Hybrid Applications over XML: Integrating the Procedural and Declarative Approaches

Paolo Manghi[†] Fabio Simeoni[‡] David Lievens[‡] Richard Connor[‡]

[‡]Department of Computer Science
University of Strathclyde, Glasgow (Scotland)

[†]Dipartimento di Informatica
Università di Pisa, Pisa (Italy)

Abstract

We discuss the design of a quasi-statically typed language for XML in which data may be associated with different structures and different algebras in different scopes, whilst preserving identity. In *declarative scopes*, data are trees and may be queried with the full flexibility associated with XML query algebras. In *procedural scopes*, data have more conventional structures, such as records and sets, and can be manipulated with the constructs normally found in mainstream languages.

For its original form of structural polymorphism, the language offers integrated support for the development of *hybrid applications* over XML, where data change form to reflect programming expectations and enable their enforcement.

1 Introduction

To date, programming over XML is essentially programming over labelled trees, according to the standard interpretation of the format [7]. This can be done in a procedural algebra, such as DOM's [8], or in the declarative algebras of dedicated query languages, XQuery's before others [3].

Based on powerful path expressions, declarative algebras offer unrivalled flexibility for retrieving and transforming the data, in the spirit of query languages. They may also offer computational completeness and, as in the case of XQuery, an appropriate notion of static typing. On the other hand, procedural algebras fit into a well known computational model inclusive of update. Embedded in mainstream languages via language-specific bindings, they also promise full integration with existing computational facilities (e.g. I/O, user interfaces, legacy code), a large user base, as well as proven and familiar development tools.

For their different qualities, the two approaches are complementary and would integrate well within, say, the same object-oriented language. In spite of their flexibility, however, labelled tree structures cannot be expected to be adequate choices for all computational tasks. More familiar programming abstractions - such as pairs, tuples, records, sets, relations - may better reflect the required interpretation of the data. In fact, the view of XML as a universal format for the exchange of data suggests that large part of that data will originate in standard programming languages and database systems. How many employees live within programs and databases as labelled trees?

1.1 XML and Data Structures

Ignoring for a moment issues of efficiency, inadequate data structures induce linguistic problems: programs become soon harder to write, read, and thus maintain. As a simple example, consider an employee record `e` materialised at the receiver as a labelled tree. The relationship between the employee and its name becomes one between two nodes and their labels, whereas it was originally one between a record and its field `name`. The sender accesses the name by writing the expression:

```
   e.name
```

whereas the receiver using, say, a DOM implementation must resort to something like:

```
NodeList children = e.getChildNodes();
  for (int j=0;j<children.getLength();j++) {
   Node child = children.item(j);
   if (child.getNodeType() == Node.ELEMENT_NODE) {
     String tagName=((Element) child).getTagName();
     if (tagName.equals("name")) ...
       ((characterData) child.getFirstChild()).getData()...}}
```

Of course language specific APIs may better tune the algebra to the particular host language (e.g. [13]). Similarly, a stronger orientation towards data (as opposed to documents) may also help to ease the linguistic problem. It should be clear, however, that the tree structure does not directly reflect the data semantics required by the receiver and, in this case, that originally intended by the sender. When the receiver is statically typed, the problem is further aggravated by a loss of safety: `name` was meta-data within the record structure and has become data within the tree structure. As such, it escapes the static knowledge of the system and its correct use may not be detected before program execution or, worse, not detected at all (e.g. when a misspelled label accidentally identifies another).

Declarative algebras alleviate partly the problem by hiding the tree structure under a navigational syntax, thereby achieving succinct and clear programs. The choice of data structures, however, extends its impact on semantics: as the procedural XML programmer, the declarative one is still forced to perceive employees as trees in spite of more intuitive models of the data.

It should also be noted that the problem here is not related to labelled trees, which remain the preferred structures for a variety of computational tasks: document manipulation, semistructured data management (cf. [11]), structural queries, flexible browsing, etc. Similar problems would surface with any structure imposed by the wire format and, in general, with any incarnation of the one-size-fits-all approach to data modelling.

The example of the SAX programming model and algebra is here appropriate [9]. With SAX, the XML data is a string served to the programmer as a temporal sequence of tokens and the induced programming model is based on callbacks for parsing events. Compared with DOM, SAX makes it easier to specify computations that interpret the data as the text in which it is encoded (e.g. token counts, deep queries, etc.). Nonetheless, it is easy to see that the linguistic and safety problems associated with DOM computations surface unsolved for SAX programmers.

