks; the equivalent JavaScript code to perform the same degree of structural checking is given in Figure 12. Secondly is the static safety shown in the *getAge* function, giving the programmer confidence that the extraction expression is the correct one as a static error would otherwise be reported.

```
type Person = { $attribute_xmlns : string, name : string, age : int }

var isPerson = function( node ) // node is a DOM tree node
{
    try{
        var p = project node onto Person
        return( p. $attribute_xmlns == "person.richard.cis.strath.ac.uk" )
    }
    catch( e ){ return( false ) }
}

var getAge = function( p ) // p is a DOM tree node
{
    var p = project node onto Person
    return( p.age )
}
```

Figure 11 : the same example expressed in Projector

```
function isPerson()
{
    if( node!= undefined &&
        node.attributes != undefined &&
        node.attributes[ 1 ] != undefined &&
        node.attributes[ 1 ].nodeName == "xmlns" &&
        node.attributes[ 1 ].nodeValue == " person.richard.cis.strath.ac.uk " &&
        node.firstChild != undefined &&
        node.firstChild.nodeName == "name" &&
        node.firstChild.nextSibling != undefined &&
        node.firstChild.nextSibling.nodeName == "age"
    )
        return( true )
    else
        return( false )
}
```

Figure 12 : the full structural test coded against the DOM

Recursive types, shared subgraphs and cycles

The representation of references within XML data is achieved by interpretation over the data content rather than by a defined semantic mechanism within the definition of XML itself. Metadata descriptions do allow the specification of references and, in conjunction with a schema, references may be deterministically identifiable within a data collection. However the DOM is defined only over the XML structure, and therefore handling of references to denote both shared subgraphs and cycles must be achieved by interpretation of these within the application code.

Recursive types are in any case required to capture regular data where references are not explicitly modelled but are represented by nesting. Figure 13 shows some example XML and a Projector program using a recursive typing and algorithm. (In this case we require to use type rule (2a) to deduce the empty array which occurs logically in the child object representations.)

```
<person>
    <name>Richard</name>
    <age>40</age>
    <child>
        <person>
            <name>Thomas</name>
            <age>5</age>
        </person>
    </child>
    <child>
        <person>
            <name>Elizabeth</name>
            <age>1</age>
        </person>
    </child>
</person>

type Person = { name : string, age : int, $array_child : [ Person ] }
type Data = { person : Person }

var listFamily = function( p : Person )
{
    print( p.name )
    for( i in p.$array_child ){ listFamily( p.$array_child[ i ] ) }
}

listFamily( ( project XMLData onto Data).person )
```

Figure 13: use of recursion over nested data

However if the same code is used against data conforming to the DTD given in Figure 8 it will not work properly as the references modelled within the data attributes will not be detected during the type projection. To achieve the same effect, the untyped tree would need to be traversed and the references interpreted before projection onto the simple *Person* type could occur, thus mixing the structural checking code with the application logic. However avoidance of such mixing is the primary intention for the Projector language.

To solve this the observation is required that the dynamic type projection already performs a traversal of the relevant data, and that the requirement is for this traversal to somehow incorporate the semantics of references within that data. This can be achieved by the incorporation of a "reference following" functionality into the projection operation.

To find the DOM nodes representing a person's children requires the following steps:

1. form a list of tokens by extracting the appropriate IDREF from each <child> node

2. form a list to contain DOM node references corresponding to these, with each node initially set to null

3. traverse the entire DOM tree; for each <person> node encountered, extract its ID. If this matches an entry in the list of tokens, then update the corresponding element in the node list.

```
type Child = { $attribute_IDREF : string }
type Person = { $attribute_ID : string, name : string, age : int, $array_child :[ Child ] }

var findIDToken = function( n )
{
    try{ return( ( project n onto Child ).$att_IDREF ) }
    catch( e ){ return( "" ) }
}

var resolveIDToken = function( n, t )
{
    try{
        var p = project n onto Person
        return( p.$att_IDREF == t )
    }
    catch( e ){ return( false ) }
}
```

