Generic Programming for XML Tools

Johan Jeuring and Paul Hagg
Institute of Information and Computing Sciences, Utrecht University
The Netherlands
johan@jeuring.net

May 27, 2002

Abstract

A generic program is written once, and works on values of a large class of data types (or DTD's, schemas, structures, class hierarchies). This paper shows how generic programming can be used to implement XML tools such as XML editors, databases, and compressors, that depend on the DTD of an input XML document. The resulting tools usually perform better because knowledge of the DTD can be used to optimise the tools, and are smaller, because all DTD handling is dealt with in the generic programming compiler. The paper shows how an XML compressor is implemented as a generic program, and it discusses which other classes of XML tools would profit from an implementation as a generic program.

1 Introduction

Generic Programming. A generic program is a program that works on values of a large class of data types (or DTD's, schemas, structures, class hierarchies). An example generic program is equality: a function that takes two values, and returns a boolean value depending on whether or not the two argument values are equal. Equality is defined on many different kinds of data types, and it can be defined once and for all as a generic program. The generic program says that two values are equal provided the top nodes are equal, and that the top nodes have equally many children, which are pairwise equal. This paper describes the relation between generic programming and XML tools, and claims that generic programming is ideally suited for implementing many XML tools.

XML Tools. Since W3C released XML [36], the de facto data format standard on the web, hundreds of XML tools have been developed. There exist XML editors, XML databases, XML converters, XML parsers, XML validators, XML search engines, XML encryptors, etc. Information about XML tools is available from many sites, see for example [16, 18]. Flynn's book [15] provides a description of some oldre tools.

Usage of DTD's in XML Tools. An XML document is usually structured according to a Document Type Definition (DTD) or a schema. An XML document is valid with respect to a DTD if it is structured according to the rules (elements) specified in the DTD. So a validator is a tool that critically depends on a DTD. Some other classes of tools, such as the class of XML editors, also critically depend on the presence of a DTD. An XML editor can only support editing of an XML document well, for example by suggesting possible children or listing attributes of an element, if it knows about the element structure and attributes of elements. These classes of tools depend on a DTD, and do the same thing (modulo structure differences) for different DTD's. In that sense these tools are very similar to the generic equality function. We claim that many classes of XML tools are generic programs, or would benefit from being viewed as generic programs. We call such tools *DTD-aware XML tools* [38].

Generic Programming for XML Tools. Since DTD-aware XML tools are generic programs, it should help to implement such tools as generic programs. Implementing an XML tool as a generic program has several advantages:

- **Development time**. Generic programming supports the construction of type- (or DTD-) indexed programs. So all processing of DTD's and programs defined on DTD's can be left to the compiler, and does not have to be implemented by the tool developer. Furthermore, the existsing library of often used basic generic programs, for example, for comparing, encoding, etc., can be used in generic programs for XML tools.
- **Correctness**. An instance of a typeable generic program is typeable. This implies that valid documents will be transformed to valid documents, possibly structured according to another DTD. Thus generic programming supports constructing type correct XML tools.
- **Efficiency**. The generic programming compiler may perform all kinds of optimisations on the code which are difficult to conceive or implement by an XML tool developer.

This paper discusses which classes of XML tools are DTD aware, and how they can be implemented as generic programs.

Generic programming can also be used for XML tools that are not DTD aware, but then most of the above advantages no longer apply.

Organisation of this paper. This paper is organised as follows. Section 2 introduces generic programming. Section 3 discusses an XML tool for compressing XML files. This is a typical example of a DTD-aware XML tool. Section 4 discusses DTD awareness of many classes of XML tools. Section 5 discusses related work, and Section 6 concludes.

2 Generic Programming

This section introduces generic programming. A generic program is a program that works on values of a large class of data types. The introduction is brief and incomplete. For more extensive introductions to generic programming see [19, 5].

Data types. We will give examples of data types and programs on these data types in the functional programming language Haskell 98 [32]. Here are two examples of data types:

```
data List  =  Nil | Cons Char List
data Tree  =  Leaf Int | Bin Tree Char Tree
```

A list, a value of type `List`, is either empty, denoted by `Nil`, or it is a character `c` followed by a list `cs`, denoted by `Cons c cs`. A tree, a value of type `Tree`, is either a leaf containing an integer, or a binary node containing two subtrees and a character.

