Generic Programming for XML Tools

Ralf Hinze, Johan Jeuring

August 27, 2002

XML Tools

Since W3C released XML, hundreds of XML tools have been developed.

- XML editors (XML Spy, XMetal)
- XML databases (XHive)
- XML converters, parsers, and validators
- XML version managament tools (XML Diff and Merge, IBM treediff)
- XML compressors
- XML encryption tools

Usage of DTD's in XML Tools

Some XML tools critically depend on, or would benefit from knowing, the DTD of a document. We call such a tool *DTD aware*. Examples of DTD-aware tools are:

- XML editors
- XML validators/parsers
- XML databases
- XML compressors
- XML version management tools

Generic Programming for XML Tools

Since DTD-aware XML tools are generic programs, it would help to implement such tools as generic programs:

- *Development time*.
  - DTD processing by the compiler.
  - Library of basic generic programs.
- *Correctness*. Valid documents will be transformed to valid documents, possibly structured according to another DTD.
- *Efficiency*. The generic programming compiler may perform all kinds of optimisations on the code.

Overview

This talk:

- DtdToHaskell (HaXml)
- XComprez, an XML Compressor

Compressing XML files

XML is a very verbose standard, and compression might considerably reduce the size of XML files.

I know of four XML compressors:

- XMLZip
- XMill
- Millau
- XML-Xpress

XMill

Consider the following book example:

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

XMill splits this document in contents and structure. The contents are collected in different containers for different elements, and the structure becomes:

```
book=#1, @lang=#2 title=#3 author=#4 date=#5 chapter=#6

#1 #2 C1 / #3 C2 / #4 C3 / #5 C4 / #6 C5 / #6 C5 / /
```

Using the DTD

Here is the book 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>
```

From the DTD you can see that you don't have to store the # 1, # 2, etc. information: the DTD enforces their presence. We only have to know how many chapters there are, and what the value of the `lang` attribute is. This implies you can compress better by taking the DTD into account.

XComprez

XComprez is a generic program for compressing values of data types. On a couple of example files it compresses almost twice as good as XMill. You can download XComprez from

```
http://www.generic-haskell.org/xmltools/XComprez
```

XComprez consists of four components:

- a component that translates a DTD to a data type
- a component that separates a value into structure and contents
- a component for compressing the structure
- a component for compressing the contents

HaXml

HaXml is a Haskell tool and a set of combinators for manipulating XML documents with Haskell. We use HaXml to generate a data type from a DTD, and to generate read and show functions from XML to values of the data type and vice versa. You can obtain HaXml from:

```
http://www.cs.york.ac.uk/fp/HaXml/
```

We will only use `DtdToHaskell.hs` from the HaXml library.

DtdToHaskell: DTD to data type

DtdToHaskell takes a DTD and generates a data type, and read and show functions from XML to values of the data type and vice versa. For example, for the book DTD you get:

```
data Book  =  Book Book_Attrs Title Author Date [Chapter]
  deriving (Eq,Show)
data Book_Attrs = Book_Attrs { bookLang :: Lang }
  deriving (Eq,Show)
data Lang = English  |  Dutch
  deriving (Eq,Show)
newtype Title    =  Title String    deriving (Eq,Show)
newtype Author   =  Author String   deriving (Eq,Show)
newtype Date     =  Date String     deriving (Eq,Show)
newtype Chapter  =  Chapter String  deriving (Eq,Show)
```

together with a function for reading (writing) an XML document into a value of this type.

DtdToHaskell: XML to value

The example XML document is read by the read function generated by DtdToHaskell into the following value:

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

Separating structure and contents

The contents of an XML document is obtained by extracting all `PCData` and `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. For the above example value, we obtain the following result:

```
["  Dead famous  "
," Ben Elton  "
,"   2001          "
,"Introduction "
,"Preliminaries"
]
```

Generic extraction

The list of strings is obtained by applying

```
extract{| Book :: * |} exBook
```

This is an instance of the following generic function.

```
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)    =  e b
extract{| Label l |} e (Label b)   =  e b
```

The shape of a book

The structure of 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). 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 ()
```

Another example of a type-indexed data type.

An example book shape

The shape of the example book is:

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

The type-indexed data type SHAPE

```
type SHAPE{| Unit |}        =  SH1 Unit
type SHAPE{| String |}      =  SHString Unit
type SHAPE{| :+: |} sa sb   =  SHEither (Sum sa sb)
type SHAPE{| :*: |} sa sb   =  SHProd (Prod sa sb)
type SHAPE{| Con |} sa      =  SHCon (Con sa)
type SHAPE{| Label |} sa    =  SHLabel (Label sa)
```

Here is its kind-indexed type:

```
type SHAPE{| t :: k |} :: KSHAPE{[ k ]}

KSHAPE{[  *   ]}  =  * -> *
KSHAPE{[ k->l ]}  =  KSHAPE{[ k ]} -> KSHAPE{[ l ]}
```

