

Status Report: Layered Streaming XML Processing with Modules

Tyng-Ruey Chuang Max Schäfer

Institute of Information Science,
Academia Sinica, Taiwan

5th October 2007

So We Had This Problem...

Problem

From an XML dump of Wikipedia, extract the contents of articles about certain countries and cities.

- ▶ the dump is 8GB to 1TB in size
- ▶ extraction does not have to be real-time, but should be reasonably fast
- ▶ it should be doable on a PC

How It Should Work

- ▶ the dump is very simply structured: just a list of articles stored in the form

```
<page>
  <title> Article Title </title>
  ...
  <text> Article content... </text>
</page>
```

- ▶ the text does not contain any markup and only predefined entity references
- ▶ to find an article:
 1. go over the page elements one by one
 2. compare title, skip rest of page if it does not match
 3. otherwise, extract article contents

Approaches That Don't Work

- ▶ DOM-based parser: out of the question due to size of input
- ▶ streaming (pull or push) parser: still too slow; too much time is spent parsing input that is being skipped
- ▶ character-level processing: works but with obvious drawbacks

So we decided to roll our own.

Layered Streaming Parsing

- ▶ we would like to be able to parse XML data at different abstraction levels and switch between them as needed
- ▶ in our example, we might want to parse the input as
 1. a stream of characters
 2. a stream of raw tokens, e.g., start and end tags, and character data
 3. a stream of normalized tokens, where proper nesting of start and end tags is ensured
 4. 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
- ▶ a higher level stream can be built on top of a lower level one, but the underlying stream can be recovered when needed

Layered Stream Parsing (ctd.)

In the example: use high-level stream to find title of next article, switch back to low-level to skip over unneeded content.

```
page [      title [      Algeria      ]      ...  ]  
<page>    <title>      Algeria      </>      ...  </>  
<page>    <title>      Algeria      </title>    ...  </page>  
< p a g e > < t i t l e > A l g e r i a < / t i t l e > ... < / p a g e >
```

Layered Stream Parsing (ctd.)

In the example: use high-level stream to find title of next article, switch back to low-level to skip over unneeded content.

```
page [ title [ Algeria ] ... ]  
<page> <title> Algeria </> ... </>  
<page> <title> Algeria </title> ... </page>  
< p a g e > < t i t l e > A l g e r i a < / t i t l e > ... < / p a g e >
```

Layered Stream Parsing (ctd.)

In the example: use high-level stream to find title of next article, switch back to low-level to skip over unneeded content.

```
page [ title [ Algeria ] ... ]  
<page> <title> Algeria </> ... </>  
<page> <title> Algeria </title> ... </page>  
< p a g e > < t i t l e > A l g e r i a < / t i t l e > ... < / p a g e >
```

Layered Stream Parsing (ctd.)

In the example: use high-level stream to find title of next article, switch back to low-level to skip over unneeded content.

```
page [ title [ Algeria ] ... ]  
<page> <title> Algeria </> ... </>  
<page> <title> Algeria </title> ... </page>  
< p a g e > < t i t l e > A l g e r i a < / t i t l e > ... < / p a g e >
```

Layered Stream Parsing (ctd.)

In the example: use high-level stream to find title of next article, switch back to low-level to skip over unneeded content.

```
page [ title [ Algeria ] ... ]  
<page> <title> Algeria </> ... </>  
<page> <title> Algeria </title> ... </page>  
< p a g e > < t i t l e > A l g e r i a < / t i t l e > ... < / p a g e >
```

Layered Stream Parsing (ctd.)

In the example: use high-level stream to find title of next article, switch back to low-level to skip over unneeded content.

```
page [ title [ Algeria ] ... ]  
<page> <title> Algeria </> ... </>  
<page> <title> Algeria </title> ... </page>  
< p a g e > < t i t l e > A l g e r i a < / t i t l e > ... < / p a g e >
```

Layered Stream Parsing (ctd.)

In the example: use high-level stream to find title of next article, switch back to low-level to skip over unneeded content.

```
page [ title [ Algeria ] ... ]  
<page> <title> Algeria </> ... </>  
<page> <title> Algeria </title> ... </page>  
< p a g e > < t i t l e > A l g e r i a < / t i t l e > ... < / p a g e >
```

The Stream Interface

- ▶ we decided to use OCaml as implementation language
- ▶ streams should offer an interface similar to the standard library's `Stream.t`:
 - ▶ parameterized over element type
 - ▶ function `peek` to look ahead one element
 - ▶ function `next` to read one element and advance the stream
 - ▶ additionally, functions `up` and `down` to build higher-level stream on top of lower-level stream and recover lower-level stream

First Attempt: Polymorphic Datatypes

- ▶ implement streams as polymorphic datatypes (maybe records) of the form

`type ('a, 'b) stream`

where `'a` is the element type, and `'b` is the underlying stream type

- ▶ then we have `down : ('a, 'b) stream -> 'b`

- ▶ this is unsatisfactory:

- ▶ not every stream is built on top of a simpler one
- ▶ we cannot ensure that `'b` is in fact a stream

- ▶ interpreting `'b` as type of elements of underlying stream does not work either:

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

First Attempt: Polymorphic Datatypes

- ▶ implement streams as polymorphic datatypes (maybe records) of the form

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

where `'a` is the element type, and `'b` is the underlying stream type

- ▶ then we have `down : ('a, 'b) stream -> 'b`

- ▶ this is unsatisfactory:

- ▶ not every stream is built on top of a simpler one
- ▶ we cannot ensure that `'b` is in fact a stream