In general, a data type is defined by means of a number of constructors, where each constructor takes a number of arguments. In generic programming we view a data type as a labelled sum of possibly labelled products. The constructors of a data type are encoded as sum labels, record names are encoded as product labels. The example types are types of kind `*`. This implies that they do not take types as arguments, as in the following example:

```
data List a  =  Nil | Cons a List
```

Here `List` is a type constructor, which, when given a type `a`, constructs a type. The type constructor `List` has kind `* -> *`. Since there is no corresponding concept in DTD's we will not mention types with kinds different from kind `*` in this paper, although they play an important role in generic programming [19].

The equality function. We define function equality on both example data types. Two lists are equal if both are empty, or if both are nonempty, the first elements are equal, and the tails of the lists are equal.

```
eqList                          :: List -> List -> Bool
eqList Nil Nil                  =  True
eqList (Cons x xs) (Cons y ys)  =  eqChar x y && eqList xs ys
eqList _           _            =  False
```

where `eqChar` is the equality function on characters.

Two trees are equal if both are a leaf containing the same integer, or if both are nodes containing the same subtrees, in the same order, and the same characters.

```
eqTree                          :: Tree -> Tree -> Bool
eqTree (Leaf i) (Leaf j)        =  eqInt i j
eqTree (Bin l c r) (Bin v d w)  =  eqTree l v &&  eqChar c d && eqTree r w
eqTree _           _            =  False
```

Generic equality. The equality functions on `List` and `Tree` follow the same pattern: compare the top level constructors, and if they equal, pairwise compare their arguments. We capture this common pattern in a single generic definition by defining the equality function by induction on the structure of data types. So we define equality on sums, denoted by `:+:`, on products, denoted by `:*:`, and on base types such as `Unit` (the empty product, in Haskell denoted by `()`, used for nullary constructors), `Int` and `Char`. Furthermore, we have to define a generic function on the sum labels, represented by `Con`, and the product labels, represented by `Label`. We define the generic equality function in Generic Haskell below. Generic Haskell [8, 9] is an extension of the functional programming language Haskell with a construct for defining generic programs. We don't expect the reader to understand this definition in detail, we just want to show the form and conciseness of generic programs.

```
eq{|t::*|}                                  :: t -> t -> Bool
eq{|Unit|}          _          _            =  True
eq{|Int|}           i          j            =  eqInt i j
eq{|Char|}          c          d            =  eqChar c d
eq{|:+:|} eqA eqB (Inl x)   (Inl y)     =  eqA x y
eq{|:+:|} eqA eqB (Inl x)   (Inr y)     =  False
eq{|:+:|} eqA eqB (Inr x)   (Inl y)     =  False
eq{|:+:|} eqA eqB (Inr x)   (Inr y)     =  eqB x y
eq{|:*:|} eqA eqB (x:*:y)   (v:*:w)     =  eqA x v && eqB y w
eq{|Con _|} eqA   (Con x)   (Con y)     =  eqA x y
eq{|Label _|} eqA (Label x) (Label y)   =  eqA x y
```

Function `eq` is called a type-indexed value, since it is a function which when given a type returns a function on that type. Functions `eq{|List|}` and `eq{|Tree|}` are semantically equal to the functions `eqList` and `eqTree` defined above.

3 An XML compression tool

Compression for XML documents. XML documents may become (very) large because of the markup that is added to the content. Because of the repetitive structure of many XML documents, these documents can be compressed by quite a large factor.

Existing XML compressors. We know of four XML compressors:

- XMLZip [12]. XMLzip cuts its argument XML file (viewed as a tree) at a certain depth, and compresses the upper part separately from the lower part, both using a variant of zip or LZW [40]. This allows fast access to documents, but results in worse compression ratios compared with the following compressors.
- XMill [30]. XMill is a compressor that separates the structure of the XML document from the contents, and compresses structure and contents separately. Furthermore, it groups related data items (such as dates), and it applies semantic compressors to data items with a particular structure.
- ICT's XML-Xpress [25] is a commercial compression system for XML files that uses 'Schema model files' to provide support for files conforming to a specific XML schema. The basic idea of this system is the same as the idea underlying the compressor we will describe below.
- Millau [17] is a system for efficient encoding and streaming of XML structures. It also separates structure and content, and uses the associated schema (if present) for compressing the structure.