1.2 A Language for Hybrid Applications

Motivated by the previous observations, we advocate the importance of applications in which the same data are subject to different structural views and are manipulated according to different algebras in order to facilitate different programming tasks. For example, some components of such *hybrid applications*, as we may call them, may benefit from a tree view over the data and from the flexibility of a query algebra. Other components may instead be safer and simpler by, say, a view based on record and set abstractions and their associated algebras.

In practice, hybrid applications are common and yet the integration of their components is essentially unsupported. XML programmers must associate components with different tools and computational environments (e.g. a mainstream programming language and a query engine) and share data between them through the the file system or the network. This forces interactions between components to be 'off-line' (i.e. planned in advance of execution) and strictly sequential (a component's output becomes another's input). Lack of integration becomes also lack of efficiency, due to the unnecessary operations of input/output and parsing of the data. Updates occur only at the file level while sharing between components require ad-hoc conversions between data structures which are prone to errors and always irrelevant to application semantics. Overall, the XML programmer is entirely responsible for understanding and maintaining the mapping between the application design and its scattered implementation.

In our research, we explore the possibility of writing hybrid applications within the context of a single programming language, where the imposition of structure over the data is transparent and entirely under the programmer's control. In this paper, in particular, we experiment with two different interpretations of the data and associate them with different scopes in the program. In *declarative scopes*, the data are labelled trees and can be manipulated with a simple query algebra based on XPath expressions [2]. In *procedural scopes*, the structure of the data is a recursive composition of records and sets and can be manipulated with conventional algebras. In the latter case, programmers may also count on a selection of the basic types and programming constructs found in most procedural languages.

Programs can then be partitioned according to the view which is syntactically in scope, with data changing form upon entering and exiting scopes whilst preserving identity. The passage to a declarative scope is straightforward, for the data can always be interpreted as a labelled tree. Different is the opposite case, when more constrained structures must be *projected* over trees with the possibility of failure.

We solve these problems by resorting to a *quasi-statically typed* language and interpreting structural projections as type assertions attached to program variables. Noticeably, type assertions are verified dynamically, when variables are bound to trees, and yet their scope within the program can be statically typechecked. Accordingly, the approach extends the practice of dynamic typing within otherwise statically typed programming languages (cf. [1]). In particular, it admits data which is completely untyped and yet sufficiently *self-describing* to allow type-checking.

The rest of the paper is organised as follows. Section 2 introduces our model for type projections while Section 3 discusses the language design by way of example. Finally, Section 5 draws some conclusions and outlines further work.

2 Type Projections

We have successfully investigated the problem of type projections over labelled tree data in `SNAQue`, an architecture for binding quasi-statically typed programming languages to XML data which emanate from outside their context. `SNAQue` is formally defined in [4, 5], while an implementation specific to the Java language is discussed in [10], where it is also thoroughly compared with related approaches, such as JAXB [12].

With `SNAQue`, the programmer projects a type over an XML file in an attempt to materialise the content of the latter as a value of the former, and thus derive the benefits discussed in Section 1. This requires parsing the file into a temporary tree structure and establishing whether the tree is an encoding of a value of the projected type, according to a pre-defined mapping between language values and labelled trees. If this is the case, the value is materialised from the tree and may be subject to application-specific programming, otherwise an indication of failure is returned to the programmer.

For generality, we have studied type projections in the context of a *canonical language* defined around a value notation, a type language, and a relationship of typing between the two. In particular, we have chosen a selection of structural types commonly found in existing procedural languages: built from a set of atomic types, they include include *record*, *collection*, and *untagged union* types, possibly recursively defined. Specifically, a type T is one of a finite set of atomic types B_i, a record type $[l_1 : T_1, \ldots, l_n : T_n]$, a collection type $coll(T)$, a union type $T_1 + T_2$, or a recursive type $\mu X.T$, where

X is a type variable and the operator μ binds occurrences of X in T[1].

Canonical values mirror the available types. A value v is an atomic value $b_k \in B_k$, a record value $[l_1 = v_1, \ldots, l_n = v_n]$, a collection value $\{v_1, \ldots, v_n\}$, or the empty collection $\{\}$. The typing relation is inductively defined in a standard fashion. An atomic value b_k has the corresponding type B_k, while a record $[l_1 = v_1, \ldots, l_n = v_n]$ has the type $[l_1 : T_1, \ldots, l_n : T_n]$ only if each v_i has type T_i. A collection $\{v_1, v_2, \ldots, v_n\}$ has the type $coll(T)$ only if all the v_i have type T, while the empty collection has the type $coll(T)$ for all T. A value v has type $T_1 + T_2$ if v has type T_1 or type T_2 and, finally, v has type $\mu X.T$ if v has the type obtained by substituting $\mu X.T$ for all the bound occurrences of X in T.

