THE DUALITY OF XML MARKUP AND PROGRAMMING NOTATION

Kurt Nørmark
Department of Computer Science,
Aalborg University, Denmark.
normark@cs.auc.dk

ABSTRACT

In web projects it is often necessary to mix XML notation and program notation in a single document or program. In mono-lingual situations, the XML notation is either subsumed in the program or the program notation is subsumed in the XML document. As an introduction we analyze XML notation and programming notation in relation to each other. As the main contribution of the paper we describe a number of general issues to consider when subsuming XML in a given programming language.

KEYWORDS

XML, program notation, web programming.

1. INTRODUCTION

In almost any use of XML there is a need to involve some kind of programming. The transformation of XML to other formats is a dominant example. The target format is often, but not necessarily HTML. Tools that generate static web contents from XML sources represent concrete examples. Programs that implement web services form other examples.

In the situations mentioned above, we find it relevant and interesting to analyze how program fragments and XML document fragments are mixed and integrated in a single document, or in a single program. Overall, the following types of combinations are possible:

• **Markup hosting**. Within the scope of a complete XML document, pieces of programs appear as constituents. The program fragments are typically located in distinguished tags, such as <% ... %>, or in brackets or braces.

• **Program hosting**. Within the scope of a complete program, pieces of XML markup appear as constituents. At the most primitive level, the XML fragments appear as text strings in print statements or in similar contexts.

• **Markup subsumption**. The programming notation is subsumed in XML. A special interpreter must exist to process these XML fragments. With this it is possible to formulate programmatic solutions, such as document transformations, using plain XML notation.

• **Program subsumption**. The XML notation is subsumed in the programming language. The XML aspects, as well as other aspects, are processed by an interpreter of the programming language. The subsumption of XML notation is done on the premises of the programming language. No verbatim XML fragments occur as part of the program.

Markup hosting is known from a number of web server programming frameworks, such as ASP, JSP, PHP, and BRL [8] (in Visual Basic, Java, the PHP language, and Scheme resp.) Using these frameworks, program fragments and XML fragments are located side by side in a single document. Program hosting is also in common use. CGI programs fall within this category. In imperative programs, many XML fragments appear as text strings in print statements. The mixing of programming notation and XML notation raises both aesthetic and technical concerns. Aesthetically, a fine-grained mixing of the notations leads to confusing source documents that are difficult to read and understand. The main reason is the clash of two entirely different syntactic cultures (XML and C, for instance). From a technical point of view, the semantic rules of the programming language are blurred due to the fragmentation and the embedding of the program pieces in a foreign surround.

There are several examples of markup subsumption related to XML. XSLT is a widespread transformation framework, which uses XML for program transformation notation [1]. XEXPR [9] is a proposed XML scripting notation inspired from Lisp. Seen as a source program notation, XML is both verbose and clumsy. Although many different kinds of programming notations have be proposed during the years, the XML style of markup seems to run counter to the succinct notation, preferred by most programmers.

Program subsumption is known from several functional and object-oriented languages, such as Haskell [14], Scheme [7,12], and Java [6]. New programming languages have also been built for such purposes, such as XDuce [2] and to some degree Curl [3]. The main attraction of subsuming XML in program notation is the availability of good and well-proven abstraction mechanisms in the programming language. By having a general-purpose programming language in close proximity of the XML fragments there is less needs for special-purpose XML processors and tools. The overall document is a program that follows the rules of the programming language, and therefore the semantic problems of markup hosting (mentioned above) do not exist. The full expressiveness of XML is available through the programming notation, and via the programming concepts. The main challenge of subsuming XML in a programming language is to make a good fit between XML and the concepts of a specific programming language. This particular aspect is the main topic of this paper.

In the remaining parts of the paper we will in a systematic way discuss a number of issues that must be considered when XML is subsumed in a given programming language. The discussion is done bottom-up, starting with lexical and syntactic fitting in section 2, and ending with the semantic and paradigmatic fitting in section 3. As the concrete background of our work, we draw on the experiences from the LAML project, in which XML is subsumed in Scheme [5]. LAML details are not reported in this paper, however. The interested reader will have to consult one of the papers on LAML [12,11].