XML compression and DTD's. XML compressors are DTD aware. For example, consider the following small XML file:

```
<book lang="English">
<title>  Dead Famous </title>
<author> Ben Elton   </author>
<date>   2001        </date>
</book>
```

This file may be compressed by separating the structure from the data, and compressing the two parts separately. For compressing the structure we can make good use of the DTD. If we know how many elements, say n, appear in the DTD (the DTD for the above document contains at least 4 elements), we can replace each occurrence of the markup of an element in an XML file which is valid with respect to the DTD by $log_2\ n$ bits. This simple idea is the main idea behind the following tool, and has been described in the context of data conversion by Jansson and Jeuring [26, 27].

3.1 Implementing an XML compressor as a generic program

We have implemented an XML compressor, called XCOMPREZ, as a generic program. XCOMPREZ separates structure from contents, compresses the structure using knowledge about the DTD, and compresses the contents using a variant of zip [40]. Thus we replace each element, or rather, the pair of open and close keywords of the element, by the minimal number of bits required for the element given the DTD. We distinguish four components in the tool: a component that translates a DTD to a data type, a component that separates a value of any data type into its structure and its contents, a component that encodes the structure replacing constructors by bits, and a component for compressing the contents. Of course, we have also implemented a decompressor, but since it is dual, hence very similar, to the compressor, we omit its description. See the website for XCOMPREZ [28] for the latest developments on XCOMPREZ. The Generic Haskell source code for XCOMPREZ can be obtained from the website.

Translating a DTD to a data type. A DTD can be translated to one or more Haskell data types. For example, the following DTD:

```
<!ELEMENT book    (title,author,date,(chapter)*)>
<!ELEMENT title   (#PCDATA)>
<!ELEMENT author  (#PCDATA)>
<!ELEMENT date    (#PCDATA)>
```

```
<!ELEMENT chapter (#PCDATA)>
<!ATTLIST book lang (English | Dutch) #REQUIRED>
```

can be translated to the following data types:

```
data Book        =  Book Book_Attrs Title Author Date [Chapter]
data Book_Attrs  =  Book_Attrs { bookLang :: Lang }
data Lang        =  English  |  Dutch
newtype Title    =  Title String
newtype Author   =  Author String
newtype Date     =  Date String
newtype Chapter  =  Chapter String
```

We have used the Haskell library HaXml [38], in particular the functionality in the module DtdToHaskell to obtain a data type from a DTD, together with functions for reading (parsing) and writing (pretty printing) valid XML documents to and from a value of the generated data type. For example, the following value of the above DTD:

```
<book lang="English">
<title>  Dead Famous  </title>
<author> Ben Elton    </author>
<date>   2001         </date>
<chapter>Introduction </chapter>
<chapter>Preliminaries</chapter>
</book>
```

is translated to the following value of the data type `Book`:

```
Book Book_Attrs{bookLang = English}
     (Title "  Dead Famous  ")
     (Author " Ben Elton    ")
     (Date "   2001         ")
     [Chapter "Introduction "
     ,Chapter "Preliminaries"
     ]
```

An element is translated to a value of a data type using just constructors and no labelled fields. An attribute is translated to a value that contains a labelled field for the attribute. Thus we can use the Generic Haskell constructs `Con` and `Label` to distinguish between elements and attributes in generic programs.

Separating structure and contents. The contents of an XML document is obtained by extracting all `PCData` and all `CData` from the document. In Generic Haskell, the contents of a value of a data type is obtained by extracting all strings from the value. Besides the string, the path to the string through constructor and record label names is also recorded. For the above example value, we obtain the following result:

```
[["Book","Title","  Dead Famous  "]
,["Book","Author"," Ben Elton    "]
,["Book","Date","   2001         "]
,["Book",":","Chapter","Introduction "]
,["Book",":",":","Chapter","Preliminaries"]
]
```

The occurrences of the constructor `":"` appear because there is a list of chapters. The cons constructors `":"` are invisible in the example value. The generic function `extract` is defined as follows:

```
extract{|t :: *|}               :: t -> [[String]]
extract{|Unit|}      Unit       =  []
extract{|String|}    s          =  [[s]]
extract{|:+:|} eA eB (Inl x)    =  eA x
extract{|:+:|} eA eB (Inr y)    =  eB y
extract{|:*:|} eA eB (x:*:y)    =  eA x ++ eB y
extract{|Con c|}   e (Con b)    =  map ((conName d):) (e b)
extract{|Label l|} e (Label b)  =  map ((labelName m):) (e b)
```

where functions `conName` and `labelName` return the name (a string) of a constructor and a label, respectively. Note that it is possible to give special instances of a type-indexed function on a particular type, as with `extract{|String|}` in the above definition. This is another example of a type-indexed value.

