Projector – a Partially Typed Language for Querying XML

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

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

Abstract

We describe Projector, a language that can be used to perform a mixture of typed and untyped computation against data represented in XML. For some problems, notably when the data is unstructured or semistructured, the most desirable programming model is against the tree structure underlying the document. When this tree structure has been used to model regular data structures, then these regular structures themselves are a more desirable programming model. The language Projector, described here in outline, gives both models within a single partially typed algebra and is well suited for hybrid applications, for example when fragments of a known structure are embedded in a document whose overall structure is unknown.

Projector is an extension of ECMA-262 (aka JavaScript), and therefore inherits an untyped DOM interface. To this has been added some static typing and a dynamic projection primitive, which can be used to assert the presence of a regular structure modelled within the XML. If this structure does exist, the data is extracted and presented as a typed value within the programming language.

1. Overview

Our context is standard programming languages used to query data formatted in XML. We are interested in "typeful" programming, by which we mean programming with regular structures used to model categories in the real world. Such structures can be intuitively modelled in XML as labelled trees; however when these trees are presented to the programmer, as in various standards such as DOM [1], SAX [2], XSLT [3] and XQUERY [4], only the structure of the tree itself, rather than the real-world structure it is used to represent, is given as the data model. Any entity model in the mind of the programmer must be explicitly reconstructed through interpretation of the tree structure.

This problem can be overcome by somehow presenting the programmer with a traditionally typed view of the data, which is mechanically reconstructed from the XML format based on metadata or type information. We know of two solutions in this category. JAXB [5] from Sun Microsystems takes a DTD and generates a set of Java classes, along with code to 'marshal' and 'unmarshal' data from the DOM model. SNAQue [6], our own prototype, takes a programming language type and projects it onto the XML data, extracting the largest subset which is correctly described by it.

In this paper, we examine the use of a structural projection mechanism in combination with a tree-based view of the XML data. This gives the effect of adding a structural equivalence type mechanism into the basic untyped framework. Through a scheme of dynamic type projection, we allow structural type assertions to be tested during program execution.

This hybrid approach gives both the versatility of tree-based programming and, when desirable, the ability to switch to a structurally typed programming style. The structural typing can be used to guide the basic tree navigation, and also allows fragments of the 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 [7], and allows an interesting mix of typed and untyped code within a single context.

2. Introduction

XML is increasingly used to model data, rather than documents. Most programming interfaces incorporate a tree-structured view of the XML as an abstraction for the programmer. This is a good abstraction if the data content is inherently irregular, however for regular data it leaves much to be desired. A better approach is to present the programmer with an abstraction corresponding to a more traditional data model.

To illustrate this concept by a simple example, consider Figure 1. The program uses the data constructors of array and record, and their associated dereference operators, to build a model and to access one of its component values.

```
var people = [   { name : ‘Richard’, age : 40 },
                 { name : “Fabio”, age : 30 } ]

print( people[ 0 ].name )
```

Figure 1 : typed data access

Figure 2 shows the same data represented in XML, along with JavaScript DOM code to perform the same dereference. The dereference here is coded over the tree structure of the XML document, rather than over the underlying data model. This is good for documents and unstructured data, but when the data is more highly structured the tree-model introduces an unnecessary complexity into the programming task. Furthermore, the class of error that may be detected by static analysis of the program before binding to the data is greatly reduced. We refer to the tree-based data model as ‘untyped’ as only the type of the tree, rather than the type of the data it represents, is available.

```
<people>
    <person>
        <name>Richard</name>
        <age>40</age>
    </person>
    <person>
        <name>Fabio</name>
        <age>30</age>
    </person>
</people>

var people = XMLDocumentElement
print( people.firstChild.firstChild.firstChild.nodeValue )
```

Figure 2 : tree-based (‘untyped’) data access

The major motivation for our technique of type projection is illustrated by the desire to use the type model-based dereference code in Figure 1 against the data in Figure 2 whenever it is appropriate to do so.

There is a further option of how to achieve this. We distinguish between two possibilities:

- type generation, where a programming language type is obtained by analysis of either the data itself or a metadata description of it, and
- type projection, where the type is taken from the context of the program and matched against the data.

Our interest here is in the second strategy, which has various advantages. Crucially in this context, these include the ability to handle partial data model specifications. This is key in the case where the overall structure of the data is not tightly specified, yet it contains structured ‘islands’ whose structure is known *a priori*. Our key hypothesis is that in such cases it is valuable to mix the paradigms of typed and untyped programmatic access to the data.

2.1. Mixing Typed and Untyped Programming

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 an untyped programming discipline, 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 we will describe gives several advantages over the basic DOM abstraction. Primarily, these correspond to the normal advantages of typed vs. untyped programming environments. Furthermore, the descriptive power of the type system can be used to handle abstractions corresponding to cycles and shared subgraph components which are present within the XML. While these are, by necessity, handled through interpretation, this occurs 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 it being used primarily as a static mechanism to ensure the partial safety of programs, it is also used as a shorthand within certain dynamic computations 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.

