Scripting XML with Generic Haskell

Frank Atanassow

Dave Clarke

Johan Jeuring

institute of information and computing sciences, utrecht university

technical report UU-CS-2003-023

www.cs.uu.nl

Scripting XML with Generic Haskell

Frank Atanassow, Dave Clarke and Johan Jeuring

July 28, 2003

Abstract

A generic program is written once and works on values of many data types. Generic Haskell is a recent extension of the functional programming language Haskell that supports generic programming. This paper discusses how Generic Haskell can be used to implement XML tools whose behaviour depends on the DTD or Schema of the input XML document. Example tools include XML editors, databases, and compressors. Generic Haskell is ideally suited for implementing XML tools:

- Knowledge of the DTD can be used to provide more precise functionality, such as manipulations of an XML document that preserve validity in an XML editor, or better compression in an XML compressor.
- Generic Haskell programs are typed. Consequently, valid documents are transformed to valid documents, possibly structured according to another DTD. Thus Generic Haskell supports the construction of type correct XML tools.
- The generic features of Generic Haskell make XML tools easier to implement in a surprisingly small amount of code.
- The Generic Haskell compiler may perform all kinds of advanced optimisations on the code, such as partial evaluation or deforestation, which are difficult to conceive or implement by an XML tool developer.

By embedding Schema and XML data into Haskell data types, we show how Generic Haskell can be used as a generic XML processing language. We will demonstrate the approach by implementing an XML compressor in Generic Haskell.

1 Introduction

A generic program is a program that works for values of each type for a large class of data types (or DTDs, 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, but it can be defined once and for all as a generic program. The generic program for equality says that two values are equal provided their top nodes are equal, and that the top nodes have equally many children, which are pairwise equal. Other, classic, examples include functions like map, fold, parse, pretty-print, and zip. Such functions can be expressed in the programming language Generic Haskell, a recent extension of Haskell that supports generic programming, by writing cases for primitive types such as Int and for data types which encode the structure of types, such as sums, products, constructors, and the unit data type. This paper describes the relation between generic programming and XML tools, and argues that generic programming is ideally suited for implementing many XML tools.

XML [45] is the core technology of modern data exchange. An XML document is essentially a tree-based data structure, usually, but not necessarily, structured according to a Document Type Definition (DTD) or a Schema. Both DTDs and Schema form the basis of a type system for ensuring document validity. Since W3C released XML, thousands of XML tools have been developed, including XML editors, XML databases, XML converters, XML parsers, XML validators,

XML search engines, XML encryptors, XML compressors, etc. Information about XML tools is available from many sites, see for example [18, 20]. Flynn's book [15] provides a description of some older tools.

An XML document is valid with respect to a DTD if it is structured according to the rules (elements) specified in the DTD. Thus 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 essentially the same thing for different DTDs. In this 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* [51].

The goal of this paper is to show how generic programming can be used to construct DTD-aware XML tools. A number of alternative means by which XML tools proccess XML documents are possible:

- **XML API's**. A conventional API such as SAX or the W3C's DOM can be used, together with a programming language such as Java or VBScript, to access the components of a document after it has been parsed.
- **XML programming languages**. A specialized programming language such as W3C's XSLT [46], XDuce [24], Yatl [11], XMλ [35, 39], SXSLT [30], XStatic [17] etc. can be used to transform XML documents into other XML documents.
- **XML data bindings**. XML values can be "embedded" in an existing programming language by finding a suitable mapping between XML types and types of the programming language: an XML data binding [36]. Examples include HaXml for Haskell [51].

Using a specialized programming language or a data binding has significant advantages over the SAX or DOM approach:

- Parsing comes for free and can be optimized for a specific Schema.
- It is easier to implement, test and maintain software in the host language.
- Both specialized programming languages and data bindings can provide a higher level of abstraction by including domain specific concepts more or less directly in the programming language.

Furthermore, a data binding has the extra advantages that existing programming language technology can be leveraged, and that a programmer need not take XML particularities into account (though, this may be be a disadvantage, depending upon the application). Programming languages for which XML data bindings have been developed include Java [34], in which Schemas are translated to classes, and Python, as well as declarative programming languages such as Prolog [12] and Haskell [51, 43], in which XML DTDs are translated to data types. Using Haskell as the host language for an XML data binding offers the advantages of using a higher-order typed programming language.

DTD-aware XML tools can be considered to be generic programs, and can thus be implemented in Generic Haskell. Implementing an XML tool as a generic program has several advantages:

- **Correctness**. Generic Haskell programs are typed. Consequently, valid documents are transformed to valid documents, possibly structured according to another DTD. Thus Generic Haskell supports the construction of type correct XML tools.
- **Functionality**. Knowledge of the DTD can be used to provide more precise functionality, such as manipulations of an XML document that preserve validity in an XML editor, or better compression in an XML compressor.

- **Development time**. Generic programming supports the construction of type- (and hence also DTD-) indexed programs. So all processing of DTDs and programs defined on DTDs can be left to the compiler, and does not have to be implemented by the tool developer. The generic features of Generic Haskell make XML tools easier to implement in a surprisingly small amount of code. Furthermore, the existing library of frequently used basic generic programs, for example, for comparing, encoding, etc., can be used in generic programs for XML tools.
- **Efficiency**. The Generic Haskell compiler may perform all kinds of advanced optimisations on the code, such as partial evaluation or deforestation, which are difficult to conceive or implement by an XML tool developer.

The contributions of this paper are twofold. Firstly, we demonstrate examples of generic XML processing in Generic Haskell. The existing Haskell data binding translates DTDs to Haskell data types, but does not translate the considerably more complicated XML Schema. Our second contribution is to fill this gap by providing a translation of XML Schema types into Haskell, in the style of Wallace and Runciman's HaXml and Thiemann's WASH, including a demonstration of its soundness. We treat only the core of XS, in particular the fragment treated in Wadler, et al.'s formal semantics [4].

This paper is organised as follows. Section 2 introduces Generic Haskell. Section 3 describes how to implement XComprez, a generic compressor for XML documents. Section 4 describes a tool for translating an XML Schema to a set of Haskell data types. Section 5 constructs a parser for parsing an XML document into a Haskell value. Finally, Section 6 shows the correctness of the translation by proving a type soundness result.

2 An introduction to generic programming in Generic Haskell

A generic program is written once and is then applicable to values from a large class of data types. Generic programs are often defined over the structure of types. We give a brief introduction here, reviewing the fundamental structure of types and outlining how knowledge of this can be used to write generic programs which apply to all Haskell data types. Our introduction is brief; we refer the reader to more extensive background material available in the literature [22, 2].

