XSieve: extending XSLT with the roots of XSLT

Oleg Parashchenko

Saint-Petersburg State University, Russia

olpa@xmlhack.ru

Introduction

XSLT is one of the most versatile programming inventions of recent time. It makes XML data conversion an easy and pleasant task. Unfortunately, XSLT has limitations. It fails when data conversion should be supported by programming. XSLT was not designed to be a general-purpose programming language and attempting to use it as one leads to frustration.

On the other hand, classical general-purpose programming languages like Perl or Java are not tailored to hierarchical data transformations. Principally, it's not a big problem to write a library for such tasks, but, most likely, the result would remind one of a poorly re-invented XPath and XSLT.

The XSLT extension mechanism might be a good solution: XSLT is for data transformation, and the extending language is for programming. But in many cases, the programming part performs transformations, which returns us back to the problem with the classical programming languages.

New XML programming languages don't help much with data transformations. Instead, they are interested in highlighting different aspects of XML. Some of them just use XML as a native syntax for writing programs (*[OXML]*), some are XML type-centric (*[XTATIC]*, *[CDUCE]*, *[SCALA]*), some introduce native XML data type to exiting programming languages (*[XJ]*, *[E4X]*), some assimilate XML technologies (*[XLINQ]*), some deal with streamed XML processing (*[STX]*, *[XTISP]*).

And anyway, from a practical point of view, new programming languages have at least two important pitfalls. First, they are new. This means that time is required to learn the language and its infrastructure. Second, the language or vendor might disappear, deprecating the value of the written code base.

A possible approach to complex data transformations is to try a language which is good for symbolic computations and adapt it to XML needs. Such languages do exist, and they are from the Lisp family. Some representatives are Common Lisp, Scheme *[SCHEME]* and DSSSL *[DSSSL]*.

DSSSL is a very good candidate, as it was specially designed for transforming SGML (now XML) data and it is an ancestor of XSLT. Unfortunately, DSSSL is not Scheme, but a simplified dialect of Scheme. It has all the pitfalls of rarely used languages and, more importantly, lacks important Scheme features such as macros and continuations.

Scheme SXML library is another good candidate. It provides all the required features — XPath-like queries, tree traversal and rewriting — and has been successfully used in practice. The only problem is that it can't be easily integrated into existing XSLT infrastructure.

XSieve fixes both the XSLT problem and the SXML problem, uniting XSLT and SXML together. Generally, XSieve is a standards-compliant XSLT language which allows its user to ignore Scheme as long as it's not necessary and employ it in the situations where its benefits are most obvious.

This paper is organized as follows. First, we introduce the SXML format and basic tools to work with it. XSieve itself appears in the third section, which contains a "hello world" example, XSieve definition and an outline of the Scheme-to-XSLT API. It is followed by two XSieve examples. The first one demonstrates a task which is non-trivial in XSLT, but trivial in XSieve. The second one shows that Scheme code is easier to use and re-use. The next section demonstrates a practical usage of XSieve for syntax highlighting of code listings. It is followed by notes on technical implementation and performance. Finally, comes the summary with directions for further work.

SXML format

The SXML format is a concrete representation of XML infoset *[INFOSET]* using S-expressions *[SEXP]*. All XML features, including namespaces, are supported. This section introduces SXML by examples. For the formal definition, see the SXML specification *[SXML]*.

XML text nodes are represented by Scheme strings. XML elements are represented as Scheme lists, whose head is the name of the element and whose tail is the list of children. For example, the following XML fragment:

```
<para>Hello, World!</para>
```

is represented by SXML as:

```
(para "Hello, World!")
```

Processing instructions and comments are represented as elements with special names **`*PI*`** and **`*COMMENT*`**, respectively. XML:

```
<!-- use table layout -->
<?db as-table?>
```

SXML:

```
(*COMMENT* " use table layout ")
(*PI* db "as-table")
```

Attributes are represented exactly as elements. In order to distinguish them, attributes are collected under the special node `@`. XML:

```
<elem a1="val1" a2="val2" />
```

SXML:

```
(elem (@ (a1 "val1")
         (a2 "val2")))
```

The XML root node corresponds to the SXML **`*TOP*`** element. To summarize the above, here is a simple document in both XML and SXML representations. XML:

```
<?xml version="1.0"?>
<!-- $Id$ -->
<article id="hw">
  <title>Hello</title>
  <para>Hello, <object>World</object>!</para>
</article>
```

SXML:

```
(*TOP*
  (*COMMENT* " $Id$ ")
  (article  (@ (id "hw"))
    "\n  "
    (title "Hello")
    "\n  "
    (para "Hello, " (object "World") "!")
    "\n"))
```

SXML tools

No API is required to work with SXML data because SXML fragments are S-expressions, and Scheme is equipped with core functions and SRFI-extensions to efficiently work with S-expressions. For example, if `elem` is an SXML-element then

```
(car elem)
```

returns the element name. The expression

```
(cdr elem)
```

returns a nodeset consisting of the children nodes of `elem`. The expression

```
(for-each func (cdr elem))
```

applies the function `func` to each children of `elem`.

