Status Report:  
Layered Streaming XML Processing with Modules *

Tyng-Ruey Chuang  Max Schäfer  
Institute of Information Science  
Academia Sinica  
Nangang 115, Taipei City, Taiwan  
{trc, xiemaisi}@iis.sinica.edu.tw

Abstract
We report on our experience in designing and implementing a layered streaming XML processing library in Objective Caml, which allows the user to parse documents at different abstraction levels and switch between these levels on the fly in a typesafe manner. The implementation makes essential use of Objective Caml’s module facility and achieves a very satisfactory performance.

Categories and Subject Descriptors D.3.3 [Language Constructs and Features]: Modules, packages; I.7.0 [Document and Text Processing]: General

General Terms Design, Languages, Performance

Keywords Streaming XML Processing, Modular Software, Functional Programming, ML, Modules, XML

1. Introduction
Over the last couple of years, much work has gone into the development of streaming XML processing software which parses an input document into a stream of tags and character data as opposed to DOM parsers (Apparao and Byrne 1998), which build an in-memory representation of a whole document. The streaming method of XML processing, while certainly more low-level, has its uses when dealing with very large documents or when only a part of the document needs to be examined.

But even streaming processors are sometimes too slow. In a geospatial information processing application developed in our group, a colleague encountered the problem of extracting information about certain place names from an XML dump of the popular online encyclopedia Wikipedia. This dump is huge in size (eight gigabytes to one terabyte, depending on the chosen version), but very simply structured, being basically nothing more than a listing of all pages with the title given in a title element followed by the content given in a text element; see Figure 1 for a short excerpt.

Thus the task could be solved by simply scanning the document for <title> tags, matching against the desired title, and, when the

article is found, extracting the character data contained between the following <text> and </text> tag pair. Obviously, using a DOM-based parser was out of the question, both due to the prohibitive size of the input and the small amount of data actually needed. But even using a low-level SAX parser (Brownell and Megginson 1998-2006) turned out to be too slow. Apparently, the parser spent a lot of time reading the character data of articles that were being skipped, looking for entity references to resolve and CDATA (quoted character data) sections to parse. In this case, however, such effort was basically wasted, since only predefined entity references and no CDATA sections were to be expected.

In a desperate bid to regain acceptable processing speed, our colleague resorted to brute-force pattern matching, doing away with the XML parser and working with raw character data instead. While in this case practicable, such a solution is of course not satisfactory. We thus came up with the design of a layered XML streaming processing library to solve this issue.

In our design, an XML document can be viewed at four different abstraction levels:

1. as a stream of characters,
2. as a stream of (raw) tokens, e.g., start and end tags, and character data,
3. as a normalized stream of tokens, where proper nesting of start and end tags is ensured, and
4. as a stream of “typed” tokens, which have been checked against a DTD to make sure that tags only refer to declared elements and are provided with appropriate attributes

These four levels are built on each other with each stream of a higher level being layered on top of a stream of the next lower level. The programmer can now choose the level she desires to work at, and if the need should arise, she can not only switch to a higher level (by building a more abstract stream on top of a more

---

*This work was partially supported by the iCAST project sponsored by the National Science Council, Taiwan, under grants no. NSC95-3114-P-001-001-Y02, NSC95-3114-P-001-002-Y02, and NSC96-3114-P-001-002-Y.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ML’07, October 5, 2007, Freiburg, Germany.  
Copyright © 2007 ACM 978-1-59593-676-9/07/0010... $5.00

---

Figure 1. A short excerpt of an XML dump of Wikipedia

```
<page>
<title>Algeria</title>
...
<text xml:space="preserve">Algeria
Introduction
Background: After a century of rule...
</text>
...
</page>
```
primitive one), but also switch back to a lower level (by recovering the primitive stream underlying a more abstract one).

In this way, the task described above can very efficiently be solved by first reading from a high-level stream looking for a title start tag: if the title is not the desired one, the lowest-level character stream can be recovered to quickly skip forward until the closing tag of the current page is found. Then one can again go back to the higher level to look for the next title.

This framework should be implemented in a general way so as to make it possible for the user to supply their own stream implementations. Thus, the framework has to provide an abstract account of streams and their transformations, as well as mechanisms for applying transformations to low-level streams and for recovering underlying streams. Above all, this should be possible in a typesafe manner such that the process of layering and un-layering can be statically checked at compile time and does not rely on run-time type information or related mechanisms.