Some data type fundamentals. The functional programming language Haskell 98 provides an elegant and compact notation for declaring data types [38]. In general, a data type is defined by means of a number of constructors, where each constructor takes a number of arguments. Here are two example data types:

```
data CharList = Nil | Cons Char CharList
data Tree     = Leaf Int | Bin Tree Char Tree.
```

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

These example types are of kind $\star$, meaning that they do not take any type arguments. The following type takes an argument; it is obtained by abstracting Char out of the CharList data type above:

```
data List a = Nil | Cons a (List a).
```

Here List is a type constructor, which, when given a type t, constructs the type List t. The type constructor List has kind $\star \to \star$. There is no corresponding concept in DTDs or Schema, though higher-kinded types will play a role later in this paper (and do so more generally in generic programming [22]).

To apply functions generically to all data types, we must first view data types as a labelled sum of possibly labelled products — all Haskell data types can be viewed this way. This encoding is based on the following data types:

```
data Con a      =  Con a
data Label a    =  Label a
data a :+: b    =  Inl a | Inr b
type a :*: b    =  (a, b)
data Unit       =  Unit.
```

The constructors of a data type are encoded as sum labels, represented by the type Con, record names are encoded as product labels, represented as the type Label. The choice between *Nil* and *Cons*, for example, is encoded as a sum using type :+:. Arguments such as the a and List a of the *Cons* are encoded as products using type :*:. In the case of *Nil*, an empty product, denoted by Unit, is used. Finally, primitive types such as Char are represented as themselves.

Now we can encode the above List type as

```
type List° a = Con Unit :+: Con (a :*: (List a)).
```

This representation is called a *structure type*; more details of the correspondence between these and Haskell types can be found elsewhere [22].

Generic functions can now be defined by writing functions (satisfying certain typing constraints) for each of the types which make up structure types. The functions are assembled together following the structure of the type to produce an instance of the generic function appropriate for the given type. To develop such a generic function, it is best to first consider a number of its instances for specific data types.

The equality function. We define the equality function on two of the example data types given above. Firstly, two character lists are equal if both are empty, or if both are non-empty, the first elements are equal, and the tails of the lists are equal.

```
eqCharList                          ::  CharList → CharList → Bool
eqCharList Nil Nil                  =   True
eqCharList (Cons x xs) (Cons y ys)  =   eqChar x y ∧ eqCharList xs ys
eqCharList _ _                      =   False
```

where *eqChar* is the equality function on characters.

Secondly, 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 CharList 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. This means that we define equality on sums (:+:), on products (:*:), and on base types such as Unit, Int and Char, as well as on the sum labels (Con) and the product

labels (Label). In Generic Haskell [9, 10], the generic equality function is rendered as follows:

$$
\begin{array}{lcl}
\textbf{type}\ \mathsf{Eq}\{\!|\star|\!\}\ \mathsf{t} & = & \mathsf{t} \to \mathsf{t} \to \mathsf{Bool} \\
eq\{\!|\mathsf{t} :: \kappa|\!\} & :: & \mathsf{Eq}\{\!|\kappa|\!\}\ \mathsf{t} \\
eq\{\!|\mathsf{Unit}|\!\}\ \_\ \_ & = & \mathit{True} \\
eq\{\!|\mathsf{Int}|\!\}\ i\ j & = & \mathit{eqInt}\ i\ j \\
eq\{\!|\mathsf{Char}|\!\}\ c\ d & = & \mathit{eqChar}\ c\ d \\
eq\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (\mathit{Inl}\ x)\ (\mathit{Inl}\ y) & = & eq\{\!|\mathsf{a}|\!\}\ x\ y \\
eq\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (\mathit{Inl}\ x)\ (\mathit{Inr}\ y) & = & \mathit{False} \\
eq\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (\mathit{Inr}\ x)\ (\mathit{Inl}\ y) & = & \mathit{False} \\
eq\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (\mathit{Inr}\ x)\ (\mathit{Inr}\ y) & = & eq\{\!|\mathsf{b}|\!\}\ x\ y \\
eq\{\!|\mathsf{a} :*: \mathsf{b}|\!\}\ (x, y)\ (v, w) & = & eq\{\!|\mathsf{a}|\!\}\ x\ v \wedge eq\{\!|\mathsf{b}|\!\}\ y\ w \\
eq\{\!|\mathsf{Con}\ \_\ \mathsf{a}|\!\}\ (\mathit{Con}\ x)\ (\mathit{Con}\ y) & = & eq\{\!|\mathsf{a}|\!\}\ x\ y \\
eq\{\!|\mathsf{Label}\ \_\ \mathsf{a}|\!\}\ (\mathit{Label}\ x)\ (\mathit{Label}\ y) & = & eq\{\!|\mathsf{a}|\!\}\ x\ y.
\end{array}
$$

We do not expect the reader to understand this definition in detail; we merely wish to demonstrate the form and conciseness of generic programs. The style in which we present generics functions is called *Dependency-style Generic Haskell* [33]. Function *eq* is called a type-indexed value, since it is a function which when given a type returns a function on that type. The type indices are given in {|funny brackets|} on the left-hand side of a definition. Instances of generic functions are given using the name of the generic function with the type at which it is applied, again within funny brackets. For example, the instances of the generic function *eq* for types CharList and Tree are denoted in Generic Haskell by $eq\{\!|\mathsf{CharList}|\!\}$ and $eq\{\!|\mathsf{Tree}|\!\}$, respectively, and are semantically equal to the functions *eqCharList* and *eqTree* defined above.

In addition to defining generic functions over the standard structure constructors, it is possible to override the default behaviour for specific types or even specific constructors [10]. Overriding the behaviour for specific types is used extensively later in this paper.

3 XComprez, a generic compressor for XML documents

As markup is added to the content, XML documents may become (very) large. Fortunately, due to the repetitive structure of many XML documents, these documents can be compressed by quite a large factor. This can be achieved if we use information from the DTD (or Schema) of the input document in the XML compressor. For example, consider the following small XML file (we consider only XML files which are valid with respect to a DTD):

```
<book lang="English">
<title>   Dead famous  </title>
<author>  Ben Elton    </author>
<date>    2001         </date>
<chapter> Nomination   </chapter>
<chapter> Eviction     </chapter>
<chapter> One Winner   </chapter>
</book>
```

In this file, 130 bytes are used for markup, and 90 bytes are used for content, not counting line breaks. This file may be compressed by separating the structure (markup) from the contents, and compressing the two parts separately. For compressing the structure we can make good use of the DTD. If we know how many different elements and attributes, say n, appear in the DTD, we can replace each occurrence of the markup of an element in a valid XML file by $log_2\ n$ bits. The DTD for the above document contains at least 6 elements and attributes, so we need at least 3 bits per element or attribute. Since there are seven occurrences of elements and attributes in the above document, we would need less than 3 bytes for the markup. Separating structure from contents, and replacing elements and attributes by smaller entities is one of the main ideas behind

XMill [32]. The (small) price that has to be paid is that the strings that appear in the data have to be separated by a special separator symbol. We improve on XMill by only recording markup if there is a choice between different tags to be made. In the above document, there is a choice for the language of the book, and the number of chapters it has. All the other elements are not encoded, since they are compulsory and can be inferred from the DTD. Using this idea, we need only 5 bits to represent the markup in the above document.