For parsing purposes, we have considered a tree interpretation of XML data which abstracts over the document-oriented features of the format (e.g. ordering, processing instructions, etc.) and concentrates on the data-oriented features (e.g. naming and nesting)[2]. Figure 1 gives a visual example of the tree interpretation of a sample XML document.

```
<store>
  <name> BooksRus </name>
  <book>
    <author> John Backus </author>
    <author> Peter Naur </author>
    <title>XML Does Not Care </title>
  </book>
  <book>
    <author>
        <fname> Stan </fname>
        <sname> Lee </sname>
    </author>
    <title> The Annotated Spiderman </title>
  </book>
  <book>
    <title> The Bible </title>
  </book>
</store>
```

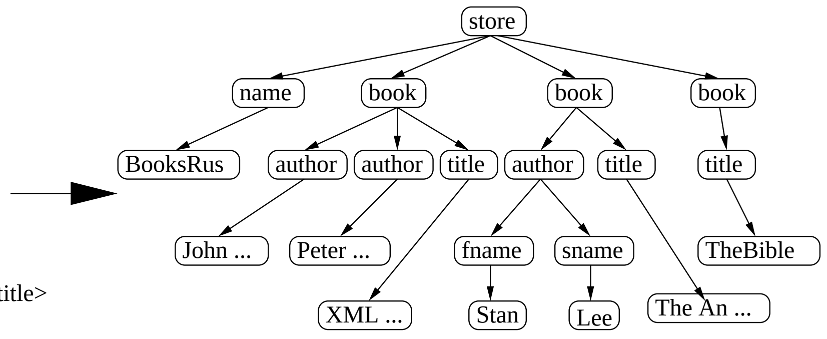

Figure 1: XML Parsing

The tree encoding of values is illustrated by way of example in Figure 2, which shows the tree corresponding to the record v=`[a=1,b={"two,"three"},c=[d="four",e=5],f={}]`. Essentially, atomic values are encoded as leaf nodes, while the encoding of record and collection values is built on that of their fields and elements, respectively. Since language values are anonymous, however, it is not immediately clear what labels should be used at the root nodes. For atomic values this is a textual encoding of the values themselves, but for records and collections our choice is that the label may only be provided by the context.

In Figure 2, a context for the entire v is missing and this explains the omitted label at the root node. Different is the case for, say, the root of the tree that encodes `[d="four",e=5]`, which is labelled with the field name `c` of v[3]. Finally, observe the encodings of the values in `{"two,"three"}` – which take their label from the record field `b` – and the encoding of the value of field `f` – which extends the previous rule to the extreme case of the empty collection.

In terms of type projections, the implications of the encoding scheme are essentially two: 1) tags of root elements such as `store` in Figure 1 are irrelevant, and 2) collection types may only be successfully projected within record types, where they never fail. For example, the projection of type *Store* over the data in Figure 1, where:

[1]Recursive types do not need to appear such theoretical guise within program, but can be derived from self-referencing type declarations.

[2]In practice, element attributes may be parsed as subelements.

[3]In previous work, we adopted a more convenient tree model where labels are on edges rather than nodes. We have here maintained labels on nodes to rely later on standard query semantics (see Section 3).

Store = [name:string,book:coll([title:string,author:coll(*Author*)]
Author = string+[fname:string,sname:string]

would result in the language value *store*, where:

store = [name="BookRus",book={*book*1,*book*2,*book*3}]
*book*1 = [title="XML Does Not Care",author={"John Backus","Peter Naur"}]
*book*2 = [title="The Annotated Spiderman",author={[fname="Stan",sname="Lee"]}]
*book*3 = [title="The Bible",authors={}]

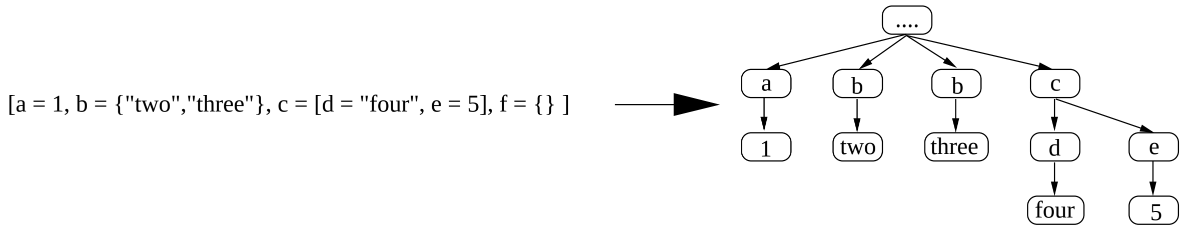
[a = 1, b = {"two","three"}, c = [d = "four", e = 5], f = { }]
....
a
b
b
c
1
two
three
d
e
four
5