2. LEXICAL AND SYNTACTICAL FITTING

In XML documents, the textual contents appear in between pairs of start tags and end tags. In most programming languages, text is represented as *string constants* delimited by pairs of double quote characters. In the most straightforward subsumption of XML in a programming language, the terminal pieces of text (PCDATA in XML parlance) will be represented as string constants. Figure 1(b) shows an example of this solution in contrast to the similar and native XML document fragment in figure 1(a).

Several designers of programming notations in the area of XML have been reluctant to accept the

```
(a)   <paper>
        <author> Kurt Normark </author>
         <title>  The Duality of <em>XML Markup</em> and <em>Programming Notation</em>. </title>
      </paper>

(b)   (paper   (author "Kurt Normark")
               (title "The Duality of " (em "XML Markup") " and " (em "Programming Notation") "."))

(c)   (paper   (author [Kurt Normark])
               (title [The Duality of ,(em [XML Markup]) and ,(em [Programming Notation])]))

(d)   (paper   (author "Kurt Normark")
               (title "The Duality of" (em "XML Markup") "and" (em "Programming Notation") _ "."))

(e)   (body  "This is a"  (a 'href "http://www.w3c.org" 'target "main" "link") "to the W3C site.")
```

Figure 1. Illustration of lexical aspects of XML markup (a), LAML markup (b,d,e) and Scribe markup (c).

representation of text in string constants [13]. The main reason is undoubtedly the awkwardness of splitting strings in small parts whenever additional structuring is needed at the markup level (such as use of the `em` tags in figure 1). This has prompted the introduction of *semi-constant strings*. A semi-constant string allows non-constant constituents of a string, and it uses different characters as start quote and end quote delimiters. Figure 1(c) shows an example with bracketed string quotes and use of commas to mark non-constant parts. The example is from Scribe [13]. Curl [3] uses a similar notation.

If the textual content is represented as many small string constants in a program, white space needs to be part of these strings. This forces the author and programmer to be careful of adding spaces around certain strings, such as around “and” and after “The duality of” in figure 1(b). This is prone to errors in the authoring process. Notice that the use of semi-constant strings (see figure 1(c)) is better in this respect. As an alternative to both, we find it better to add syntactical constituents that control white spacing. This can be done either positively (having a marker that adds white space) or negatively (having a marker that removes white space). Figure 1(d) shows the solution in LAML, in which the underscore marker suppresses white space. With this solution, there is white space between every string and markup clause, unless explicitly suppressed by the special marker.

In XML, the characters ’ < ’, ’ > ’, ’&’ and others need special attention, via use of *character references* (for instance, `&lt;` and `&amp;`). The programming language, in which we subsume XML, has other needs for escaping of special-purpose characters. Using string constants there is no technical arguments for prohibiting the special-purpose XML characters in the textual contents. The special-purpose XML characters should be translated automatically to the similar XML character references when linearized to the textual XML format. In LAML we translate every PCDATA character and every attribute value character by means a *character transformation table*. In addition, this table may be extended to do translation of national characters to the corresponding character references in XML.

The syntactical composition of an XML fragment needs to have a natural counterpart when expressed in a programming language. Nested function calls or nested object constructors are natural solutions. In some programming languages, the use of manifest notation of data structures such as lists [7] or arrays can be used instead. In LAML we use nested calls of *mirror functions* (see section 3) for composition of XML documents.

The handling of XML attributes is a main issue in relation to the passing parameters to XML abstractions. The use of keyword parameters is a good solution. In our work on LAML in Scheme we simulate a keyword parameter mechanism, in order to come close to the native XML handling of attribute names and values. Figure 1(e) shows an example of the use of attributes in LAML.

As a syntactical concern, the XML related clauses should be validated in the same way as the syntactical checking of the conventional program clauses. Thus, XML validation errors should be revealed side by side with ordinary syntax and type errors. In programming languages with static type checking, much work has been done to subsume XML validation in the compiler’s type checking [14,6]. In general, it turns out to be difficult to deal with full XML validation in that way. In LAML and Scheme, which relies on ‘dynamic type checking’, we do a comprehensive XML validation at run time.

In a programmatic context, XML related clauses should be represented internally as tree structures (along the line of the DOM). Thus, the abstractions of the programming language should generate tree structures instead of text. A naive, textual representation of XML clauses makes it difficult to validate the document. A tree representation is also useful in relation to transformation of XML documents. In LAML we have been able to include a minimal set of transformation facilities, which we find simpler and more straightforward to use than an external XSLT processor. Textual XML linearization, which produces conventional XML markup, can be thought of as a particular example of a transformation. In general, it is relevant both to provide a compact linearization without superfluous white spacing, and a pretty printed linearization for human readability.