This section describes a tool based on this idea, which was first described by Jansson and Jeuring in the context of data conversion [27, 28]. We use HaXml [51] to translate a DTD to a data type, and write generic functions for separating the contents (the strings) and the structure (the constructors) of a value of a data type, and for encoding the structure of a value of a data type using information about the (number of) constructors of the data type.

In this section we implement an XML compressor as a generic program. The example shows how generic programming can be used to implement DTD-aware XML tools such as XML compressors, databases, and editors, that depend on the DTD of an input XML document.

3.1 Implementing an XML compressor as a generic program

We have implemented an XML compressor, called XCompprez, as a generic program. XCompprez separates structure from contents, compresses the structure using knowledge about the DTD, and compresses the contents using a compressor for strings. It works by replacing each element, or rather, the pair of open and close tags of the element, by the minimal number of bits required for the element given the DTD. Our tool consists of the following components:

- 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,
- a component for compressing the contents, and
- inverses for all of the above components.

The inverses of the components for encoding an XML document combine together to form a decompressor. As these are very similar to the components of the compressor, they have been omitted. See the website for XCompprez [29] for the Generic Haskell source code and for the latest developments on XCompprez.

Translating a DTD to a data type. A DTD can be translated to one or more Haskell data types. Later in the paper we will describe a tool for translating a Schema to a (set of) Haskell data type(s) and show how generic programming can be used in such a tool. But in this section we focus on DTDs. We use the Haskell library HaXml [51], in particular the functionality in the module DtdToHaskell, to obtain a (set of) data type(s) 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 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>
```

is 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.
```

The following value of the above DTD:

```
<book lang="English">
<title>   Dead famous  </title>
<author>  Ben Elton    </author>
<date>    2001         </date>
<chapter> Nomination   </chapter>
<chapter> Eviction     </chapter>
<chapter> One Winner   </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 "␣Nomination␣␣␣"
     , Chapter "␣Eviction␣␣␣␣␣"
     , Chapter "␣One␣Winner␣␣␣"
     ].
```

HaXml translates an element 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. For the above example value, we obtain the following result:

```
["␣␣␣Dead␣famous␣␣"
, "␣␣Ben␣Elton␣␣␣␣"
, "␣␣␣␣2001␣␣␣␣␣␣␣␣"
, "␣Nomination␣␣␣"
, "␣Eviction␣␣␣␣␣"
, "␣One␣Winner␣␣␣"
].
```

The generic function *extract*, which extracts all strings from a value of a data type, is defined as follows:

$$
\begin{array}{lcl}
\textbf{type}\ \mathsf{Extract}\{\!|\star|\!\}\ \mathsf{t} & = & \mathsf{t} \to [\mathsf{String}] \\
\mathit{extract}\{\!|\mathsf{t} :: \kappa|\!\} & :: & \mathsf{Extract}\{\!|\kappa|\!\}\ \mathsf{t} \\
\mathit{extract}\{\!|\mathsf{Unit}|\!\}\ \mathit{Unit} & = & [\,] \\
\mathit{extract}\{\!|\mathsf{String}|\!\}\ s & = & [s] \\
\mathit{extract}\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (\mathit{Inl}\ x) & = & \mathit{extract}\{\!|\mathsf{a}|\!\}\ x \\
\mathit{extract}\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (\mathit{Inr}\ y) & = & \mathit{extract}\{\!|\mathsf{b}|\!\}\ y \\
\mathit{extract}\{\!|\mathsf{a} :*: \mathsf{b}|\!\}\ (x, y) & = & \mathit{extract}\{\!|\mathsf{a}|\!\}\ x +\!\!+ \mathit{extract}\{\!|\mathsf{b}|\!\}\ y \\
\mathit{extract}\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ (\mathit{Con}\ a) & = & \mathit{extract}\{\!|\mathsf{a}|\!\}\ a.
\end{array}
$$

Note that it is possible to give a special instance of a generic function on a particular type, as with $\mathit{extract}\{\!|\mathsf{String}|\!\}$ in the above definition. Furthermore, because DtdToHaskell translates any DTD to a data type of kind $\star$, we could have defined *extract* just on data types of kind $\star$. However, higher-order kinds pose no problems, and the data binding for Schema given in the next section uses higher-order kinds. Finally, the operator $+\!\!+$ in the product case is a source of inefficiency. It can be removed using a standard transformation that lifts function *extract* to return a value of type $[\mathsf{String}] \to [\mathsf{String}]$.

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 the empty string. For example, the structure of the example value is

$$
\begin{array}{l}
\mathit{shapeBook} = \mathit{Book}\ (\mathit{Book\_Attrs}\{\,\mathit{bookLang} = \mathit{English}\,\}) \\
\qquad\qquad\quad (\mathit{Title}\ \texttt{""}) \\
\qquad\qquad\quad (\mathit{Author}\ \texttt{""}) \\
\qquad\qquad\quad (\mathit{Date}\ \texttt{""}) \\
\qquad\qquad\quad [\mathit{Chapter}\ \texttt{""} \\
\qquad\qquad\quad , \mathit{Chapter}\ \texttt{""} \\
\qquad\qquad\quad , \mathit{Chapter}\ \texttt{""} \\
\qquad\qquad\quad].
\end{array}
$$

The generic function *shape* returns the shape of a value of any data type.

$$
\begin{array}{lcl}
\textbf{type}\ \mathsf{Shape}\{\!|\star|\!\}\ \mathsf{t} & = & \mathsf{t} \to \mathsf{t}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{shape}\{\!|\mathsf{t} :: \kappa|\!\} & :: & \mathsf{Shape}\{\!|\kappa|\!\}\ \mathsf{t} \\
\mathit{shape}\{\!|\mathsf{Unit}|\!\}\ \mathit{Unit} & = & \mathit{Unit} \\
\mathit{shape}\{\!|\mathsf{String}|\!\}\ s & = & \texttt{""} \\
\mathit{shape}\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (\mathit{Inl}\ a) & = & \mathit{Inl}\ (\mathit{shape}\{\!|\mathsf{a}|\!\}\ a) \\
\mathit{shape}\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (\mathit{Inr}\ b) & = & \mathit{Inr}\ (\mathit{shape}\{\!|\mathsf{b}|\!\}\ b) \\
\mathit{shape}\{\!|\mathsf{a} :*: \mathsf{b}|\!\}\ (a, b) & = & (\mathit{shape}\{\!|\mathsf{a}|\!\}\ a, \mathit{shape}\{\!|\mathsf{b}|\!\}\ b) \\
\mathit{shape}\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ (\mathit{Con}\ a) & = & \mathit{Con}\ (\mathit{shape}\{\!|\mathsf{a}|\!\}\ a)
\end{array}
$$

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*:

$$
\mathit{insert}\{\!|\mathsf{t} :: \star|\!\} :: \mathsf{t} \to [\mathsf{String}] \to \mathsf{t}.
$$

The generic definition of function *insert* is omitted.