In the rest of this report, we describe how to implement such a framework using Objective Caml’s module system and present some benchmarks showing the feasibility of our approach.

2. Implementing the Framework

2.1 Streams as Polymorphic Types

In its standard library, Objective Caml already has a datatype of (stateful) streams. Here, streams are represented by objects of the polymorphic type

```ocaml
type 'a Stream.t
```

where 'a is the type of elements of the stream. A function

```ocaml
peek : 'a Stream.t -> 'a option
```

is provided that performs a one-element lookahead (without altering the stream state), and another function

```ocaml
next : 'a Stream.t -> 'a
```

reads one element from the stream, advancing the stream by one element.

We first tried to build our framework on a similar approach. One would then need a function down to recover the more primitive stream underlying a higher-level stream, so the stream type would have to be abstracted over an additional type variable expressing the underlying stream’s type, i.e.

```ocaml
type ('a, 'b) stream
```

Then down would have the type

```ocaml
down : ('a, 'b) stream -> 'b
```

But this is not satisfactory: We have no way of ensuring that the underlying type 'b is in fact a stream, so this is not a good way of implementing our specification.

One could try to remedy this by interpreting 'b as the type of elements of the underlying stream, giving down a type like

```ocaml
down : ('a, 'b) stream -> ('b, ?) stream
```

How do we know which type to use for '?'? Apparently, we have to abstract over an additional type variable expressing the element type of the underlying stream’s underlying stream, and so on. This approach clearly does not work.

1 A prototype implementation is available on the second author’s homepage at http://www.iis.sinica.edu.tw/~xiemaisi/.

2.2 Streams as Modules

The main problem is that in our design, the type of elements and of the underlying stream should be a fixed part of the stream’s type (existential quantification, so to speak); a polymorphic datatype (which provides universal quantification) does not seem to provide an adequate solution. So we decided to use Objective Caml’s module system instead.

A (not necessarily layered) stream is represented by a module conforming to signature STREAM:

```ocaml
module type STREAM = sig
  type elt
  type t
  val peek : t -> elt option
  val next : t -> elt
end
```

Type t is a type of streams with elements of type elt; peek and next are as before.

A layered stream, on the other hand, is represented by a module conforming to signature LSTREAM:

```ocaml
module type LSTREAM = sig
  include STREAM
  module Base : STREAM
  val up : Base.t -> t
  val down : t -> Base.t
end
```

The inner module Base represents the underlying stream type, and functions up and down serve to build up instances of this stream type from instances of the base stream type, and to recover the underlying instance of the base stream type, respectively. Since Objective Caml supports a form of subtyping for modules, a module of type LSTREAM can be provided anywhere a STREAM is expected.

In our application scenario, the type of character streams (the lowest level of our hierarchy) is contained in a module CharStream of signature STREAM which just wraps the standard library’s char Stream.t type. The tokenized stream type (one level higher up) is a module TokenizedStream of signature LSTREAM with TokenizedStream.Base = CharStream, and similarly for the higher levels.

2.3 Stream Transformers

Implementing every layered stream module from scratch would be very tedious. So we introduce a number of different stream transformers, which are functors that take a module of signature STREAM and yield a module of signature LSTREAM with the input module as Base.

The simplest kind of stream transformer is element-wise mapping. First, we need a module that describes the mapping to be performed. It has the following signature:

```ocaml
module type MAPPING = sig
  type i
  type o
  val map : i -> o
end
```

Then we can easily define a functor Map that applies such a mapping to a stream:

```ocaml
module Map (M:MAPPING) =
  (S:STREAM with type elt = M.i) =
  struct
    type elt = M.o
    type t = S.t
  end
```

```ocaml
open Map
```

```ocaml
let transform = Map.Map (S)
```

```ocaml
let transformed_stream = transform stream
```

```ocaml
let transformed_element = transformed_stream.peek
```

```ocaml
let transformed_next = transformed_stream.next
```
implement our XML processing library.

3. Implementing the XML Processors

The lowest level character stream module CharStream is in fact nothing more than a wrapper around the standard library's Stream.t type, instantiated to the element type char. If needed, one could implement character set conversion on this level, too.