The structure from an XML document is obtained by removing all `PCData` and `CData` from the document. In Generic Haskell, the structure, or `shape`, of a value of a data type is obtained by replacing all strings by units (empty tuples). Thus we obtain a value of a new data type, in which occurrences of the type `String` have been replaced by the type `()`. Such a type is a *type-indexed type* [20]. For example, the type we obtain from the data type `Book` is isomorphic to the following data type:

```
data SHAPEBook        =  SHAPEBook SHAPEBook_Attrs
                                   SHAPETitle
                                   SHAPEAuthor
                                   SHAPEDate
                                   [SHAPEChapter]
data SHAPEBook_Attrs  =  SHAPEBook_Attrs { bookLang :: SHAPELang }
data SHAPELang        =  SHAPEEnglish  |  SHAPEDutch
newtype SHAPETitle    =  SHAPETitle ()
newtype SHAPEAuthor   =  SHAPEAuthor ()
newtype SHAPEDate     =  SHAPEDate ()
newtype SHAPEChapter  =  SHAPEChapter ()
```

and the structure of the example value is

```
shapeBook = SHAPEBook (SHAPEBook_Attrs { bookLang = SHAPEEnglish })
                      (SHAPETitle ())
                      (SHAPEAuthor ())
                      (SHAPEDate ())
                      [SHAPEChapter ()
                      ,SHAPEChapter ()
                      ]
```

We only give the type of the generic function `shape`.

```
shape{| t :: * |}  ::  t -> SHAPE{| t |}
```

where `SHAPE{| t |}` is the type-indexed type which denotes the shape type of type `t`. Given the shape and the contents (obtained by means of function `extract`) of a value we obtain the original value by means of function `insert`:

```
insert{| t :: * |}  ::  SHAPE{| t |} -> [[String]] -> t
```

Encoding constructors. A constructor of a value of a data type is encoded as follows. First calculate the number n of constructors of the data type. Then calculate the position of the

constructor in the list of constructors of the data type. Finally, replace the constructor by the bit representation of its position, using $\log_2\ n$ bits. For example, in a data type with 6 constructors, the third constructor is encoded by `010`. Note that we start counting with 0. Furthermore, note that a value of a data type with a single constructor is represented using 0 bits. So the values of all types except for `String` and `Lang` in the example are represented using 0 bits. The generic function `encode` has the following (slightly simplified) type:

```
encode{| t :: * |}  ::  SHAPE{| t |} -> [Bit]
```

Compressing the contents. Finally, the contents of an XML document have to be compressed. At the moment we use zip to compress the strings obtained from the document. In the future, we envisage more sophisticated compression methods for the contents, similar to the methods used in XMill.

Huffman coding. A relatively simple way to improve XCOMPREZ it is to analyze some source files that are valid with respect to the DTD, count the number of occurrences of the different elements (constructors), and apply Huffman coding. We have implemented this rather simple extension [28].

3.2 Analysis

How does the compressor described in the previous subsection compare with the existing XML compressors? Since the goal of XMLZip is different from our and the other compressors goal (fast access to compressed documents), we only compare our compressor with XMill, XML-Xpress, and Millau.

Compression ration. XML-Xpress has been tested extensively against XMill, and achieves compression results that are about 80% better than XMill. We have performed some initial tests comparing XCOMPREZ and XMill. The tests are not representative, and it is impossible to draw hard conclusions from the results. However, on our test examples XCOMPREZ is 40% to 50% better than XMill. We think this improvement in compression ratio is considerable. As a schema contains more information about an XML document than a DTD, it is not surprising that our compressor does not achieve the same compression ratios as XML-Xpress. However, when we replace HaXml by a tool that generates a data type for a schema, we expect that we can achieve similar compression ratios as XML-Xpress. We have not been able to test against Millau, but from its description we expect that Millau achieves compression ratios that are a bit worse than the compression ratios achieved by XCOMPREZ, as Millau uses a fixed number of bits for some elements or attributes, independent of the DTD or Schema.