Storing strings in containers. Function *extract* returns the strings that appear in an XML document in inorder. Strings that have the same markup are often related. For example, strings that are marked up with `<date>` are likely to represent a date. We now describe a generic function that stores strings that appear in different elements in different so-called *containers*. Since strings

that appear in a container are likely to be similar, standard compression methods can compress a container with a larger factor than the single file obtained by storing all strings that appear in the XML document, as returned by function *extract* [32]. The generic function *containers* takes any value, and returns all containers for that value. A container is a pair consisting of a constructor name, and a list of strings that directly appear under that constructor.

```
type Container    =    (String, [String])
```

Function *containers* takes a value, a string denoting the current enclosing constructor, and the list of current containers, and if it encounters a string, it inserts it in the current containers.

```
type Containers{|*|} t    =    t → String → [Container] → [Container]

containers{|t :: κ|}                ::    Containers{|κ|} t
containers{|Unit|} Unit             =     λd cs → cs
containers{|String|} s              =     λd cs → insertc d s cs
containers{|a :+: b|} (Inl a)       =     λd cs → containers{|a|} a d cs
containers{|a :+: b|} (Inr b)       =     λd cs → containers{|b|} b d cs
containers{|a :*: b|} (a, b)        =
        λd cs → containers{|a|} a d (containers{|b|} b d cs)
containers{|Con c a|} (Con a)       =
        λd cs → containers{|a|} a (conName c) cs

insertc :: String → String → Container → Container
insertc c s []                  =    [(c, [s])]
insertc c s ((c', ss) : xs)     |    c ≡ c' = (c, s : ss) : xs
                                |    otherwise = (c', ss) : insertc c s xs
```

Encoding constructors. The constructor of a value 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. We start counting with 0. Observe that a value of a data type with a single constructor is represented using 0 bits. Consequently, the values of all types except for List, String and Lang in the running example are represented using 0 bits.

To implement this function, we assume there is a function *constructorPosition* which given a constructor returns a pair of integers: its position in the list of constructors of the data type, and the number of constructors of the data type.

```
constructorPosition :: ConDescr → (Int, Int)
```

Function *constructorPosition* can be defined by means of function *constructors*, which returns the constructor descriptions of a data type. This function is defined in the module *Collect*, which can be found in the library of Generic Haskell. (We omit the definitions of both function *constructors* and function *constructorPosition*.)

```
constructors{|t :: *|} :: [ConDescr]
```

The function *encode* takes a value, and encodes it as a value of type Bin, a list of bits, defined by

```
type Bin    =    [Bit]
data Bit    =    O | I

type Encode{|*|} t    =    t → Bin.
```

The interesting case in the definition of function *encode* is the constructor case. We first give the simple cases:

$$
\begin{array}{lcl}
encode\{\!|\mathsf{t} :: \kappa|\!\}\ \_ & :: & \mathsf{Encode}\{\!|\kappa|\!\}\ \mathsf{t} \\
encode\{\!|\mathsf{Unit}|\!\}\ \_\ Unit & = & [\,] \\
encode\{\!|\mathsf{String}|\!\}\ \_\ s & = & [\,] \\
encode\{\!|\mathsf{a} :*: \mathsf{b}|\!\}\ (a, b) & = & encode\{\!|\mathsf{a}|\!\}\ a +\!\!+ encode\{\!|\mathsf{b}|\!\}\ b \\
encode\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (Inl\ a) & = & encode\{\!|\mathsf{a}|\!\}\ a \\
encode\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (Inr\ b) & = & encode\{\!|\mathsf{b}|\!\}\ b.
\end{array}
$$

For Unit and String there is nothing to encode. The product case encodes the components of the product, and concatenates the results. The sum case strips of the *Inl* or *Inr* constructor, and encodes the argument.

The encoding happens in the constructor case of function *encode*. We use an auxiliary function *intinrange2bits* to calculate the bits for the position of the argument constructor in the constructor list, given the number of constructors of the data type currently in scope. The definition of *intinrange2bits* is omitted.

$$
encode\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ (Con\ a) \quad = \quad encodeCon\ c +\!\!+ encode\{\!|\mathsf{a}|\!\}\ a
$$

$$
\begin{array}{lcl}
encodeCon & :: & \mathsf{ConDescr} \to \mathsf{Bin} \\
encodeCon\ c & = & intinrange2bits\ (constructorPosition\ c) \\
intinrange2bits & :: & (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Bin}
\end{array}
$$

Huffman coding. A relatively simple way to (in many cases) improve XCOMPREZ 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. Function *countCon* counts constructors.

$$
\mathbf{type}\ \mathsf{CountCon}\{\!|\star|\!\}\ \mathsf{t} \quad = \quad \mathsf{t} \to [(\mathsf{ConDescr}, \mathsf{Int})]
$$

$$
\begin{array}{lcl}
countCon\{\!|\mathsf{t} :: \kappa|\!\} & :: & \mathsf{CountCon}\{\!|\kappa|\!\}\ \mathsf{t} \\
countCon\{\!|\mathsf{Unit}|\!\}\ Unit & = & [\,] \\
countCon\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (Inl\ a) & = & countCon\{\!|\mathsf{a}|\!\}\ a \\
countCon\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (Inr\ b) & = & countCon\{\!|\mathsf{b}|\!\}\ b \\
countCon\{\!|\mathsf{a} :*: \mathsf{b}|\!\}\ (a, b) & = & merge\ (countCon\{\!|\mathsf{a}|\!\}\ a)\ (countCon\{\!|\mathsf{b}|\!\}\ b) \\
countCon\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ (Con\ a) & = & add\ (c, 1)\ (countCon\{\!|\mathsf{a}|\!\}\ a)
\end{array}
$$

$$
\begin{array}{lcl}
merge & :: & [(\mathsf{ConDescr}, \mathsf{Int})] \to [(\mathsf{ConDescr}, \mathsf{Int})] \to [(\mathsf{ConDescr}, \mathsf{Int})] \\
add & :: & (\mathsf{ConDescr}, \mathsf{Int}) \to [(\mathsf{ConDescr}, \mathsf{Int})] \to [(\mathsf{ConDescr}, \mathsf{Int})]
\end{array}
$$

Using Huffman coding on the list returned by function *countCon* we obtain a table $[(\mathsf{ConDescr}, \mathsf{Bin})]$, which we use in function *encodeCon* to replace constructors.