Anyway, a useful set of APIs is provided by the SXML library. The most important are SXPath and SXSLT *[SXMLPAPER]*, which are counterparts of XPath and XSLT, respectively.

SXPath is implemented as a sequence of projection and filtering primitives — converters — joined by combinators. Each step in SXPath can be a Scheme expression, so one can use arbitrary complex axes and predicates, which are beyond the standard XPath.

SXSLT *[SXSLT]* is a head-first tree re-writing system. Even though it's quite simple, when used together with Scheme features, complex transformations can be written. The paper about SXSLT *[SXSLT]* provides several impressive examples and our solution for syntax highlighting in `ProgramListing` is also based on SXSLT.

XSieve

Here is an example of a simple XSieve program:

```
<xsl:stylesheet
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
  xmlns:s   = "http://xsieve.sourceforge.net"
  extension-element-prefixes="s"
  version   = "1.0">

<xsl:output indent="yes"/>

<xsl:template match="/">
  <s:scheme>
    '(article (@ (id "hw"))
        (title "Hello")
        (para  "Hello, " (object "World") "!"))
  </s:scheme>
</xsl:template>

</xsl:stylesheet>
```

Scheme code is embedded into XSLT using the extension element **`s:scheme`**. The attributes `xmlns:s` and `extension-element-prefixes` are used to declare the XSieve extension to the XSLT processor.

The same data is represented as XML when in XSLT mode, and as SXML when in Scheme mode. The conversion is automatic and transparent to the user.

In the above example, the result of the Scheme code execution is the S-expression "`(artile ...)`", which is converted to XML. The result of execution of the XSieve program is:

```
<article id="hw">
  <title>Hello</title>
  <para>Hello, <object>World</object>!</para>
</article>
```

In addition to `s:scheme`, there is another Scheme code container **`s:init`**, which should be a direct child of `xsl:stylesheet`. The code in `s:init` is executed before XSLT processor starts to apply templates.

In `s:scheme`, the code can communicate with the XSLT processor using a very small but useful API.

The simplest function is **`x:current`**, which returns the context node as SXML. The function **`x:string`** converts an SXML expression into string. These functions are analogues of the XSLT/XPath functions `current()` and `string()`, respectively.

The function **`x:eval`** evaluates an XPath expression using the XPath engine of the XSLT processor. An optional `x:eval` argument specifies the base node for relative XPath expressions.

Finally, the function **`x:apply-templates`** switches to the XSLT mode and starts the usual XSLT processing of its argument. XSLT mode and parameters are supported. The nesting level of XSLT and Scheme modes isn't limited.

Examples

Quantity times price

Let us consider the task of calculating the sum of each `price*qty` from the following XML fragment:

```
...
<item price="20" qty="2"/>
<item price="30" qty="1"/>
<item price="40" qty="4"/>
...
```

It's an example of a task which is considered trivial, but has a non-intitive XSLT solution. The XSLT FAQ provides a solution *[FAQSUM]*, which takes 25 lines and uses recursion. As result, the code is hard to understand. The XSieve solution is straightforward:

```
(apply +
  (map (lambda (node)
      (* (x:eval "number(@qty)"   node)
         (x:eval "number(@price)" node)))
    (x:eval "//item")))
```

The code is easily read and understood by those who know Scheme. The `map`

function takes a list of all `item` elements, applies an anonymous lambda-function to each item, and returns the results as a list, which in turn, is processed by +.

Escaping characters

XSLT is usually used to convert XML trees to XML trees, but sometimes one wants to generate plain text fragments. It could be, for example, JavaScript or TeX code. In this case special characters shouldn't appear as is, but escaped somehow. For example, here is an rewriting table for JavaScript strings:

```
\ to \\
" to \"
' to \'
```

And here is a part of the rewriting table for LaTeX:

```
_ to \_
^ to \^{}
< to \textless
...
```

XSLT isn't the best language for multiple string replacements, especially if we want to separate the code and the data about the replacements. Actually, it's possible, and the XSLT FAQ references to a solution *[FAQSTR]*, but it's long and somehow limited. For example, the code of the solution assumes that there is only one rewriting table, which might be wrong. Principally, it's possible to improve the code, but it leads to further complication of the solution.