2.2. Static typing in 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. 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 3 shows some essential features of the extended language. The two functions *getName* and *getName2* both return the name of their argument *p*, which is expected to represent a person. Both function bodies are standard JavaScript. However, the typed version is checked statically and the function body is therefore guaranteed to behave correctly. The untyped version instead has a series of dynamic checks 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" ) }
}
```

Figure 3 : structure checking by type and by algorithm

At a call to *getName*, a static error will occur if the argument cannot be deduced to be of the appropriate structure, whereas at a call to *getName2* no such restriction is imposed. However, total static checking is not possible when some of the data is imported into the context during execution, and so the statically typed function cannot be called directly under these circumstances. A type projection primitive is therefore introduced, which is a static assertion of the structure of a value only available dynamically.

If the assertion is correct no action is taken; if it is incorrect an exception is raised. This allows the target expression to be safely statically retyped for the remainder of the execution block. Figure 4 illustrates the use of a projection expression to allow a statically verified call to *getName* with an argument whose type is not statically known. If the argument *u* turns out to have an incorrect structure, the function call will not occur.

```
var showName = function( u )
{
    try{ alert( getName( project u as person ) ) }
    catch( e ){ alert( 'showName: u is not a person' ) }
}
```

Figure 4 : statically typed call using projection

Notice that the dynamic behaviour of the projection expression is exactly the same checking code that appears at the head of the *getName2* function. However, as well as significantly better succinctness of expression, the use of the projection primitive also allows the static retyping of the expression.

So far we have talked only about core JavaScript language values. In the next section we will consider

[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.

a similar mechanism applied over DOM data structures.

2.3. 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 given in Figure 5.

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

Figure 5 : example trivial DTD

It is important to point out that the type projection scheme directly relates the programming language type and the data itself, and does not require any involvement with DTD or XMLSchema metadata. The DTD is used here only to explain the structure of the envisaged data, and there is no mechanical requirement for it to exist.

A Projector function to calculate the average age of any such collection may be written as in Figure 6.

The *Collection* type describes a JavaScript model of the expected structure of the whole XML document. The document element, *people*, is described as a record with a single field of this name. The type of this field is a record, with a single field *array_person*. The syntactic form of this label reflects the fact that a uniformly typed collection within JavaScript is typically modelled as an array, whereas within XML is signified by a repeated element label. When mapping between these formats, the mechanically significant prefix *array_* is used to maintain human readability. The type of the array elements, a record with scalar name and age fields, corresponds to the scalar fields as defined in the DTD[3].

The *XMLProject* keyword signifies a static assertion of the expected structure of the function parameter *domData*, which should be a DOM tree conforming to the DTD given in Figure 5. 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*. The significance of this is that the code in the rest of the block 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( domData ) {
    var count = 0; var total = 0

    var data = XMLProject domData as Collection

    var people = data.people.array_person
    for( i in people ){ count++; total += people[ i ].age }
    return( total / count )
}
```

Figure 6 : 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 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.

3. The projection mechanism

We have described type projection schemes for semistructured data in detail elsewhere in more formal terms [6], and give a simple and relatively informal overview here.

The projection mechanism is based upon a subtype relation between the 'high level' type contained in the program, and a 'low level' type assigned to the XML data collection. If the XML type is a subtype of the high-level type, then the projection of this high-level type onto the data corresponds, in database terms, to a typed view onto the data.

There is however some tension between type systems for these two domains. For example, a type system for the document model would include the concept of ordering of elements, and would allow repeated element names within a single scope. On the other hand, a type system for a programming language will include higher-level data constructors such as arrays,

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

abstractions such as unions, and other higher-level concepts.

In general, we avoid these problems by considering a single, hybrid type system which contains sub-languages suitable for both domains. In this description, we define type languages *T*, the hybrid system, which contains languages *PL* (the programming language subset of types) and *SS* (the semistructured language subset of types). We define a meaning for *PL* within the programming language, and a meaning for the whole of *T* (rather than *SS*) within the semistructured domain.

Given this semantic framework, we can demonstrate the soundness of a subtype relation over *T* in terms of the semistructured data. Furthermore, this implies that if we can accordingly relate an expression in *SS* with one in *T*, then we also have a mechanism for interpreting the corresponding subset of the semistructured data as a value within the programming language.

The strategy for performing this typecheck is essentially to start with the type assigned to the data and continually rewrite it, trying to achieve the projection type. As the type assigned to the data is isomorphic to the data itself, in reality we can operate over the data rather than explicitly generating the assigned type. This is the mechanism by which the data extraction is performed in parallel with typechecking. For this reason we give the subtype relation as a set of rewrite rules, rather than in a more conventional notation.