Code size. With respect to code size, the difference between XMill and XCOMPREZ is dramatic: XMill is written in almost 20.000 lines of C++. The main functionality of XCOMPREZ is less than 300 lines of Generic Haskell code. Of course, for a fair comparison we have to add some of the HaXml code (which is a library distributed together with almost all compiler and interpreters for Haskell), the code for handling bits, and the code for implementing the as yet unimplemented features of XMill. We expect to be able implement all of XMill's features in about 20% of the code size of XMill. We have not been able to obtain the source code of the (commercial) XML-Xpress.

4 DTD-awareness of XML Tools

This section discusses DTD awareness of classes of XML tools. We briefly introduce each of the classes of tools, and we discuss whether the class is DTD aware and whether the available tools make use of this fact. Furthermore, whenever applicable, we discuss where HaXml and Generic Haskell might help in implementing an XML tool. Note that some (classes of) XML tools develop very fast, and that some of the information given in this section may be out of date.

We will discuss the following classes of tools:

- XML converters, parsers, and validators
- XML databases and search tools
- XML editors
- XML encryptors
- XML publishing tools
- XML version management tools

The class of XML compressors does not appear in this list, but has been discussed in the previous section. This is not a complete list of classes of XML tools, but this list includes many of the XML tools in use today.

XML converters, parsers, and validators. Since they are very similar, we discuss the classes of XML converters, parsers, and validators together.

For an overview of XML parsers, see [16]. There are several variants of XML parsers. Most XML parsers parse an arbitrary XML document to a universal tree (a DOM). DTD's play no role when parsing: validity of an XML document with respect to a DTD is checked in a separate phase, for example by an XML validator. These parsers are not DTD aware.

Using Generic Haskell to develop an XML parser we would obtain a tool that takes a DTD as argument, and returns a parser for documents of the argument DTD. Thus the parser is automatically a validator. Any element that would turn the document into an invalid document would lead to a parse error. The HaXml library, in particular the module DtdToHaskell, contains a generic parser of this kind. Since this technology lies at the basis of our tools, we want to reimplement the read and show functions from DtdToHaskell in Generic Haskell, and add a module Schema2Haskell.

For an overview of XML converters, see [16]. There exist two classes of XML converters: XML to XML converters, and non-XML to XML converters. For both of these classes there exist specific and generic or DTD aware tools. An example of a specific non-XML to XML converter is RTF2XML [23]. Examples of generic non-XML to XML converters are Some2xml and Jedi [35, 22]. In both Some2xml and Jedi it is possible to specify patterns that are to be mapped on XML. If it were also possible to specify the result DTD, we would obtain a generic parser. We think an attribute grammar system, such as [4], is of more use here than a generic programming system, but future versions of Generic Haskell might offer functionality that is useful for implementing generic converters. The converters from XML to another format are discussed in the XML publishing tools section.

XML databases and search tools. XML documents can be searched by means of queries. Since XML query languages also play an important role in XML databases, we discuss XML databases and search tools under a common heading.

XML databases are used to store XML documents in a database. There are several ways to store an XML document in a database, but each of these can be classified as either structured or unstructured. The XML databases that store documents in a structured way are DTD-aware XML tools. The DTD is used to determine the tables in the database, and may be used to optimise queries etc. DTD awareness can be of great help when searching or querying XML documents: indexes can be built based on DTD's, subtrees can be skipped when searching, etc. According to Abiteboul et al [1] current databases are not DTD aware. However, the field of XML databases is developing fast, and we expect that there may already be DTD-aware XML databases. Since most of these tools are commercial tools, see for example [11], it is difficult to check DTD awareness, and to compare implementations.

XML editors. An XML editor supports editing XML documents. Most XML editors support viewing XML documents in different ways, and they suggest elements and attributes that may

be inserted at a given position. There are too many XML editors to list here. An incomplete list of XML editors can be found at the Proxima site [29]. An XML editor is a nice example of a DTD-aware XML tool. Most XML editors are DTD aware, and we think they should be. We have started on the core of a generic editor [39], but a lot of work remains in order to obtain a full-fledged XML editor. Again, as most of the existing XML editors are commercial tools, see for example [34, 2], it is difficult to compare implementations.

XML encryptors. An XML encryptor encrypts an XML document. Since the encrypted document gives nothing away of the structure of the input document, we see no application for generic programming here. Indeed, encryption would be weaker if it were based on the structure of a document.