A simple XML tokenizer Tokenizer can now be implemented as a MEALY_MACHINE called Normalizer which is Run on the module CharStream to produce a module of XML token streams:

```ocaml
module TokenStream = Run(Tokenizer)(CharStream)
```

On top of this, another MEALY_MACHINE called Normalizer implements the normalizing step described above. In particular, it checks that start and end tags are properly matched and discards inter-tag whitespace unless the surrounding tag has its xml:space attribute set to "preserve":

```ocaml
module NormalizedTokenStream = Run(Normalizer)(TokenStream)
```

Streams of this level provide more or less the functionality of a normal XML pull parser (Haustein and Slominski 2002-2006). But there is more. Given a DTD, we can map its collection of element types to an Objective Caml variant type, and then bind XML start tags to Objective Caml objects of this type. For the DTD underlying the Wikipedia dump, for example, part of the type definition would look like this:

```ocaml
type element = Page | Title | ...
```

Observe that to conform to Objective Caml's naming convention, the element names have to be capitalized.

Some elements, like Wikipedia's text, have attributes, which we can model as records:

```ocaml
type text_type = { text'xml_space : string }...
type element = Page | Title | Text of text_type | ...
```

In this way, a rather high-level representation of the token stream can be achieved. Compared to complete validation of the input document (which is somewhat tricky for streaming processors), this partial validation is easy to perform and assures at least that only declared elements are used, and that they are provided with all necessary attributes.

For any given DTD, one can easily derive the type definitions outlined above and implement a mapping function which maps normalized tokens to these "typed" tokens. Thus one obtains a module of signature MAPPING that can be used as a stream transformer.

In fact, we have implemented a preprocessor dtpp (for "DTD preprocessor"), based on Objective Caml's CamlP4 framework, which automatically derives a mapping module from a DTD and compiles it into an Objective Caml object file. If the DTD is, for example, stored in the file mediawiki.dtd, a command like the following will compile it into the binary object file mediawiki.cmo:

```ocaml
ocamlc -c -pp 'camlp4o dtpp.cmo'

ocamlc -impl mediawiki.cmo
```

This compiled module can then be used to obtain a typed token stream:

```ocaml
module TypedTokenStream = Map(Mediawiki)(NormalizedTokenStream)
```

Now one can, using the corresponding up functions, construct instances of different streams like this:

```ocaml
let chrstrm = CharStream.of_channel stdin
let tkstrm = TokenStream.up chrstrm
let ntkstrm = NormalizedTokenStream.up tkstrm
let ttkstrm = TypedTokenStream.up ntkstrm,
```

To read from them, one uses the appropriate peek and next functions, and to recover underlying streams one uses the down function as in this example:

```ocaml
let ntkstrm' = TypedTokenStream.down ttkstrm
```

See the appendix for a more detailed explanation of the translation process from a DTD into an Objective Caml module.

4. Performance

To convince ourselves of the linear time complexity of our library, we first wrote a simple stream consumer program that consumed progressively longer prefixes of the Wikipedia dump, operating on character streams, token streams, normalized token streams, and typed token streams, respectively. The results, shown in Figure 2, are encouraging: Not only is the time complexity clearly linear,
but adding abstraction layers does not noticeably influence performance (though this may partly be due to the very simple structure of the document being processed).

Our main test case was now to implement a program solving the article extraction problem mentioned in the introduction. We first implemented it to only use the highest-level stream, thus simulating a normal pull parser, and then optimized it to switch back to the lower level character stream to skip over uninteresting sections.

Both programs were compiled using the native Object Caml compiler version 3.09.3 and put to the test extracting the article contents for Lithuania, Nauru, Peru, Sudan, Uruguay, and Zimbabwe. We used the smallest available Wikipedia dump, which at eight gigabytes incidentally was still too large to unzip on our harddrive and hence had to be unpacked and fed to the program on the fly. The timings, obtained on a four-CPU 2.53 GHz Intel machine with 1.5 GB of RAM running Linux 2.6.17, are shown in Figure 3, ordered by relative position of the articles inside the dump (which is not quite alphabetic). The results show a performance increase of about 25% for the optimized version over the original.