Arithmetic coding. Using Huffman coding we always get a discrete number of bits per constructor. But if we are encoding lists of average length 10,000, we would like to use less than one bit per *Cons* constructor. Arithmetic encoding can be used to encode constructors with fractions of bits. We have used (Adaptive) Arithmetic Coding [3] to compress the constructors in a value of a data type. To use arithmetic coding, we have to add a model argument to function *encode*, which is used and updated whenever we encounter a constructor.

Compressing the contents. The last step of XCOMPREZ is to compress the contents of the XML document. At the moment we use the Unix compress utility [52] to compress the strings obtained from the document. In the future, we envisage more sophisticated compression methods for the contents.

3.2 Results

While XCOMPREZ is mainly a proof of concept, rather than an attempt at serious XML compression, it nonetheless performs quite well. We now describe some existing XML compressors and compare ours with one in particular, namely, XMill.

Related work. It is well known that structure-specific compression methods give much better compression results [5, 16, 14, 41] than conventional compression methods such as the Unix compress utility [52]. In the context of XML, a number of compressors exist. We briefly compare some of these with our approach:

- XMLZip [13] 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 [52]. This allows fast access to documents, but results in worse compression ratios compared with the following compressors.
- XMill [32] is a compressor that separates the structure of an 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 [26] 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 XCOMPREZ.
- Millau [19] 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.
- XMLPPM [7] uses a SAX encoding of an XML document, and and an online, adaptive, XML-conscious encoding based on Prediction by Partial Match (PPM) to compress XML documents.
- XGrind [44] is an XML compressor that preserves the original structure of an XML document in order to support queries on the compressed document.
- Cannataro et al. [6] describe lossy compression for XML documents: only parts of the document that are considered interesting to the user are preserved.

Analysis. The following analysis is very limited, because we have not been able to obtain the executables or the source code of most of the existing compressors, and did not have the time to compare against some others. We will only compare XCOMPREZ with XMill. Since the goals of XMLZip (fast access to compressed documents), XGrind (fast querying of compressed documents), and the lossy XML compression tool are different from our goal, we will not compare XCOMPREZ with these XML compressors.

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 some small test examples XCOMPREZ is between 0% and 50% better than XMill. This is not very surprising, since we use a very similar approach to compression as XMill, but use less space to represent markup. On the downside, XCOMPREZ runs slower than XMill.

From the description of Millau 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.

XML-Xpress has been tested extensively against XMill, and achieves compression results that are about 80% better than XMill. 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. When we replace HaXml by a tool that generates a data type for a schema, such as the one we describe in the following sections, we expect that we can achieve better compression ratios than at the moment.

With respect to code size, the difference between XMill and XCOMPREZ is dramatic: XMill is written in almost 20k lines of C++. The main functionality of XCOMPREZ is less than 500 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 every compiler and interpreter 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.

3.3 Conclusions

We have shown how to implement an XML compressor as a generic program. XCOMPREZ compresses better than for example XMill because it uses the information about an XML document present in a DTD. More importantly, XCOMPREZ is written in 500 lines of Generic Haskell code, whereas XMill is written in 20k lines of C++.

4 From Schema to Haskell

Though XML has achieved widespread popularity, the DTD formalism itself has been deemed to be too restrictive in practice, and this has motivated the development of alternative type systems for XML documents. The two most popular systems are the RELAX NG standard promulgated by OASIS [37], and the W3C's own XML Schema Recommendation [48, 49, 50]. Both systems include a set of primitive datatypes such as numbers and dates, a mechanism for combining and naming them, and ways of specifying context-sensitive constraints on documents.

We focus on XML Schema (or simply "Schema" for short—we use lowercase "schema" to refer to the actual type definitions themselves). We would like to write generic programs over documents conforming to schemas, and for this purpose require a translation of schemas to Haskell analagous to the HaXml translation of DTDs to Haskell described in Section 3.

We begin this section with a very brief overview of Schema syntax which highlights some of the differences between Schema and DTDs. Next, we give a more formal description of the syntax with an informal sketch of its semantics. With this in hand, we describe a translation of schemas to Haskell data types, and of schema-conforming documents to Haskell values.

Our translation and the variant syntax used here is based closely on the Schema formal semantics of Brown *et al.*, called the Model Schema Language (MSL) [4]; that treatment also forms the basis of the W3C's own, more ambitious but as yet unfinished, formal semantics [47]. We do not treat all features of Schema, but only the subset covered by MSL (except wildcards). This subset, however, arguably forms a representative subset and suffices for many Schema applications.

4.1 An overview of XML Schema

A schema describes a set of type declarations which may constrain the form of, and may affect the processing of, XML documents (values). Typically, an XML document is supplied along with a Schema file to a Schema processor, which parses and type-checks the document according to the declarations. This process is called *validation* and the result is a Schema value.

Syntax. Schemas are written in XML. As an example, consider the following declarations which define an element and a compound type for storing bibliographical information:

```
<element name="doc" type="document"/>
<complexType name="document">
  <sequence>
    <element ref="author" minOccurs="0"
```

```
                          maxOccurs="unbounded"/>
    <element ref="title"/>
    <element ref="year" minOccurs="0"/>
  </sequence>
</complexType>
```

This declares an element `doc` whose content is of type `document`, and a type `document` which consists of a sequence of zero or more `author` elements, followed by a mandatory `title` element and then an optional `year` element. (We omit the declarations for `author`, *etc.*) An example document which validates against `doc` is:

```
<doc>
  <author>Dorothy Sayers</author>
  <title>Murder Must Advertise</title>
  <year>1933</year>
</doc>
```

While they may have their advantages in large-scale applications, for our purposes XML and Schema syntax are rather too long-winded and irregular. We use an alternative syntax close to that of MSL [4], which is more orthogonal and suited to formal manipulation. In our syntax, the declarations above are written:

def doc[*document*] ;
def *document* = author* , title, year? ;

and the example document above is written:

doc[author["Dorothy␣Sayers"],
     title["Murder␣Must␣Advertise"],
     year["1933"]]

Differences with DTDs. Schemas are more expressive than DTDs in several ways. The main differences we treat here are summarized below.

1. Schema defines a larger number of primitive types, organized into a subtype hierarchy.
2. Schema allows the declaration of user-defined types, which may be used multiple times in the contents of elements.
3. Schema's notion of mixed content is more general than that available in DTDs.
4. Schema includes a notion of "interleaving" like SGML's `&` operator. This allows specifying that a set of elements (or attributes) must appear, but may appear in any order.
5. Schema has a more general notation for repetitions.
6. Schema includes two notions of subtype derivation.