Figure 14: resolution of references

A solution to this coded in Projector gives rise to the functions shown in Figure 14. Although this will lead to a relatively elegant implementation of the above algorithm, it remains unsatisfactory as the typing of Child and Person captures their implementation rather than the semantics of the model. The solution to this is to perform the algorithm at the time of projection, and allow the type projection to occur over the resulting logical graph, rather than the simple tree of the DOM. This is achieved by an extension of the syntax of project to include generic "find" and "resolve" functions as illustrated in Figure 15.

```
type Person = { name : string, age : int, $array_child : [ Person ] }

var listFamily = function( p : Person )
{
    print( p.name )
    for( i in p.$array_child ){ listFamily( p.$array_child[ i ] ) }
}

listFamily( project XMLData onto Person using findIDToken, resolveIDToken )
```

Figure 15 : type projection over interpreted references

During traversal of the DOM tree, the find and resolve functions will be used wherever appropriate to present a transformed tree to the projection algorithm. The result in this case will be the building of a tree structure corresponding to that of Figure 13, even although the data is presented in a flat list, allowing the recursive algorithm to operate correctly. XML data representing shared subgraphs and cycles are translated into the corresponding JavaScript data structures via type projection.

The find and resolve functions are in general programmer provided, and so the string token passed from one to the other can be used to model arbitrary structures in cases where the reference is resolved by more than a simple token, or when many different types of reference may occur using intersecting sets of tokens; however such cases are probably rare. It is straightforward to generate the functions automatically in cases where a DTD is available, yet the mechanism is also sufficiently flexible to allow other conventions to be coded if unique references are coded in the XML in a non-standard manner.

Related work

Computations over XML data can be specified in a variety of paradigms, models and languages. Two kinds of approaches, however, appear to prevail: dedicated query languages and bindings to programming languages, typically object-oriented ones.

The first resort to regular expressions to match data with irregular or partially known structure (e.g. XML-QL, XQL, LOREL). They also include Turing-complete and/or strongly typed functional languages, which exploit structural regularity to ensure correctness of arbitrary computations (e.g. XQuery, XSLT, Xduce, TeQuyLA).

Language bindings are instead defined by implementing programming interfaces to an in-memory representations of the data. Differences between our and approach and the DOM approach has been largely discussed in the paper, and the main observations can be immediately extended to the SAX approach [SAX].

The novel Sun's JAXB [JAXB] binding model is instead more similar in spirit to ours, in that relies on static type information to preserve the semantics of real-world entities. However, JAXB bindings are automatically defined from XML document descriptions. This complicates their definitions, limits their granularity with respect to the target data, and restricts their ability to evolve within heterogeneous and distributed systems. The Ozone system [OZONE] combines structured and unstructured query approaches to semistructured data.

Implementation status

Projector is a new language specification and at time of writing a full and rigorous implementation does not yet exist. Various partial systems have been built and some are available on the web [IMPL]; it is implemented in, and compiles to, ECMA-262, and so can be executed in a standard browser. Every example given in this paper has been implemented and seen to work.

The projection algorithms themselves are robust and have been extensively investigated, and proofs of soundness and completeness have been performed. Two robust implementations exist and have been used to solve real-world problems; one is CORBA-based and projects via IDL, the other is a Java language version. Anyone interested in using any of these systems should get in touch with the authors.

Conclusions

A new programming language Projector has been introduced largely by motivating examples. The particular paradigm of mixing typed and untyped program segments against XML data looks novel and exciting; however the project is at an early stage and the language has not yet been used "in anger" against real world data collections or problems. There are very many unresolved issues to be investigated.

References and bibliography

[DOM] W3C Document Object Model: http://www.w3.org/DOM/

[ECMA] ECMAScript Language Specification:
http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM

[IMPL] http://www.cis.strath.ac.uk/~richard/typescript/

[JAXB] The Java Architecture for XML Binding: http://java.sun.com/xml/jaxb/

[NAME] W3C XML Namespaces: http://www.w3.org/TR/REC-xml-names/

[OZONE] Tirthankar Lahiri, Serge Abiteboul, Jennifer Widom: Ozone: Integrating Structured and Semistructured Data. DBPL 1999: 297-323

[SCHEMA] W3C XML Schema: http://www.w3.org/XML/Schema

[SNAQUE] Simeoni, Manghi, Lievens, Connor & Neely An Approach to High-Level Lnaguage Bindings to XML Information and Software Technology, 44 (2002) 217 – 228, Elsevier

[SYN] www.cis.strath.ac.uk/~richard/typescript/projector240502/p_syntax.doc

[XQUERY] W3C XML Query: http://www.w3.org/XML/Query