Figure 2. Results of a simple stream consuming benchmark

Table 1. Results of a simple stream consuming benchmark

<table>
<thead>
<tr>
<th>Article title</th>
<th>Original</th>
<th>Optimized</th>
<th>Xmlm</th>
<th>ocaml-xmrl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lithuania</td>
<td>42s</td>
<td>32s</td>
<td>44s</td>
<td>33s</td>
</tr>
<tr>
<td>Nauru</td>
<td>52s</td>
<td>39s</td>
<td>53s</td>
<td>39s</td>
</tr>
<tr>
<td>Sudan</td>
<td>1m18s</td>
<td>51s</td>
<td>1m11s</td>
<td>52s</td>
</tr>
<tr>
<td>Uruguay</td>
<td>1m19s</td>
<td>1m8s</td>
<td>1m21s</td>
<td>1m6s</td>
</tr>
<tr>
<td>Zimbabwe</td>
<td>1m24s</td>
<td>1m4s</td>
<td>1m26s</td>
<td>1m5s</td>
</tr>
<tr>
<td>Peru</td>
<td>4m9s</td>
<td>3m9s</td>
<td>4m22s</td>
<td>3m15s</td>
</tr>
</tbody>
</table>

Figure 3. Results of the geo-spatial article extraction benchmark

5. Further Issues and Conclusion

While the results of our experiments are certainly encouraging, there are a number of issues we encountered.

First, we would have liked to use a stateless stream implementation, but tests showed that the performance penalty is severe: Nearly 50% of execution time is spent in garbage collection even if all stream elements are immediately discarded upon reading.

A problem with using the stateful implementation is that reading from a lower-level stream can cause a higher-level stream built on it to lose synchronization, so one cannot simply switch back to using the higher-level stream, but has to build a new instance on top of the changed lower-level stream. This issue could be solved in a cleaner way if stateless streams were used, but the price to pay is too high.

On the bright side, we think that the use of modules to structure an XML processing library is worth pursuing further. We already explained why it suits our needs better than a solution based on polymorphic types, and it seems that an object oriented approach would ultimately fail for the same reasons: In order to accommodate streams of different element types, one would have to use generic classes, which leads to a similar dilemma as in the case of polymorphic types.

Using parametrized functors like Map and Run to implement stream transformers makes our library more than a one-off solution: The current architecture with its four abstraction levels can easily be adapted or extended, all the while staying within the same framework of layered streams.

As for future work, we would like to expand our current prototype implementation into a full-fledged XML processing library and add support for namespaces and XML Schema. We are also contemplating the addition of further validation or transformation layers to the current architecture.

Finally, one might of course ask whether such an effort is justified. After all, there are already many libraries, languages, and tools to support XML programming: why are we re-inventing the wheel and building from scratch yet another toolset for XML processing? XML itself contains multiple levels of abstraction features to facilitate efficient and flexible definitions of new vocabularies, such as the following:

- Different text encodings (iso-8859-1, utf-8, etc.) can be used in different XML documents; the encoding just needs to be specified at the very beginning.
- Entities can be defined and references be made so that foreign text can be easily represented (e.g., &copy; refers to ©).
- DTDs, which constrain the structure validity of a document, can be modified as needed in the document itself by, e.g., defining new content models for certain element types.

As such, identical information content can be marked up in different character encodings, with or without using specific entity references, and being constrained by different structural rules if so preferred. The resulting documents are syntactically different but their information content remains the same.

---

4 This increase is somewhat compiler dependent: when using the bytecode compiler, the optimized version is twice as fast as the original.
Likewise, documents marked up by XML can be, and often are, processed at different levels of abstraction. For example, an XML document need not be validated against its DTD if it is processed just for transcoding purposes or a simple search as in our previous example. A problem with many supporting libraries, languages, and tools for XML processing is that they operate at pre-defined, fixed abstraction levels.

On the one end of the spectrum, there is SAX (Brownell and Megginson 1998-2006) which is a low-level, event-driven interface for parsing XML documents. On the other end, there is, e.g., CDuce (Benzaken et al. 2003) which is a full-fledged functional language with an XML-native type system. Using SAX to write validating XML processors is a pain, while CDuce is not helpful for transcoding XML documents. Worse, there are great difficulties to mix-and-match different tools as they often use greatly different interfaces and even assume different processing models.

Take StAX (Java WSP Team 2005) and JAXB (Sun Microsystems 2004), both Java APIs for XML processing, as examples. StAX is designed for pull parsing of XML documents (Haustein and Slominski 2002-2006) suitable for streaming processing, while JAXB is for typeful binding of XML elements to Java objects suitable for in-memory processing. However, one cannot easily use StAX to scan an XML document to locate a specific element in the document and then use JAXB to bind the element to a Java object.