We will treat these points more fully below, but first let us give a very brief overview of the Schema type system.

Overview. A document is typed by a *(model) group*; we also often refer to a model group as a *type*. An overview of the syntactic structure of groups is given by the grammar g.

g	::=		**group**
		ϵ	empty sequence
	|	g , g	sequence
	|	$\emptyset$	empty choice
	|	g | g	choice
	|	g & g	interleaving
	|	$g\{m, n\}$	repetition
	|	**mix**(g)	mixed content
	|	x	component name
m	::=	$\langle$natural$\rangle$	**minimum**

x	::=		
	|	$@a$	attribute name
	|	e	element name
	|	t	type name
	|	**anyType**	
	|	**anyElem**	
	|	**anySimpleType**	
	|	p	primitive
n	::=		**maximum**
		m	bounded
	|	∞	unbounded

This grammar is only a rough approximation to the actual syntax of Schema types. For example, in an actual schema, all attribute names appearing in an element's content must precede the subelements.

The sequence and choice operators should be familiar from DTDs and regular expressions. The forms $@a$, e and t are variables which reference, respectively, attribute, element and type bindings in the schema. We now consider the remaining features in turn.

Primitives. Schema defines some familiar primitives types like *string*, *boolean* and *integer*, but also more exotic ones (which we do not treat here) like *date*, *language* and *duration*. In most programming languages, the syntax of primitive constants such as string and integer literals is distinct, but in Schema they are rather distinguished by their types. For example, the data `"35"` may be validated against either *string* or *integer*, producing respectively distinct Schema values `"35"` $\in$ *string* and $35 \in$ *integer*. Thus, validation against a schema produces an "internal" value which depends on the schema involved.

The primitive types are organized into a hierarchy, via restriction subtyping (see below), rooted at **anySimpleType**.

User-defined types. An example of a user-defined type (or "group"), *document*, was given above. DTDs allow the definition of new elements and attributes, but the only mechanism for defining a new type (something which can be referenced in the content of several elements and/or attributes) is the so-called parameter entities, which behave more like macros than a semantic feature.

Mixed content. Mixed content allows mixing structured and unstructured text, as in a paragraph with emphasized phrases. The opposite of mixed content is called element-only content. In the DTD formalism, mixed content is specified by a declaration such as:

```
< !ELEMENT text ( #PCDATA | em )* >
```

This allows `em` elements to be interspersed with character data when appearing as the children of `text` (but not as descendants of children). In our syntax, the content type of `text` would be expressed as **mix**(em*).

To see how Schema's notion of mixed content differs from DTDs', observe that a reasonable translation of the DTD content type above is

$$[\,\mathsf{String} :\!+\!: [\![\mathtt{em}]\!]_G\,]$$

where $[\![\mathtt{em}]\!]_G$ is the translation of `em`. This might lead one to think that we can translate a schema type such as **mix**(g) more simply as $[\,\mathsf{String} :\!+\!: [\![g]\!]_G\,]$. However, this would not respect the

semantics of MSL, which implies that d matches **mix**(g) if *unmix*(d) matches g, where *unmix*(d) is obtained from d by deleting all character text at the top level. There are at least two problems with this translation. First, it is too generous, because while the schema type allows simple documents such as:

`"hello", e[], "world"` $\in$ **mix**(e)

it does not allow repeated occurrences, such as:

`"hello", e[], "world", e[],` "!" $\notin$ **mix**(e)

Thus, the user could construct a value which could not be written to an XML file conforming to the schema. Second, such a translation cannot account for more complex types such as **mix**$(\mathsf{e}_1, \mathsf{e}_2)$. A document matching such a type consists of two elements e_1 and e_2, possibly interspersed with text, but the elements *must occur in the given order*. This might be useful, for example, if one wants to intersperse a program grammar given as a type

def *module* = header, imports, fixityDecl*, valueDecl* ;

with comments: **mix**(*module*). An analogous model group is not expressible in the DTD formalism since groups involving `#PCDATA` can only appear in two forms, either alone:

`#PCDATA`

or in a repeated disjunction involving only element names:

`( #PCDATA |` e_1 `|` e_2 `|` $\cdots$ e_n `)*`

Interleaving. Interleaving is rendered in our syntax by the operator `&`, which behaves like the operator `,` but allows values of its argument types to appear in either order, *i.e.*, `&` is commutative. An example use is a schema which describes email messages.

def *email* = subject & from & to & body ;

Although interleaving does not really increase the expressiveness of Schema over DTDs, they are a much-welcomed convenience. Interleavings can be expanded to a choice of sequences, but these very rapidly become unwieldy. For example,

$[\![a \,\&\, b]\!] = a, b \mid b, a$

but

$[\![a \,\&\, b \,\&\, c]\!] \quad = \quad a, (b, c \mid c, b) \quad | \quad b, (a, c \mid c, a) \quad | \quad c, (a, b \mid b, a)$

(Note that $[\![a \,\&\, b \,\&\, c]\!] \neq [\![a \,\&\, [\![b \,\&\, c]\!]]\!]$!)

Repetition. In DTDs, one can express repetition of elements using the standard operators for regular patterns: *, $^+$ and ?. Schema has a more general notation: if g is a type, then $g\{m, n\}$ validates against a sequence of between m and n occurrences of documents validating against g, where m is a natural and n is a natural or ∞. Again, this does not really make Schema more expressive than DTDs, since we can expand repetitions in terms of sequence and choice, but the expansions are generally much larger than their unexpanded forms.

Derivation. XML Schema also supports two kinds of *derivation* (which we sometimes also call *refinement*) by which new types can be obtained from old. The first kind, called *extension*, is quite similar to the notion of inheritance in object-oriented languages. The second kind, called *restriction*, is an 'additive' sort of subtyping, roughly dual to extension.

As an example of extension, we declare a type *publication* obtained from *document* by adding fields at the end:

def *publication* **extends** *document* = journal | publisher ;

A value of type *publication* is a *document* followed by either a journal or publisher field.

Extension is slightly complicated by the fact that attributes are extended 'out of order'. For example, if types t_1 and t_2 are defined:

def $t_1 = @a_1$, e_1 ;
def t_2 **extends** $t_1 = @a_2$, e_2 ;

then the content of t_2 is:

$(@a_1$ & $@a_2)$, e_1 , e_2

As an example of restriction, we declare a type *article* obtained from *publication* by fixing some of the variability. For example, if an *article* is always published in a journal, we can write:

def *article* **restricts** *publication* = author* , title , year , journal ;

So a value of type *article* always ends with a journal, never a publisher, and the year is now mandatory. Note that, when we derive by extension we only mention the new fields, but when we derive by restriction we must mention all the old fields which are to be retained.