As for Scheme, it also isn't targeted at string processing and provides only a minimal set of string operations. Anyway, the Scheme solution (the SXML library function `make-char-quotator`) is concise and can be used with different rewriting tables, defined anywhere.

```
<s:init>
(load "util.scm")
(define q (make-char-quotator
  '((#\\ . "\\\\")  (#\" . "\\\"")  (#\' . "\\'"))))
</s:init>

<xsl:template match="/">
  <html><body><script>
```

```
      <xsl:text>alert("Hello, </xsl:text>
      <s:scheme>(q (x:eval "string(/what)"))</s:scheme>
      <xsl:text>");</xsl:text>
    </script></body></html>
  </xsl:template>
```

In `s:init`, `make-char-quotator` takes an rewriting map for JavaScript strings and returns a function which performs substitutions. We bind this function to the name `q` and use it in `s:scheme`.

If the input XML file looks like

```
<what>"'\\|//'"</what>
```

the result is:

```
<html><body><script>alert("Hello,
  \"\"\'\\\\|//\'\"\"");</script></body></html>
```

Here is yet another use of `make-char-quotator`:

```
(make-char-quotator '((#\newline . ((br) "\n"))))
```

It creates a function which inserts tags `br` before the newline characters. The same task in XSLT requires writing code which is similar to the code for escaping characters. Conversely, XSieve encourages code re-use.

In Practice: Syntax Highlighting

Code examples are more readable when syntax is highlighted. Unfortunately, the most part of documentation generated with XSLT contains monotone code boxes instead of nicely formatted programs.

A small problem is that XSLT isn't a language suitable for writing syntax-highlighting programs. The simplest solution is to use an external software to process code listings, either before the XSLT transformation, or during it. For example, there exists a Saxon extension that can perform syntax highlighting during an XSLT transformation *[SYNSAXON]*. One pitfall of such solutions is that they are very implementation-dependent and aren't portable between XSLT processors.

But the main issue is that code listings are not plain text only. Here is a quite possible DocBook fragment:

```
<programlisting>
```

```
&lt;para>Hello, <emphasis role="bold">&amp;who;</emphasis
 >!&lt;/para> <co id="who-entity"/>
</programlisting>
```

In this fragment we emphasis entity usage and annotate it by a callout.

A colorizer should not only highlight the listing, but also retain the original inner markup. I'm not aware of such tools, and writing them is a non-trivial task.

We have developed an XSieve solution *[SYNXSIEVE]* for highlighting program listings in DocBook. XSieve hooks processing of `ProgramListing` elements. First, it uses the `colorer` *[COLORER]* program to highlight the listing ignoring the DocBook markup. Then, it joins two parallel sets of markup: one comes from the DocBook source, and another one from the highlighted listing. Finally, the result is processed as usual.

Joining parallel sets of markup is a challenging task, but has a surprisingly short XSieve solution. In the first version, it takes only 132 lines, including comments and empty lines.

Technical Implementation

There is only one XSieve implementation at the moment. It is based on the XSLT processor xsltproc (libxslt library *[LIBXSLT]*) and Scheme implementation Guile *[GUILE]*. The main technical problems in development were the following.

- Correct handling of namespaces.
- Uniting the memory management of libxslt and the garbage collection of Guile.
- Switching to XSLT mode from inside the extension code.

While the last two issues might not be a problem for other XSLT-Scheme implementation pairs, the first one is essential. When converting between XML and SXML, correct handling of namespace declarations and usage needs efforts.

XSieve is implemented as an xsltproc plugin. As result, libxslt bindings for high-level languages, such as Python, Perl or PHP, support execution of XSieve stylesheets.

To test XSieve, we have taken the DocBook XSLT stylesheets and automatically

converted them to XSieve. For example,

```
<xsl:apply-templates select="smth"/>
```

is converted to

```
<s:scheme>(x:apply-templates (x:eval "smth"))</s:scheme>
```

The converter was written in XSieve itself, which is the first practical usage of XSieve.

The generated XSieve DocBook stylesheets produce the same results as the original XSLT stylesheets. This gives us confidence in the correctness of the XSieve implementation.

The DocBook test is also a performance test for the worst case. The generated stylesheets switch excessively between XSLT and Scheme modes, which leads to performance degradation. The main source of the problem is non-optimal code, which will be corrected in the next releases.

In real stylesheets, performance difference is hard to measure. Both XSLT and XSieve solutions work equally fast. But as XSieve is used to replace XSLT quirks, which are not always optimal, XSieve solutions should work faster. For example, XSLT solution for quantity times price is noticeable slower when number of items is 1000. When the number is 10000, it hangs for several minutes, while XSieve finishes in a few seconds.

Conclusion and future work

The paper has explained the need for a new XML transformation language and described XSieve, a possible solution. XSieve is a combination of the well-known programming languages XSLT and Scheme. More precisely, XSieve is a standards-compliant XSLT which uses Scheme as the extension programming language. Here are some of the benefits:

- For hierarchical data transformation, Scheme is as good as XSLT. In some aspects, the SXML library outperforms XPath and XSLT.
- XSieve stylesheets can vary the proportion of XSLT and Scheme, selecting the best appropriate syntax and execution model for subtasks.
- Both XSieve stylesheets and generic Scheme programs can re-use the same

Scheme libraries.