Figure 2: Encoding example

3 Programs and Queries

Essentially, this work originates from the hypothesis of moving SNAQue bindings and tree data within the language, rather than at its boundary with the file system. At this early stage of investigation, our aim is not to present a complete language, rather to explore design options. In particular, we use a pseudo-syntax to illustrate a possible extension of the canonical language with value operators (primitive operators, record and collection algebras, etc.), standard procedural constructs (e.g. type declarations, static scoping, assignments, functional abstractions, control structures, etc.), and library support (e.g. facilities for access to relational data). As a first example, the following function addBook augments a collection of books:

```
type Book = [isbn:string,price:number]

fun void addBook(Book book, coll(Book) booklist){
  for b in booklist
    if (b.isbn = book.isbn) remove b from booklist
  add book to booklist
}
```

The code should be self-explaining. In line with the assumption that data preserves identity across scopes, addBook is applied under by-reference semantics. The new type void has the conventional meaning while the operators for-in, add-to, and remove-from form the bulk of the collection algebra. Type assertions are statically enforced and, though the issue will not arise, type equivalence may be assumed to be structural to match the nature of type projections.

We then introduce XML in the language in the form of tree values and combine it with a query algebra built on XPath expressions into a second form of functional abstraction. For example, the following query findBooks takes an XML representation of an online bookshop's catalog and returns the ISBN number and price of all the books scattered in the data:

```
query findBooks(catalog) {
  for book in catalog//book union catalog/reviews/entry
  return {
    <book>
```

```
      book/isbn
      book/price
    </book>}
}
```

The query semantics is standard, and in any case its details have little relevance in this context, where the focus is on the integration of programming paradigms. Accordingly, we assume the reader's familiarity with XPath and FLWR-like expressions and point out only that the query processor embeds the sequence of `book` elements produced by the `return` clause in a system-defined `ROOT` element, so as to return well-formed XML, and thus trees, to the invoking scope.

This brings us to query invocations, which represent boundaries between procedural and declarative scopes and thus the points of polymorphic behaviour of data. Specifically, actual parameters are encoded as trees before binding to formal variables while return values may be implicitly projected over via assignments to typed variables. This is illustrated by the next example, where two trees enter the same procedural scope. One is parsed from a local file (via the primitive operation `read`) but exits the scope immediately as the actual parameter of an invocation of `findBooks`. The other returns from the query and remains in scope with a record form before being written back to file (via the primitive operation `write`):

```
type BookList [book:coll(Book)]
...
BookList books := findBooks(read("catalog.xml"))
...
write(books,"books.xml")
```

The example captures the essence of the language, for it shows that assignments may entail type projections, implicitly. Here, the type `BookList` is projected over the tree output of `findBooks` with the intention of binding the resulting record to the variable `books`. The model outlined in Section 2 and the body of `findBooks` ensure that the projection is successful. Had it not been, an error would be returned to the programmer, such as a `null` value (of type `void`) or some kind of exception.

As a larger example, consider the case of a simple hybrid application that selects the cheapest price for a list of books (cf. Figure 3). Assume that the list is scattered in a relational database and comprises the recommended books for the courses offered by a university department. Suppose also that two functions `getRecBooks` and `updateRecBooks` are available for accessing the course database and, respectively, retrieve and update the list of recommended books. Finally, assume that information on a list of online stores is locally available in an XML file `"stores.xml"`.

The application is very simple and the reader should compare it with the solutions currently available. First, the list of recommended books is retrieved from the database as a collection value `recList`. Book prices in `recList` are then initialised to some large constant to ensure that an update will actually take place.

Similarly, the store list is read from`"stores.xml"` and assigned to variable `stores`. This entails the projection of type `StoreList`, which asserts that, for each store, the file contains a name and the URL of a corresponding XML catalog. Assuming the assertion to be correct, the catalog of each store is fetched from its URL (with an overloaded version of `read`) and passed to a query `getLowerPrices`. The latter receives also the `recList`, after its value has been encoded as a tree upon entering the declarative scope. This offers an example of a conventional language value that changes its form to be queried.