In both cases, when a type t' is derived from a type t, values of type t' may be used anywhere a value of type t is called for. For example, the document:

```
author["Patrik␣Jansson"],
author["Johan␣Jeuring"],
title["Polytypic␣Unification"],
year["1998"],
journal["JFP"]
```

validates not only against *article* but also against both *publication* and *document*.

Every type that is not explicitly declared as an extension of another is treated implicitly as restricting a distinguished type called **anyType**, which can be regarded as the union of all types. Additionally, there is a distinguished type **anyElem** which restricts **anyType**, and from which all elements are derived.

4.2 An overview of the translation

The object of the translation is to write Haskell programs on data corresponding to schema-conforming documents. At minimum, then, we expect the translation to satisfy a type-soundness result which ensures that, if a document validates against a particular schema type, then the translated value is typeable in Haskell by the translated type.

Let us outline the motivations and difficulties posed by the above-mentioned features. As a starting point, consider how we might translate regular patterns into Haskell datatypes.

$$\begin{aligned}
[\![\epsilon]\!]_G &= () & [\![\emptyset]\!]_G &= \mathit{Void} \\
[\![g_1\ ,\ g_2]\!]_G &= ([\![g_1]\!]_G, [\![g_2]\!]_G) & [\![g_1\ |\ g_2]\!]_G &= [\![g_1]\!]_G \mathbin{:+:} [\![g_2]\!]_G \\
[\![g^*]\!]_G &= [\,[\![g_1]\!]_G\,] & &= [\![g^+]\!]_G = ([\![g]\!]_G, [\![g^*]\!]_G) \\
[\![g?]\!]_G &= [\![g]\!]_G \mathbin{:+:} ()
\end{aligned}$$

This is the sort of translation employed by HaXml [51], and indeed we follow the same tack. In contrast, WASH [43] takes a decidedly different approach, encoding the state automaton corresponding to a regular pattern at the type level, and makes extensive use of type classes to express the transition relation.

Primitives. The primitives are basically translated to the corresponding Haskell types, wrapped by an isomorphism. For example,

```
data T_string mixity = T_string String.
```

The purpose of the mixity argument is explained below.

User-defined types. Types are translated along the lines described above, using products to model sequences and sums to model choices. Here are the exact types we use:

```
data Empty mixity       =   Empty
data Seq g1 g2 mixity   =   Seq (g1 mixity) (g2 mixity)
data None mixity  {- no constructors -}
data Or g1 g2 mixity    =   Or1 (g1 mixity) | Or2 (g2 mixity).
```

The translation takes each group to a Haskell type of kind $\star \to \star$ (we explain why when addressing mixed content in a moment):

$$[\![\epsilon]\!]_G = \mathsf{Empty} \qquad [\![g_1\ ,\ g_2]\!]_G = \mathsf{Seq}\ [\![g_1]\!]_G\ [\![g_2]\!]_G$$
$$[\![\emptyset]\!]_G = \mathsf{None} \qquad [\![g_1\ |\ g_2]\!]_G = \mathsf{Or}\ [\![g_1]\!]_G\ [\![g_2]\!]_G$$

As an example, the *document* type is translated as:

```
data T_document u = T_document
  (Seq Empty
       (Seq (Rep LE_E_author ZI)
            (Seq LE_E_title
                 (Rep LE_E_year (ZS ZZ)))) u).
```

Here the leading T_ serves to indicate that this declaration refers to the type *document*, rather than some element (or attribute) of the same name, which would be indicated by a prefix E_ (A_, respectively). The LE_ prefixes relate to derivation and are explained below. The Rep, ZS, ZZ, and ZI type relate to repetition, also explained below. Finally, the leading Seq Empty after the constructor *T_document* results from the translation of attributes and are also explained later.

Mixed content. The reason each group g is translated to a first-order type $\mathsf{t} :: \star \to \star$ rather than a ground type is that the argument, which we call the 'mixity', indicates whether a document occurs in a mixed or element-only context.[1] Accordingly, mixity is restricted to be either String or (). For example, $e[t]$ is translated as $\mathsf{Elem}\ [\![e]\!]_G\ [\![t]\!]_G\ ()$ when it occurs in element-only content, and $\mathsf{Elem}\ [\![e]\!]_G\ [\![t]\!]_G\ \mathsf{String}$ when it occurs in mixed content. The definition of this datatype

```
data Elem e g u = Elem u (g ())
```

stores with each element a value of type u corresponding to the text which immediately precedes a document item in a mixed context. (The type argument e is a so-called 'phantom type', serving only to distinguish elements with the same content g but different names.) Any trailing text in a mixed context is stored in the second argument of the *Mix* data constructor.

```
data Mix g u = Mix (g String) String
```

[1] We use the convention u for mixity because m is used for bounds minima.

For example, the document

$$\texttt{"one"}, \mathsf{e}_1[], \texttt{"two"}, \mathsf{e}_2[], \texttt{"three"} \in \mathbf{mix}(\mathsf{e}_1, \mathsf{e}_2)$$

is translated as

$$\mathit{Mix}\ (\mathit{Seq}\ (\mathit{Elem}\ \texttt{"one"}\ (\mathit{Empty}\ ()))\ (\mathit{Elem}\ \texttt{"two"}\ (\mathit{Empty}\ ()))\ \texttt{"three"}$$

Each of the group operators is defined to translate to a type operator which propagates mixity down to its children. For example:

$$\mathbf{data}\ \mathsf{Seq\ g1\ g2\ u} = \mathit{Seq}\ (\mathsf{g1\ u})\ (\mathsf{g2\ u})$$

There are three exceptions to this 'inheritance'. First, $\mathbf{mix}(g)$ ignores the context's mixity and always passes down a String type. Second, $e[g]$ ignores the context's mixity and always passes down a () type, because mixity is not inherited "across element boundaries." Finally, primitive content p always ignores its context's mixity because it is atomic.

An alternative to this treatment of mixed content is to translate mixed content with a separate semantic function, say $[\![-]\!]_{mG}$. Our treatment, though, has several advantages. For example, every type appearing in mixed content would also have to be translated differently. This means that there would be two versions of each type bound in the schema. We would also have to account for refinement of both versions of each type, leading to a dual hierarchy. Furthermore, the client programmer would have to write two functions each time she wanted to a process a type which could appear in both element-only and mixed contexts; in our translation, she need write only a single function which is polymorphic in the mixity type argument.

Interleaving. Interleaving is modeled essentially the same way as sequencing, except with a different abstract datatype.

$$\mathbf{data}\ \mathsf{Inter\ g1\ g2\ u} = \mathit{Inter}\ (\mathsf{g1\ u})\ (\mathsf{g2\ u})$$

An unfortunate consequence of this is that we lose the ordering of the document values.