3. SEMANTICAL AND PARADIGMATIC FITTING

Most of the lexical and syntactical issues discussed in section 2 are orthogonal to the specifics of the programming language involved. But at an overall level, it is important to care about a good fit between the subsumption of XML in a programming language and the paradigmatic belonging of the programming language.

The most important advantage of subsuming XML in a given programming language is the availability of good abstraction mechanisms. In order to deal with document complexity, it is important to allow encapsulation of XML details in abstractions. Almost any programming language comes with well-proven abstraction mechanisms, which can be used immediately on XML details that are subsumed in the program. We will now discuss the fitting between major abstractions mechanisms and subsumptions of XML.

```
public HTML countHTML(int count){
  HTML h;

  h = <HTML>
       <HEAD> <TITLE> Test Page </TITLE>
       </HEAD>
       <BODY>
         You are number <b> {count} </b>
         to use this servelet.
       </BODY>
      </HTML>

  return h;}
```

(a)

```
(define (countHTML count)
   (html
     (head (title "Test Page"))
     (body
        "You are number"
        (b (as-string count))
        "to use this servelet.")
   )
)
```

(b)

Figure 2. Subsumption of XML in XOBE (a) and LAML (b). The XOBE example is from [6].

For languages that belong to the object-oriented paradigm it is obvious to represent fragments of XML documents as objects. In one extreme, a generic mirror of XML can be provided for by means of a few classes. In another extreme, each XML element should be mirrored as a separate class [6]. In any case, it is a challenge to provide XML notation through the means of the object-oriented programming language. The programmer should not feel the burden of manually parsing the XML structure. This observation caused Kempa and Linnemann to propose an extension of Java, called XOBE, that allows XML fragments to appear directly (and almost verbatim) within a Java program [6], see figure 2(a). However, from a program notation point of view this brings us back to mixed notation, which can be understood as a program hosted solution.

In the functional programming paradigm it is natural to model each XML element as a *mirror function*, which returns an instance of the internal tree structure. The composition of XML element instances can be dealt with by nested function calls. Figure 2(b) shows an example from LAML, which allows direct comparison with the XOBE example in figure 2(a). Using this notation it is possible to approach the verbatim XML notation from a source program written in the functional programming language Scheme.

Our work on LAML is related to the programming language Scheme [5], which is rooted in the functional programming paradigm. Each XML element mirror function possesses detailed knowledge about the properties and constraints of corresponding the XML element. In the functional programming paradigm the use of higher-order functions is a major characteristics. Of this reason, it is important to ensure a good fit between the XML mirror abstractions and the use of higher-order functions.

Scheme is a member of the Lisp family of programming languages, in which linear lists are the main data structures. When subsuming XML in a list language it is important to adapt it to the dominant data structure of the language. In LAML, lists of attributes and lists of content can be passed as parameters to the XML mirror functions. LAML will unfold such lists recursively. In our experience, this aspect of the mirror is very important for a natural organization of document fragments as lists.

Haskell [4] is a major functional programming language, in which several subsumptions of XML have been proposed. Validation of XML documents and XML applications by use of the Haskell type system is the driving force for most Haskell researchers, who work on XML issues. Thiemann's work is probably the most complete in that regard [14]. In the work of Wallace and Runciman [15] the contrast between a generic modeling of XML and a more specific modeling (in which the DTD gives rise to a number of Haskell type definitions) is discussed.

In imperative programming languages, procedures form the natural abstraction mechanism. But a subsumption of XML as procedures is not as natural as in the object-oriented and the functional programming paradigms. The reason is that procedure *calls* do not nest in the same way as function calls. Of this reason, procedural subsumptions of XML will have to be done at a lower level of abstraction.

Fragments of XML that belong to different XML languages can be mixed by supplying name space information. This aspect of XML should also be taken into consideration in a programming language subsumption. If name space information is ignored, name clashes will appear if two identically named fragments from two different XML languages are used in the same program. Programming languages that support modules or packages should make use of these concepts to deal with simultaneous use of two or more XML languages with conflicting set of names.