XML publishing tools. An XML publishing tool, like for example Cocoon [3], takes an XML document and a target type on which to publish the document, and maybe a style sheet for this type, and returns a document which can be published. The tool traverses the input document, using the style sheet. Existing publishing tools are not DTD aware. DTD awareness might help in constructing a publishing tool, since knowledge about the DTD can be used to optimise the traversal.

XML version management tools. IBM has developed a tool called treediff [24] which compares two XML files and points out the differences between the two files. A similar tool has been developed by Dommit [13]. We think that it would not be difficult to implement such a tool in Generic Haskell. Chawathe [6] has developed algorithms for comparing hierarchally structured data (such as XML documents). It is easy to implement the minimum-cost edit distance algorithm given by Chawathe as a generic program, by printing values to the format expected by the algorithm, and parsing, to a value of the original type, the tree obtained by applying the minimum cost edit script to the printed argument. Types do not play an essential role in this algorithm.

5 Related work

Most XML tools are built using the DOM or the SAX for manipulating XML documents. Using the DOM or the SAX usually implies that an XML document does not have a type. It follows that these standards are not a lot of help when developing tools that critically depend on the type (DTD) of a document.

There are a number of XML-specific (query) languages, for example XDuce [21], XMλ [31, 33], XSLT [37], XML query algebras [14], Yatl [10]. In many of these languages, XML documents are native values. Each of these languages has a number of features, such as regular expression pattern matching, type inference, or regular expression types, that support the construction of programs that manipulate XML documents. However, none of these languages have features that support the construction of DTD-aware XML tools. We expect that extending XMλ with a construct that supports defining DTD-indexed functions would result in a very useful language for developing XML tools, but unfortunately an implementation of XMλ does not seem to exist.

6 Conclusions and future work

We have shown that the combination of HaXml and generic programming as in Generic Haskell is very useful for implementing DTD-aware XML tools. Using generic programming, such tools become easier to write, because a lot of the code pertaining to DTD handling and optimisation is obtained from the generic programming compiler, and the resulting tools are more effective, because they directly depend on the DTD. For example, DTD-aware XML compressors, such as XCompprez described in this paper, compress considerably better than XML compressors that don't take the DTD into account, such as XMill. Furthermore, our compressor is much smaller than XMill.

Conversely, because of the popularity of XML tools, we think XML tools are the killer ap for generic programming. It only remains to develop other DTD-aware XML tools, and a library that supports the development of XML tools using Generic Haskell. We have started on an XML editor in Generic Haskell, see [39], but a lot of work remains to be done. However, we hope to (further) develop at least our XML compressor, an XML editor, part of an XML version management tool, and an XML database this year.

Although we think Generic Haskell is very useful for developing DTD-aware XML tools, there are some features of XML tools that are harder to express in Generic Haskell. Some of the functionality in the DOM, such as the methods `childNodes` and `firstChild` in the `Node` interface, is hard to express in a typed way. Flexible extensions of type-indexed data types [20] might offer a solution to this problem. We think fusing HaXml, or a tool based on Schemas, with Generic Haskell, obtaining a 'domain-specific' language [7] for generic programming on DTD's or Schemas is a promising approach.

For tools that do not depend on a DTD we can use the untyped approach from HaXml to obtain a tool that works for any document. However, most of the advantages of Generic Programming no longer apply.

Acknowledgements. We want to thank the 2001 XML and Generic Programming class for investigating different classes of XML tools. Andres Löh helped us with the implementation of the XML compressor. Dave Clarke and Andres Löh commented on a previous version of this paper.

References

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web*. Morgan Kaufmann Publishers, 2000.

[2] Altova. XML Spy. Whitepaper available from `http://www.xmlspy.com`, 2002.

[3] The Apache Project. Cocoon. Available from `http://xml.apache.org/cocoon/`, 2002.

[4] Arthur Baars and Doaitse Swierstra. The Micro Attribute Grammar System. Available from `http://www.cs.uu.nl/groups/ST/Software/index.html`, 2001.

[5] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In S. Doaitse Swierstra, Pedro R. Henriques, and José N. Oliveira, editors, *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1999.

[6] S. Chawathe. Comparing hierarchical data in external memory. In *The Twenty-fifth International Conference on Very Large Data Bases*, pages 90–101, 1999.

[7] Dave Clarke. Towards GH(XML). Talk at the Generic Haskell meeting, see `http://www.generic-haskell.org/talks.html`, 2001.