Function shape

Function shape returns the shape of a value, that is, it replaces all strings by units (empty tuples).

```
type Shape{[  *   ]} t  =  t -> SHAPE{| t |}
type Shape{[ k->l ]} t  =
  forall a. Shape{[ k ]} a -> Shape{[ l ]} (t a)

shape{| t :: k |}                  :: Shape{[ k ]} t
shape{| Unit |} u                  =  SH1 Unit
shape{| String |} s                =  SHString Unit
shape{| :+: |} sa sb (Inl a)       =  SHEither (Inl (sa a))
shape{| :+: |} sa sb (Inr b)       =  SHEither (Inr (sb b))
shape{| :*: |} sa sb (a :*: b)     =  SHProd (sa a :*: sb b)
shape{| Con c |} sa (Con b)        =  SHCon (Con (sa b))
shape{| Label l |} sa (Label b)    =  SHLabel (Label (sa b))
```

Function insert

Function insert takes a SHAPE value and a list of strings and inserts the strings at the right positions. It is the inverse of function extract.

```
type Insert{[  *   ]} t  =
  SHAPE{| t |} -> [String] -> (t,[String])
type Insert{[ k->l ]} t  =
  forall u . Insert{[ k ]} u -> Insert{[ l ]} (t u)

insert{| t :: k |}    :: Insert{[ k ]} t
insert{| Unit |} u    =  \ss -> (Unit,ss)
insert{| String |} s  =  \ss -> (head ss,tail ss)
insert{| :+: |} iA iB (SHEither (Inl a))  =
  \ss -> let {v = iA a ss} in (Inl (fst v),snd v)
...
```

Encoding constructors

A constructor of a value of a data type is encoded as follows.

- Calculate the number n of constructors of the data type.
- Calculate the position of the constructor in the list of constructors of the data type.
- 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 by 0 bits. So the values of all types except for `String` and `Lang` in the example are represented by 0 bits. The generic function `encode` has the following (slightly simplified) type:

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

The type of function `encode`

Here is the type of function `encode`. When computing the encoding, function `encode` also needs the list of constructors of the data type. For this purpose we use a dependency on `constructors`

```
dependency encode <- constructors encode

encode{| t :: k |}  ::  Encode{[ k ]} t

type Encode{[  *   ]} t     =
  Maybe [ConDescr] -> SHAPE{| t |} -> [Bit]
type Encode{[ k->l ]} t  =
  forall u . Encode{[ k ]} u     ->
             Constructors{[ k ]} u ->
             Encode{[ l ]} (t u)
```

The definition of function `encode` I

Encoding `Unit` and `String` is easy:

```
encode{| Unit |}    =  \_ _ -> []
encode{| String |}  =  \_ _ -> []
```

A sum is encoded by calculating the constructors, and using them in the right component of the sum.

```
encode{| :+:  |} cA eA cB eB  =
  let cR = cA ++ cB
  in  \cs -> let cs' = maybe cR id cs
             in  eA (Just cs') `shapejunc` eB (Just cs')
```

The definition of function `encode` II

On constructors, `encode` calculates the bits which are returned in the bit list.

```
encode{| Con c |} cA eA  =
 \cs (SHCon (Con a)) ->
   let cs' = maybe [c] id cs
   in  intinrange2bits (length cs')
                       (fromJust (elemIndex c cs'))
       ++ eA Nothing a
```

Note the argument `Nothing` in the call of `eA`: the constructors used for encoding below a constructor might come from another data type.

Finally, products are encoded by calculating the encodings of the components.

```
encode{| :*:  |} cA eA cB eB  =
 \_ (SHProd (a :*: b)) -> eA Nothing a ++ eB Nothing b
```

Function decode

Function decode is the inverse of function encode: given a list of bits it produces a value of a shape of a data type. It has the following kind-indexed type.

```
dependency decode <- constructors decode

type Decode{[ * ]} t  =
  [Bit] -> Maybe [ConDescr] -> (Maybe SHAPE{| t |},[Bit])
```

This function is used to calculate the shape of a value. Given the shape and a list of strings, we can use function insert to obtain a value of the data type corresponding to the original XML document. Showing this with the generated show function gives the original XML document.

Exercise: Huffman coding

If we know that some constructors occur much more often in a document than others, we can use this information to obtain better compression. For example, suppose we have the following list-like data type:

```
data FList a  =  Empty | Leaf a | Cons a (FList a)
```

The standard algorithm (XComprez) would use two bits for every constructor. However, any value of type `FList a` contains exactly one occurrence of either `Empty` or `Leaf`, and possibly many occurrences of `Cons`. So it is much better to use one bit for `Cons`, and two bits for the other constructors. It is not difficult to implement Huffman coding for this application.

Conclusions and Future work

Generic programming is useful for constructing XML tools: using HaXml/Generic Haskell allows for quick implementations of XML tools.

We are working on implementing several XML tools (XML editor,...)

Future work:

- Types get in the way in several places:
  - XSL rules contain match expressions that should be matched against any XML document.
  - Some of the DOM functions are hard to type in Generic Haskell.
- We want to be able to handle Schemata.