Recursion-free XPath 1.0 implementation

Oleg Parashchenko
olpa@ http://uucode.com/
Saint-Petersburg State University, Russia
Extended abstract proposal for GTTSE'2007

June 3, 2007

Abstract

XPath is a popular topic of research. Unfortunately, many papers interpret the XPath 1.0 standard incorrectly or use a subset of XPath. Unlike such works, Generative XPath *[GXPATH]* is a complete XPath 1.0 implementation, which is also of interest from a theoretical point of view. No recursion construction is used, except for list comprehension and morphisms. This paper gives informal semantics of the main operations, highlighting what is often missed, overlooked, or just not obvious for XML non-specialists.

XPath to code

XPath *[XPATH]* is a language for addressing parts of an XML document. In the first approximation, an XPath expression is either 1) an XPath expression, filtered by predicates, either 2) a series of steps. A step consists of an axis with a node test and zero or more predicates. Examples of XPath expressions:

```
child::*/child::para
/child::doc/child::chapter[@type='warning'][5]
```

The main data type in XPath is a node set, a collection of unordered, unique nodes. But due to technical implementation reasons, we use an ordered list, and also allow duplicates during intermediate calculations. XPath expressions can return scalar values, in this case we represent them as a list of one element.

XPath expressions are translated to code blocks with the free variables c, k and n, which are the context node, position and size, respectively. The parent code block is responsible for correct binding of these variables. Sorting the returned nodes in the document order is also the task of the parent block.

The most complex thing is compiling expressions with predicates: ($expr$)[$pred_1$]...[$pred_n$]. It is equivalent to ($expr$[$pred_1$])...[$pred_n$], therefore it is enough to show how $expr$[$pred$] is compiled.

```
(list-ec
  (:list Mc (gx:send-position-count-context
              (gx:sidoaed <<code for ''expr''>>))
  (if (gx:with-position-count-context (k n c) Mc
        (gx:predicate-to-boolean
          k <<code for ''pred''>>)))
  (gx:with-position-count-context (k n c) Mc c)))
```

The code is written in the programming language Scheme *[R5RS]*, the list comprehension library SRFI-42 *[EGNER]* is used. An equivalent code, written in the traditional list comprehension notation, might look this way.

$$
\begin{aligned}
[\ c \mid\ \ & Mc \leftarrow \text{gx:send-position-count-context} \\
& \qquad (\text{gx:sidoaed } \textit{«code for “expr”»}), \\
& (k, n, c) = \text{gx:with-position-count-context } Mc, \\
& \text{gx:predicate-to-boolean } k \ \textit{«code for “expr”»}\ ]
\end{aligned}
$$

- First, *expr* is evaluated.
- Before applying the predicates to the node set, the function `gx:sidoaed` sorts it in the document order. It's one of the most often overlooked tasks.
- The helper function `gx:send-position-count-context` annotates the nodes in the node set, adding the context position and the size of the set to each node.
- The macro `gx:with-position-count-context` unpacks the annotated node and binds the variables k, n and c.
- The result of evaluating *pred* needs interpretation. If it is a number, than the actual result is $k = pred$. Again, research papers sometimes ignore this step. Finally, the result is converted to a boolean. If it is *true*, the node is takes, otherwise discarded.

As an optimization, we can omit calling `gx:sidoaed` for $pred_2$ and the following predicates, as filtering retains the order of nodes in node sets.

The same approach is used to compile an XPath step. The only difference is that the call to `gx:sidoaed` isn't needed at all, as the axes query functions return the nodes in the correct order. Note that for four axes the correct order is the reverse document order.

Joining steps is easy: $step_1$/*steps* is compiled to:

```
(list-ec
  (:list c  «code for ''step1''»),
  (:list c2 «code for ''steps''»),
  c2))
```

The second non-trivial component of GXPath is the implementation of the function `gx:sidoaed` (the name means "sort in document order and eliminate duplicates"). It is nearly the literal transcription of the quicksort example from *[SORTING]*, expressed in the terms of the binary tree hylomorphism. The only difference is how the list is partitioned. One step of processing is comparing the median node with the current node.

- If the median node comes before the current node in the document, then the current node is put into the greater-than partition.
- If the median comes after the node, then the node is put into the less-than partition.
- If the median and the node are the same, the node is discarded.
- What's not always obvious: the median and the node can be from different documents. (For example, due to using variables or the function *document()*.) In this case we put the node into the greater-than partition.

Many functions (including type conversions such as `gx:string`), when applied to a non-empty node set, are actually applied to the first node of the node set, and the rest nodes are thrown away.

For the most cases, equality and relational expressions "*expr1 op expr2*" can be evaluated as:

$$\text{or } [\text{gx:type}(e_1)\ op\ \text{gx:type}(e_2) \mid e_1 \leftarrow expr_1, e_2 \leftarrow expr_2]$$

In short, the result is *true* when *op* is *true* for at least one node from *expr1* and one node from *expr2*, converted to some type. But when *expr1* (or *expr2*) is a boolean value, the second expression is converted to a boolean, and the booleans are compared.

The peculiarities of XPath are enumerated, and the rest is quite simple. Just don't forget about types and about using `gx:sidoaed` in appropriate places. For example, *expr1* + *expr2* is compiled to (`gx:unit` makes an one-element list from a scalar value):

```
(gx:unit
 (+ (gx:number (gx:sidoaed «code for "expr1"»))
    (gx:number (gx:sidoaed «code for "expr2"»)))))
```

Conclusion

There is a set of irregularities in XPath, but they are not fatal. It is possible to compile XPath expressions to such code, that the only recursion forms are list comprehensions and morphisms. Generative XPath is an implementation of this idea in Scheme. The system can be used as a testbed for applying algebra of programming to practice.

References

[EGNER] Sebastian Egner: Eager comprehensions in Scheme: The design of SRFI-42. In Proceedings of the ACM SIGPLAN 2005 Workshop on Scheme and Functional Programming. `http://repository.readscheme.org/ftp/papers/sw2005/egner.pdf`

[GXPATH] Oleg Parashchenko: Generative XPath, to appear in the proceedings of XML Prague 2007. `http://xmlhack.ru/protva/generative-xpath.pdf`

[R5RS] 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/`

[SORTING] Lex Augusteijn: Sorting morphisms. Advanced Functional Programming, pages 1-27, 1998. `http://citeseer.ist.psu.edu/augusteijn98sorting.html`

[XPATH] James Clark and Steve DeRose (eds.): XML Path Language (XPath) version 1.0. W3C Recommendation. `http://www.w3.org/TR/xpath`