[8] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001. Also available from `http://www.generic-haskell.org/`.

[9] Dave Clarke and Andres Löh. Generic Haskell, specifically. In *Proceedings Working Conference on Generic Programming*. Kluwer Academic Publishers, 2002. To appear.

[10] Sophie Cluet and Jérôme Siméon. YATL: a functional and declarative language for XML, 2000.

[11] X-Hive Corporation. X-Hive. Available from `http://www.xhive.com`, 2002.

[12] XMLSolutions Corporation. XMLZip. Available from `http://www.xmlzip.com/`, 1999.

[13] Dommitt. XML Diff and Merge Tool. Available from `http://www.dommitt.com/`.

[14] Mary Fernandez, Jérôme Siméon, and Philip Wadler. A semistructured monad for semistructured data. In *Proceedings ICDT*, 2001. Also available via `http://www.research.avayalabs.com/user/wadler/`.

[15] Peter Flynn. *Understanding SGML and XML Tools*. Kluwer Academic Publishers, 1998.

[16] Lars M. Garshol. Free XML tools and software. Available from `http://www.garshol.priv.no/download/xmltools/`.

[17] Marc Girardot and Neel Sundaresan. Millau: an encoding format for efficient representation and exchange of XML over the Web. In *IEEE International Conference on Multimedia and Expo (I) 2000*, pages 747–765, 2000.

[18] Google. Web Directory on XML tools. `http://www.google.com/`.

[19] Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2-3):129–159, 2002.

[20] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. To appear in Proceedings of the 6th Mathematics of Program Construction Conference, 2002.

[21] Haruo Hosoya and Benjamin C. Pierce. Xduce: A typed XML processing language. In *Third International Workshop on the Web and Databases (WebDB2000), volume 1997 of Lecture Notes in Computer Science*, pages 226–244, 1997,2000.

[22] Gerald Huck, Peter Fankhauser, Karl Aberer, and Erich J. Neuhold. Jedi: Extracting and synthesizing information from the web. In *Conference on Cooperative Information Systems*, pages 32–43, 1998.

[23] AlphaWorks IBM. RTF2XML. Available from `http://www.alphaworks.ibm.com/`, 2000.

[24] AlphaWorks IBM. XML treediff. Available from `http://www.alphaworks.ibm.com/tech/xmltreediff`, 2000.

[25] INC Intelligent Compression Technologies. XML-Xpress. Whitepaper available from `http://www.ictcompress.com/products\_xmlxpress.html`, 2001.

[26] Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In Doaitse Swierstra, editor, *ESOP'99*, volume 1576 of *LNCS*, pages 273–287. Springer-Verlag, 1999.

[27] Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

[28] Johan Jeuring and Paul Hagg. XComprez. Available from `http://www.generic-haskell.org/xmltools/XComprez/`, 2002.

[29] Johan Jeuring, Lambert Meertens, Steven Pemberton, Martijn Schrage, Gert van der Steen, and Doaitse Swierstra. Proxima - a generic presentation-oriented XML editor. Available from `http://www.cs.uu.nl/research/projects/proxima/`, 2002.

[30] Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153–164, 2000.

[31] Erik Meijer and Mark Shields. XMLambda: A functional language for constructing and manipulating XML documents. Available from `http://www.cse.ogi.edu/~mbs/`, 1999.

[32] Simon Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Simon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 — A non-strict, purely functional language. Available from `http://www.haskell.org/definition/`, 1999.

[33] Mark Shields and Erik Meijer. Type-indexed rows. In *The 28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, pages 261–275, 2001. Also available from `http://www.cse.ogi.edu/~mbs/`.

[34] SoftQuad Software. XMetal. Available from `http://www.xmetal.com`, 2002.

[35] Paul A. Tchistopolskii. Some2xml. Available from `http://www.pault.com/pault/old/Some2xml.html`, 1999.

[36] W3C. XML 1.0. Available from `http://www.w3.org/XML/`, 1998.

[37] W3C. XSL Transformations 1.0. Available from `http://www.w3.org/TR/xslt`, 1999.

[38] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *International Conference on Functional Programming*, pages 148–159, 1999.

[39] Jan de Wit. A technical overview of Generic Haskell. Master's thesis, Department of Information and Computing Science, Utrecht University, 2002. INF/SCR-02-03.

[40] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.