The practical example of syntax highlighting of code listing supports our confidence that XSieve is very powerful.

The closest goal of the XSieve project is to enhance DocBook XSLT stylesheets. For example, for the needs of technical publishing, we'd like to develop a re-usable CALS tables converter, which could be used both in XSLT and other software.

We expect more XSieve implementations in the future. Scheme is a simple language with a plenty of realizations, so the hardest thing in implementing XSieve is to write a converter between XML and SXML representations.

This way XSieve could be an alternative or a partner to EXSLT *[EXSLT]* extensions. Instead of implementing EXSLT functionality each time for each XSLT processor, it would be possible to implement it once in XSieve and use everywhere.

Acknowledgements

I'd like to thank to Google, and especially my mentor Yoshiki Hayashi, for accepting XSieve as the Google Summer of Code 2005 project. It helped to grow XSieve from a research prototype to an end-user software. I'm also very grateful to Kirill Lisovsky for his fruitful comments on the early version of the paper.

Bibliography

[CDUCE] CDuce: Home page, http://www.cduce.org/

[COLORER] Colorer-take5 library, http://colorer.sourceforge.net/

[DSSSL] International Standards Organization, Document Style Semantics and Specification Language, 1996, International Standard ISO/IEC 10179:1996(E)

[E4X] Ecma International, ECMAScript for XML (E4X) Specification, Standard ECMA-357, 2nd edition, December 2005, http://www.ecma-

international.org/publications/standards/Ecma-357.htm

[EXSLT] EXSLT, http://exslt.org/

[FAQSTR] Multiple string replacements, General String Replace, XSLT Questions and Answers, http://www.dpawson.co.uk/xsl/sect2/StringReplace.html

[FAQSUM] Quantity times price, Math, XSLT Questions and Answers, http://www.dpawson.co.uk/xsl/sect2/N5121.html

[GUILE] Guile, Project GNU's extension language, http://www.gnu.org/software/guile/

[INFOSET] XML Information Set (Second Edition), W3C Recommendation 4 February 2004, http://www.w3.org/TR/xml-infoset/

[LIBXSLT] libxslt, The XSLT C library for Gnome, http://xmlsoft.org/XSLT/

[OXML] o:XML - object-oriented XML, http://www.o-xml.org/

[SCALA] Scala Overview: XML Processing, http://scala.epfl.ch/intro/xml.html

[SCHEME] Kelsey, R., Clinger, W., Rees, J. (eds.): Revised5 Report on the Algorithmic Language Scheme. Higher-Order and Symbolic Computation, Vol. 11, No. 1, August, 1998, http://www.brics.dk/~hosc/11-1/, http://schemers.org/Documents/Standards/

[SEXP] John McCarthy, Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Comm. ACM, 3(4):184-195, April 1960, http://www-formal.stanford.edu/jmc/recursive/recursive.html

[STX] Streaming Transformations for XML (STX), http://stx.sourceforge.net/

[SYNSAXON] Bob Stayton, Syntax highlighting, Program listings, Chapter 26, DocBook XSL: The Complete Guide, Third Edition, http://www.sagehill.net/docbookxsl/SyntaxHighlighting.html

[SYNXSIEVE] Syntax Highlighting for DocBook Program Listings, http://tohtml.com/dbsy/

[SXML] Oleg Kiselyov, SXML specification, March 12, 2004, http://okmij.org/ftp/Scheme/SXML.html

[SXMLPAPER] O. Kiselyov, K. Lisovsky, XML, XPath, XSLT implementations as SXML, SXPath, SXSLT, International LISP conference, San Francisco, 2002, http://pobox.com/~oleg/ftp/papers/SXs.pdf

[SXSLT] Oleg Kiselyov, Chriram Krishnamurthi, SXSLT: Manipulation Language for XML, In PADL, LNCS 2562, 2003, http://okmij.org/ftp/papers/SXSLT.ps.gz

[XJ] XJ: XML Enhancements for Java, http://www.alphaworks.ibm.com/tech/xj

[XLINQ] Erik Meijer, Brian Beckman, XLinq: XML Programming Refactored (The Return Of The Monoids), In Proc. XML 2005, http://research.microsoft.com/~emeijer/Papers/XLinq%20XML%20Programming%20Refactored%20(The%20Return%20Of%20The%20Monoids).htm

[XTATIC] The Xtatic Project: Native XML processing for C#, http://www.cis.upenn.edu/~bcpierce/xtatic/index.html

[XTISP] XTiSP, An Implementation Framework for XML Transformation Languages Intended for Stream Processing, http://xtisp.psdlab.org/en/