3.1. Semi-formal definition

In the context of JavaScript, an appropriate set of type languages is shown in Figure 7. The curly and square brackets in *PL* are syntactic forms representing JavaScript object and array type constructors. No union type is included as the JavaScript untyped model makes this unnecessary for our purpose. One unusual aspect of the grammar *SS* is that repeated label names are allowed within object types; when the normal restriction of non-repetition is imposed, the *PL* subset is derived.

$\mathbf{T} ::= \mathbf{PL} \mid \mathbf{SS}$

$\mathbf{PL} ::= \mathbf{scalar} \mid \{\, l_1 : \mathbf{PL}, \ldots, l_n : \mathbf{PL} \,\} \mid [\, \mathbf{PL} \,] \qquad (\, l_i \neq l_j \,)$

$\mathbf{SS} ::= \mathbf{scalar} \mid \{\, l_1 : \mathbf{SS}, \ldots, l_n : \mathbf{SS} \,\}$

$\mathbf{scalar} ::= \text{int} \mid \text{string} \mid \text{bool}$

Figure 7 : A type grammar 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* 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 reserved label prefix *attribute_* and the reserved label *mixedContent* as shown by example in Figure 8.

```
<person xmlns="person.richard.cis.strath.ac.uk">
    <name>Richard</name>
    <age>40</age>
    <motto>XML doesn’t care.</motto>
    <motto><i>Never</i> mix content.</motto>
</person>

    :

{ person : { attribute_xmlns : string,
             name : string,
             age : int,
             motto : string,
             motto : { i : string, mixedContent : string } } }
```

Figure 8 : an example type assignment

The subtype relation is based on the following semantic interpretation of the type grammar *T* 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.

At this point it is worth mentioning ordering. The interpretation above is driven mainly by the programming language type system, as we require to present the result of the projection within this semantic domain. This will not normally be a perfect match as mentioned above, but as long as the translation rules are clear to the programmer this should not pose a problem. In the JavaScript translation, there is no concept of ordering of labels within a record, and so this aspect of the source data is lost. This loss of ordering does not preclude the extraction of order from the original data, but this must occur in the untyped view of the data. In the case of arrays, which are ordered data structures, the relative ordering of components will be maintained by the system.

The subtype relation is informally defined by the rewrite rules given in Figure 9, 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.

(1: record subtyping)
$\{ l_1 : T_1, \dots , l_m : T_m , \dots , l_n : T_n \} \Rightarrow \{ l_1 : T_1, \dots , l_n : T_n \}$

(2: array introduction)
$\{ l_1 : T_1, \dots , l_m : T_m , \dots , l_n : T_n \} \Rightarrow$
$\{ l_1 : T_1, \dots , \text{array\_}l_m : [T_m] , \dots , l_n : T_n \}$

(3: array assimilation)
$\{ l_1 : T_1, \dots , \text{array\_}l_m : [T_m] , \dots , l_m : T_m , \dots, l_n : T_n \} \Rightarrow$
$\{ l_1 : T_1, \dots , \text{array\_}l_m : [T_m] , \dots , l_n : T_n \}$

(4-5: scalar widening)
int ⇒ string bool ⇒ string

Figure 9 : The subtype relation expressed by rewrite rules

The rules given are not formally exhaustive but informally give the main axioms of subtyping. 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 an 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 (4-5) are just an admission that the eager typing of scalar values according to their microstructure does not necessarily reflect their intended meaning.

A variant of rule (2), which will be used later in the paper, is given in Figure 10. 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.

(2a : empty array introduction)
$\{ l_1 : T_1, \dots , l_n : T_n \} \Rightarrow$
$\{ l_1 : T_1, \dots , l_n : T_n, \text{array\_}l_p : [T_p] \}$

Figure 10 : introduction of empty arrays

4. *Mixed mode programming*

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 example data model to that given by the DTD in Figure 11.

```
<!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 11 : 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 11 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.

Generic code to calculate the average age of all person records embedded in any data source can be written using JavaScript first class functions against the standard DOM model as in Figure 12. This code abstracts the details of the functions *isPerson* and *getAge*, which respectively test for whether a DOM node represents a person, and return the age field of such a node..

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

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

Figure 12 : generic traveral of the DOM

Minimal JavaScript code for the two functions *isPerson* and *getAge* is shown in Figure 13. 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
{
    return( node.namespaceURI ==
            "person.richard.cis.strath.ac.uk" )
}

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

Figure 13 : “dynamically typed” DOM code

Figure 14 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 13, 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 15. 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 = XMLProject node as 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
{
    return( ( XMLproject node as Person ).age )
}
```