XML is a language to define other languages. The term XML also refers to a family of languages helpful for the definition and processing of XML documents (XLink, XML Schema, XSLT, XQuery, etc.). When contemplating altogether the issues of different abstraction levels, various processing models, and disparate XML vocabularies, it is a challenge to design and implement a consistent and modular tool chain for efficient and flexible XML processing. Our experience with Objective Caml in building the layered streaming XML processors, however, indicates that this challenge would be better met using a functional language with high-level module facility. There are existing XML programming tools implemented in functional languages, such as FXP in Standard ML (Neumann 2000), HaXml in Haskell (Wallace and Runciman 1999), SXML in Scheme (Kiselyov 2002), and Xml-Light in Objective Caml (Cannasse 2001-2005). These tools are libraries for parsing XML documents at pre-defined abstraction levels assuming specific processing models. Moving to another abstraction level or adapting a different processing model will require modification to the library code or adding layers of user code.

The use of modules in Objective Caml and other ML-like languages allow tool builders to create reusable components and check their interoperability at design time, at implementation time, as well as at deployment time. The components are modules with precise module types. At design time, the various module types are specified and refined as required by the software architecture. The compiler checks if the interfaces fit together. Note that higher-order module types, which specify modules that take in modules as input and generate a module as output, are extremely useful in piecing the components together. At implementation time, multiple modules adhering to the same module type are checked again. These modules may exhibit different performance characteristics, for example, and are to be used under different deployment environments. At deployment time, different implementations are assembled to build the application. Again, the Objective Caml module facility will check to see if the assembled pieces fit together. The piecing-together process is illustrated by a module-as-lego diagram in Figure 4. The separation of interface from implementation, the use of higher-order modules, and type-safety checking of module implementations to their interfaces, from our experience, are the essential language features that enable the design, implementation, and deployment of layered and reusable programming tools.

![Figure 4. A module-as-lego illustrative diagram](image-url)
A. DTD Preprocessing: An Example

We now give a detailed example showing how the dtdpp preprocessing tool converts a DTD into an Objective Caml module of signature MAPPING.

Our running example is the (non-normative) Mediawiki DTD, available from http://www.mediawiki.org, which describes the XML language used in the Wikipedia dumps we ran our benchmarks on. This very simple DTD (the first half of which is depicted in Figure 5) declares 24 element types, three of them with additional attributes.

In the generated code, each element type is mapped onto a constructor of a new type element. If an element type has no attributes, then neither has its constructor. Otherwise, the constructor takes a record containing all the needed attributes as an argument.

Thus, the preprocessor generates the following type definition:

```ocaml
type element = Mediawiki of mediawiki_type
| Siteinfo | Sitename | Base | Generator | Case
| Namespaces | Namespace of namespace_type | (....)
```

The record type definitions for element types mediawiki and namespace are as follows:

```ocaml
type namespace_type = { namespace’key : string }
type mediawiki_type =
{ mediawiki’xml_lang : string;
mediawiki’xsi_schemaLocation : string;
mediawiki’xmlns_xsi : string;
mediawiki’xmlns : string;
mediawiki’version : string }
```

Note that although we try to stay close to the original naming of elements and attributes, some compromises have to be made. For example, constructors always have to be capitalized in Objective Caml, and we have to take care to disambiguate the record field names, since the same attribute name could very well be used by multiple elements.

Using the above definitions, we can now define the type of XML tokens (which is the same for any DTD being processed):

```ocaml
type token = StartTag of element | CData of string
| EndTag | Meta of Normalizer.meta
```

The token type Meta is a catch-all for comments and processing instructions, which are parsed but not processed in any way.

The mapping’s task now is to convert elements from the Normalizer stream into such tokens:

```ocaml
type i = Normalizer.o

let map t = match t with
| Normalizer.StartTag (n, a) ->
  StartTag (map_element (n, a))
| Normalizer.CData txt -> CData txt
| Normalizer.EndTag -> EndTag
| Normalizer.Meta m -> Meta m
```

Now we can finally give the definition of the main mapping function:

```ocaml
let map i = match i with
| Normalizer.StartTag (n, a) ->
  StartTag (map_element (n, a))
| Normalizer.CData txt -> CData txt
| Normalizer.EndTag -> EndTag
| Normalizer.Meta m -> Meta m
```

Notice that all of the above code is not actually generated textually. Instead, the preprocessor directly builds the corresponding abstract syntax tree and hands it to the compiler backend for further processing.

This is both more efficient and more convenient for the user, who does not have to deal with temporary files.

---

Figure 5. The (non-normative) Mediawiki DTD