In particular, the query filters the books in the catalog which are recommended and currently offered at a lower price. When returned to the procedural scope within the record value of `catalog`, the new offers are reflected in `recList` with repeated invocations of the function `addBook` defined above. After all the catalogs have been so processed, `recList` is eventually written back to the database.

```
type Store = [name:string,catURL:string]
type StoreList = [store:coll(Store)]

// database access
fun coll(Book) geRecBooks() {...}
fun void updateRecBooks (coll(Book) rl) {...}

query getLowerPrices(list, catalog) {
  for b1 in list, b2 in catalog//book
  where b1/isbn = b2/isbn and b2/price < b1/price
  return {
    <book>
      b2/isbn
      b2/price
    </book>
 }

val coll(Book) recList := getRecBooks()

for b in recList
  b.price := MAX_PRICE

val StoreList stores := getStores(read("stores.xml"))

for s in stores.store {
  val BookList catalog = getLowerPrices(recList,read(s.catURL))
  for b in catalog.book
    addBook(b,recList)
 }

updateRecBooks(recList)
```

Figure 3: A Simple Hybrid Application

4 Conclusions

We have discussed the design of a language for hybrid applications over XML. This requires a form of structural polymorphism whereby data assume the form of a tree or that of more conventional language values in order to adequately support programming expectations. By associating a query algebra to the tree view and an imperative algebra to the procedural view of the data, the language offers the best of declarative and procedural approaches to XML programmers.

At this stage of investigation, our interest is in experimenting with design options. Due to the novelty of the approach, it is not yet clear what language features would best support hybrid applications. Here, we can only hint at two lines of further development.

Our first concern is for *partial type projections*, whereby types are allowed to match subsets of the data. This is already a feature of `SNAQue`, where it allows bindings to disciplined subsets of particularly semistructured data, minimality of specifications, and program resilience to irrelevant changes in the external data (cf. [10]).

Not only do the latter reasons reveal the strong analogy between partial projections and conventional notions of record subtyping, and thus suggest they should coexist in the language. They also introduce similar problems of update which are well-known in the literature (cf. [6]). Among the available solutions, we consider the static advantages of bounded universal quantification for record subtyping and investigate novel schemes for partial type projections which exploit their inherent dynamicity.

Finally, we plan to extend type projections to graphs, and thus face the problems raised by the presence of cycles and sharing in the data. The first concern the termination of algorithms and appear to push the formalisation into a co-inductive framework. The second are exactly those associated with partial projections and will require a uniform solution.

References

[1] M. Abadi, L. Cardelli, B.C. Pierce, and D. Plotkin. Dynamic Typing in a Statically-typed Language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

[2] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. Technical report, World Wide Web Consortium, April 2002. W3C Recommendation.

[3] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, apr 2002. W3C Working Draft.

[4] R.C.H. Connor, D. Lievens, P. Manghi, S. Neely, and F. Simeoni. Extracting Typed Values from XML Databases. *OOPSLA'01 Workshop on Objects, <XML> and Databases, Tampa Bay, Florida, USA*, October 2001.

[5] R.C.H. Connor, D. Lievens, P. Manghi, S. Neely, and F. Simeoni. An Approach to High-Level Language Bindings for XML. *Elsevier Journal on Information and Software Technology, Special Issue on Object, XML and Databases*, 44 (a):217–228, 2002.

[6] Richard C.H. Connor. *Types and Polymorphism in Persistent Programming Systems.* PhD thesis, University of St. Andrews, UK, 1990.

[7] John Cowan. XML Information Set. Technical report, World Wide Web Consortium, October 2001. W3C Working Draft.

[8] Document object model (DOM). http://www.w3.org/DOM.

[9] Megginson Technologies Ltd. SAX 2.0.1: The Simple API for XML, 2002. http://www.saxproject.org/.

[10] Connor R.C.H. Lievens D. Manghi P. and Simeoni F. Language Bindings to XML. *Submitted to IEEE Journal on Internet Computing*, 2001. Contact Author, Fabio Simeoni: Fabio.Simeoni@cis.strath.ac.uk.

[11] Y. Papakonstantinou, J. Widom, and H.G. Molina. Object Exchange Across Heterogeneous Information Sources. *Proceedings of IEEE Int. Conference on Data Engineering, Birmingham, England*, 1996.

[12] Inc. Sun Microsystems. Java Architecture for XML Binding (JAXB), 2001. http://java.sun.com/xml/downloads/jaxb.html.

[13] The JDOM Project. Jdom beta 8, 2002. http://www.jdom.org.