1. As the first and foremost concern, use the appropriate abstraction mechanism for subsumption of XML in the programming notation.
2. Be sure to fit especially important data structures of the programming language well in the mirroring of XML elements.
3. Go for a specific mirror of each XML element, including detailed knowledge of the element content model and the attributes.
4. Give special care to the representation of XML attributes in the way parameters are passed to the abstractions.
5. Integrate XML validation into the error-checking model of the programming language.
6. The programmatic XML notation should give rise to an abstract, structural representation of the XML document when the program is executed.
7. As a detailed issue of great importance, provide for a good representation of the textual contents.
8. Avoid lexical representation of white space in string constants. Use syntactical markers instead.
9. Provide for direct notation of special purpose XML characters in the programming language notation. Use a character transformation table to control the necessary character translations.

Figure 3. Summary of advice for subsuming XML in a programming language.

4. CONCLUSION

In this paper we have explored the duality of XML and programming language notation, mainly in a context where XML documents are processed by a program written in a general-purpose programming language. To put the main topic of the paper a proper context, we have also identified four broad categories of integration between XML and pieces of programs. Figure 3 summarizes the concrete findings.

The main part of the paper has been written from a program-centered position, and for readers who prefer well-integrated solutions in general-purpose programming languages. We have concentrated on a structural representation of XML that fits well with both the paradigmatic belonging and the dominant data structures of the programming language.

Throughout the paper we have focused on several aspects of a programmatic notation for XML. This is relevant, because in many applications it is necessary to include large fragments of semi constant XML fragments in the program. The variable part of the XML fragments is expressed in native programming notation, and it will have to “sneak into” the XML parts in smooth and natural ways. In the ideal case, the programming language allows us to notate XML fragments in a way which is programmatically convenient, and not too far distanced from native XML notation.

Some programming languages are better than others for satisfactory subsumption of XML. The programming languages best suited are syntactically very flexible, especially in the neighborhood of the central abstractions, and with respect to parameter passing. In stiff and rigid languages it may be worthwhile to alleviate some of the problems by means of preprocessing.

REFERENCES

[1] James Clark. XSL transformations (XSLT) version 1.0. W3C recommendation in http://www.w3.org/TR/xslt, November 1999.

[2] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language. In *International Workshop on the Web and Databases (WebDB)*, Dallas, TX, 2000.

[3] M. Hostetter, D. Kranz, C. Seed, C. Terman, and S. Ward. Curl: A gentle slope language for the web. *World Wide Web Journal*, 2(2), 1997. Available from http://www.w3j.com/6/s3.kranz.html.

[4] Simon Peyton Jones and John Hughes (editors). Haskell 98: A non-strict, purely functional language, February 1999. http://haskell.systemsz.cs.yale.edu/onlinereport/.

[5] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

[6] Martin Kempa and Volker Linnemann. On XML objects. The e-proceedings of PLAN-X: Programming Language Technologies for XML, October 2002. Available in http://www.research.avayalabs.com/user/wadler/planx/planx-eproceed/papers/E00-699879232.pdf.

[7] Oleg Kiselyov. SXML. http://okmij.org/ftp/Scheme/SXML.html, August 2002.

[8] Bruce R. Lewis. BRL—a database-oriented language to embed in HTML and other markup, October 2000. http://brl.sourceforge.net/.

[9] Thomas Nicol. XEXPR - a scripting language for XML, November 2000. W3C note located at http://www.w3.org/TR/xexpr/.

[10] Kurt Nørmark. The LAML home page, 1999-2003. http://www.cs.auc.dk/~normark/laml/.

[11] Kurt Nørmark. Programmatic WWW authoring using Scheme and LAML. In *The proceedings of the Eleventh International World Wide Web Conference - The web engineering track*, May 2002. ISBN 1-880672-20-0. Available from http://www2002.org/CDROM/alternate/296/.

[12] Kurt Nørmark. Web programming in Scheme with LAML. Submitted for publication, April 2003. Available via [10].

[13] Manuel Serrano and Erick Gallesio. This is scribe! Presented at the *Third Workshop on Scheme and Functional Programming*, October 2002. http://www-sop.inria.fr/mimosa/fp/Scribe/doc/scribe.html.

[14] Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(5):435–468, July 2002.

[15] Malcolm Wallace and Colin Runciman. Haskell and XML: generic combinators or type-based translation? In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 148–159. ACM Press, 1999. Published in Sigplan Notices vol 34 number 9.