Figure 14 : the same example expressed in Projector

```
function isPerson()
{
    return(
        node!= undefined &&
        node.namespaceURI =
                    "person.richard.cis.strath.ac.uk " &&
        node.firstChild != undefined &&
        node.firstChild.nodeName == "name" &&
        node.firstChild.nextSibling != undefined &&
        node.firstChild.nextSibling.nodeName == "age"
    )
}
```

Figure 15 : structural test coded against the DOM

5. *Recursive types, shared subgraphs and cycles*

Recursive types are required to capture regular data structures. In XML such structures can be modelled either by implicit nesting of a common substructure or, more commonly, by interpreted references. Figure 16 shows some example XML with nested instances of a person representation, and a Projector program using a recursive type to describe them. In this case we require to use type rule (2a) to deduce the empty array which occurs logically in the child object representations. The move from non-recursive to recursive types is highly significant in terms of the underlying type theory, but does not present any significant new challenges in this context.

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 internal 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.

Other more sophisticated mechanisms for encoding both inter- and intra-document pointers are emerging; however it is not clear that the tree-based data structures representing such documents will support these. The jump from supporting tree-based to general graph-based traversals is extremely significant and will not be undertaken lightly. It therefore seems likely that all related computing paradigms will continue to require interpretation of such references at the application level for the foreseeable future.

```
<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( ( XMLProject domData as Data).person )
```

Figure 16 : use of recursion over nested data

However if the same code is used against data conforming to the DTD given in Figure 11 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{
        var c = XMLProject n as Child
        return( c.attribute_IDREF )
    }
    catch( e ){ return( "" ) }
}

var resolveIDToken = function( n, t )
{
    try{
        var p = XMLProject n as Person
        return( p.attribute_IDREF == t )
    }
    catch( e ){ return( false ) }
}
```

Figure 17 : resolution of references

A solution to this coded in Projector gives rise to the functions shown in Figure 17. 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 18.

```
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 ] )
    }
}

var p = XMLProject domData as Person using
                              findIDToken, resolveIDToken
listFamily( p )
```

Figure 18 : 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 16, 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. It is possible 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.

6. 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 [8], XQL [9], Lorel [10]). They also include Turing-complete and/or strongly typed functional languages, which exploit structural regularity to ensure correctness of arbitrary computations (e.g. XQUERY [11], XSLT [3], XDuce [12], TeQuyLA [13], and TQL [14]). All of these approaches are primarily designed for working against unstructured or semistructured data, and therefore suffer from the basic problem of the programmer being required to reconstruct higher-level semantic data models whenever these are encoded in the underlying tree format. Some have their own weak type systems, and notably those of XDuce and TQL bear some resemblance in their underlying structures to those we use. However, we are more interested in moving fluidly between completely untyped (tree-structured) and strongly typed programming modes.

Language bindings are instead defined by implementing programming interfaces to a structured representation 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 [2].

Sun Microsystems have introduced the JAXB [5] Java to XML binding model, which is similar in spirit to ours in that it relies on static type information to preserve the semantics of real-world entities. However, JAXB bindings are automatically generated from DTD document descriptions. This forces generation of typed structure to match the entire documents description, which must be known a priori as validation is required to preserve soundness. This limits its granularity with respect to the target data, and restricts its ability to evolve within heterogeneous and distributed systems.

Other type-related support is available by use of the XML namespace (xmlns) standard [15], in conjunction with XML data validated with respect to DTD or XMLSchema [16] metadata. The combination can allow 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. 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.

7. 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 [17]; it is implemented in, and compiles to, ECMA-262, and so can be executed in a standard browser.

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.

8. 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.

9. Acknowledgements

This work has been financially supported by EPSRC (GR/M 72265) and BBSRC (17/BIO 12052). The work has further benefited from discussions with Prof Al Dearle of St Andrews University and Dr Miles Whitehead of Reuters Ltd. David Lievens is supported by EU Framework V project GLOSS (EU IST-2000-26070), and George Russell is supported by a University of Strathclyde PhD studentship.

10. References and bibliography

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

[2] SAX: http://www.saxproject.org/

[3] XSLT: http://www.w3.org/TR/xslt

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

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

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

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

[8] XML-QL: http://www.w3.org/TR/NOTE-xml-ql/

[9] XQL: http://www.w3.org/TandS/QL/QL98/pp/xql.html

[10] Lorel: http://www-db.stanford.edu/lore/

[11] XQuery: http://www.w3.org/TR/xquery/

[12] XDuce: http://xduce.sourceforge.net/

[13] A. Albano, D. Colazzo, G. Ghelli, P. Manghi, C. Sartiani *A Type System for Querying XML documents* ACM SIGIR 2000 Workshop On XML and Information Retrieval, Athens, Greece 2000.

[14] Tree Query Language: http://tql.di.unipi.it/tql/

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

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

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

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