- ▶ interpreting `'b` as type of elements of underlying stream does not work either:

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

First Attempt: Polymorphic Datatypes

- ▶ implement streams as polymorphic datatypes (maybe records) of the form

`type ('a, 'b) stream`

where `'a` is the element type, and `'b` is the underlying stream type

- ▶ then we have `down : ('a, 'b) stream -> 'b`
- ▶ this is unsatisfactory:
 - ▶ not every stream is built on top of a simpler one
 - ▶ we cannot ensure that `'b` is in fact a stream
- ▶ interpreting `'b` as type of elements of underlying stream does not work either:

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

Second Attempt: Modules

- ▶ (not necessarily layered) streams are represented as modules with signature

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

- ▶ layered streams conform to an extended signature

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

- ▶ **LSTREAMs** can be used anywhere **STREAMs** are expected

Example

In our example, we have four stream modules:

1. `CharStream`: not layered; `elt` is `char`
2. `TokenizedStream`: layered on `CharStream`; `elt` is

```
type token=StartTag of string * (string * string) list
  | CData of string
  | EndTag of string
  | Meta of meta
```
3. `NormalizedStream`: layered on `TokenizedStream`; `elt` is similar
4. a custom module (tailored after the Wikimedia DTD): layered on `NormalizedStream`; `elt` is

```
type element = Page | Title | Text of text_type | ...
```

where `text_type` is a record to hold attributes

Stream Transformers

- ▶ implementing layered stream modules from scratch becomes old very quickly
- ▶ in our example, we need only two ways of building higher streams:
 1. element-wise mapping of an input stream to an output stream
 2. running a Mealy machine on an input stream to produce an output stream
- ▶ both are implemented as functors

Stream Mappings

- ▶ stream mappings are also represented as modules:

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

- ▶ a functor `Map` applies mappings to streams:

```
module Map (M:MAPPING) (S:STREAM with type elt = M.i) =
  (struct type elt = M.o
    type t = S.t
    module Base = S
    let peek (s:t) = option_map M.map (S.peek s)
    let next (s:t) = M.map (S.next s)
    let up (s:S.t) = s
    let down (s:t) = s
  end)
: LSTREAM with module Base = S and type elt = M.o
```

Mealy Machines

- ▶ Mealy machines are represented like this:

```
module type MEALY_MACHINE = sig
  type i
  type o
  type s
  val init : s
  val trans : s * i -> s * o option
  val finish : s -> o option
end
```

- ▶ a functor `Run` to apply a machine to a stream is not too difficult to implement

Implementing the XML Processors

- ▶ we start with the module `CharStream`
- ▶ a simple XML tokenizer is implemented as module `Tokenizer` : `MEALY_MACHINE`, so we can build the token stream:

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

- ▶ another machine `Normalizer` does the normalization
- ▶ to performe the final step, we can automatically extract the definition of a `MAPPING` transformer from an XML DTD, using a command like

```
ocamlc -c -pp 'camlp4o dtdpp.cmo' -impl mediawiki.dtd
```

- ▶ the resulting compiled module can be used to build the typed token stream:

```
module TTokenStream = Map(Mediawiki)(NTokenStream)
```

Using the XML Processors

- ▶ instances of different streams can successively be built up:

```
let chrstrm = CharStream.of_channel stdin
let tkstrm = TokenStream.up chrstrm
let ntkstrm = NTokenStream.up tkstrm
let ttkstrm = TTokenStream.up ntkstrm,
```

- ▶ now we can use `ttkstream` to parse our input at a high abstraction level
- ▶ if needed, we can switch down:

```
let ntkstrm' = TTokenStream.down ttkstrm
```

- ▶ when switching back up, there might be synchronization issues. . .

Performance

- ▶ time consumption grows linearly in input size
- ▶ a small benchmark of our library versus some other XML processing libraries in OCaml:

Article title	Original	Optimized	Xmlm	ocaml-xmlr
Lithuania	42s	32s	44s	33s
Nauru	52s	39s	53s	39s
Sudan	1m8s	51s	1m11s	52s
Uruguay	1m19s	1m0s	1m21s	1m0s
Zimbabwe	1m24s	1m4s	1m26s	1m5s
Peru	4m9s	3m9s	4m22s	3m15s

Issues

- ▶ we would have liked to use a stateless implementation, but performance penalty is severe
- ▶ using stateful implementation means that switching down and back up again is problematic
- ▶ the XML subset we handle is quite small

Conclusion

- ▶ we have discussed the implementation of a streaming XML processing framework for OCaml based on the module system
- ▶ its layered architecture allows for flexible parsing at different levels of abstraction
- ▶ the use of modules hides internals, alternative implementations could be substituted for parts of the library
- ▶ performance is promising

Generating MAPPINGS from DTDs

Here is a simple DTD together with the **MAPPING** it corresponds to:

```
<!ELEMENT page (title,text)>
<!ELEMENT title #PCDATA>
<!ELEMENT text #PCDATA>

<!ATTLIST text
  lang CDATA "en">
```

```
type text_type =
  { text'lang : string option }

type element = Page | Title
  | Text of text_type

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

type i = Normalizer.o
type o = token
```

Generating MAPPINGS from DTDs

Here is a simple DTD together with the **MAPPING** it corresponds to:

```
<!ELEMENT page (title,text)>
<!ELEMENT title #PCDATA>
<!ELEMENT text #PCDATA>

<!ATTLIST text
  lang CDATA "en">

type text_type =
  { text'lang : string option }

type element = Page | Title
  | Text of text_type

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

type i = Normalizer.o
type o = token
```