For example, suppose we have a schema which describes a conference schedule where it is known that exactly three speakers of different types will appear. A part of such a schema may look like:

$$\mathbf{def}\ \mathsf{schedule[\,speaker\ \&\ invitedSpeaker\ \&\ keynoteSpeaker\,]}\,;$$

The schema processor should be able to determine the order in which schedule elements appeared, but since we do not track the permutation we cannot say what the document ordering was.

More commonly, since attribute groups are modeled as interleavings of attributes, this means in particular that schema processors using our translation do *not* have access to the order in which attributes are specified in an XML document.

Repetition. Repetitions $g\{m, n\}$ are modeled using a datatype

$$\mathsf{Rep}\ [\![g]\!]_G\ [\![m, n]\!]_B\ \mathsf{u}$$

and a set of datatypes modeling bounds:

$$[\![0, 0]\!]_B = \mathsf{ZZ} \qquad [\![0, m+1]\!]_B = \mathsf{ZS}\ [\![0, m]\!]_B$$
$$[\![0, \infty]\!]_B = \mathsf{ZI} \qquad [\![m+1, n+1]\!]_B = \mathsf{SS}\ [\![m, n]\!]_B$$

defined by[2]:

```
data Rep g b u   =  Rep (b g u)
data ZZ g u      =  ZZ
data ZI g u      =  ZI [g u]
data ZS b g u    =  ZS (Maybe (g u)) (Rep g b u)
data SS b g u    =  SS (g u) (Rep g b u).
```

Some sample translations are

$$[\![e\{2,4\}]\!]_G = Rep\ [\![e]\!]_G\ (SS\ (SS\ (ZS\ (ZS\ ZZ))))$$
$$[\![e\{0,\infty\}]\!]_G = Rep\ [\![e]\!]_G\ ZI$$
$$[\![e\{2,\infty\}]\!]_G = Rep\ [\![e]\!]_G\ (SS\ (SS\ ZI))$$

Derivation. Derivation poses one of the greatest challenges for the translation, since Haskell has no native notion of subtyping, though type classes are a comparable feature. We avoid relying on type classes here, however, because one of our goals is to develop a data representation which makes it easy to write Schema-aware programs in Generic Haskell. Since generic programs operate by recursing over the structure of a type, encoding the subtyping relation in a non-structural manner such as *via* the type class relation would be counterproductive.

This situation seems to be complicated by the need to support **anyType**. The **anyType** behaves as the *union* of all types, which immediately suggests an implementation in terms of Haskell datatypes: **anyType** should be translated to a datatype which has one constructor for each type which directly restricts it, the direct subtypes, and one constructor for values which are 'exactly' of type **anyType**.

In the case of our bibliographical example, we have:

```
data T_anyType mixity       =  T_anyType
data LE_T_anyType mixity    =  EQ_T_anyType (T_anyType mixity)
                            |  LE_T_anySimpleType (LE_T_anySimpleType mixity)
                            |  LE_T_anyElem (LE_T_anyElem mixity)
                            |  LE_T_document (LE_T_document mixity).
```

The alternatives *LE_* indicate the direct subtypes while the *EQ_* alternative is 'exactly' **anyType**. The *document* type and its subtypes are translated similarly:

```
data LE_T_document u        =  EQ_T_document (T_document u)
                            |  LE_T_publication (LE_T_publication u)
data LE_T_publication u     =  EQ_T_publication (T_publication u)
                            |  LE_T_article (LE_T_article u)
data LE_T_article u         =  EQ_T_article (T_article u).
```

When we *use* a Schema type in Haskell, we can choose to use either the 'exact' version, say T_document, or the version which also includes all its subtypes, say LE_T_document. Since Schema allows using a subtype of t anywhere t is expected, we translate all references to a variable to references to its LE_ variant. This explains why, for example, T_document refers to LE_E_author rather than E_author in its body.

What about extension? To handle the 'out-of-order' behavior of extension on attributes we define a function *split* which splits a type into a (longest) leading attribute group (ϵ if there is none) and the remainder. For example, if we recall our definitions above:

```
def t1 = @a1 , e1 ;
def t2 extends t1 = @a2 , e2 ;
```

[2]Actually the Haskell kind inferencer, which assumes unused type arguments are of kind $\star$, requires some hints to infer the correct kinds for these datatypes, so these datatypes have some extra, unused constructors which serve only to force the kinds of the arguments. We omit them here.

then $split(t_1) = (@a_1, \mathsf{e}_1)$ and, if t_2' is the 'extended part' of t_2, then $split(t_2') = (@a_2, \mathsf{e}_2)$. We then define the translation of t_2 to be

$$fst(split(t_1))\ \texttt{\&}\ fst(split(t_2'))\texttt{,}\ (snd(split(t_1))\ \texttt{,}\ snd(split(t_2')))$$

In fact, to accomodate extension, every type is translated this way. Hence T_document above begins with 'Seq Empty ...', since it has no attributes, and the translation of *publication*:

```
data T_publication u    =    T_publication
 (Seq (Inter Empty Empty)
      (Seq (Seq (Rep LE_E_author ZI)
                (Seq LE_E_title
                     (Rep LE_E_year (ZS ZZ))))
           (Or LE_E_journal LE_E_publisher)) u).
```

begins with 'Seq (Inter Empty Empty) ...', which is the concatenation of the attributes of *document* (namely none) with the attributes of *publication* (again none). Hence the attributes are accumulated at the beginning of the type declaration.

In contrast, the translation of *article*, which derives from *publication via* restriction, corresponds more directly with its declaration as written in the schema.

```
data T_article u = T_article
  (Seq Empty
       (Seq (Rep LE_E_author ZI)
            (Seq LE_E_title
                 (Seq LE_E_year LE_E_journal))) u)
```

This is because, unlike with extensions where the user only specifies the new fields, the body of a restricted type is essentially repeated as a whole.

Discussion. We have given an informal description of a translation of schema types and values into Haskell.

It is a bit surprising that MSL translates into Haskell as well as it does: indeed, the syntax of the Haskell types corresponding to MSL groups is almost exactly the same as that of the MSL groups themselves. We find the treatment of mixed content, which is often cited as an *ad hoc* feature of XML, *via* a mixity type parameter to be particularly elegant.

We have so far developed a prototype implementation of the translation and checked its correctness with a few simple examples and some slightly larger ones, such as the generic parser described in the next section, and a generic pretty-printer. The many wrapper isomorphisms involved in translated data make them rather unwieldy in standard Haskell. This is not really an issue when writing schema-aware XML tools in Generic Haskell, since most of the pattern-matching cases of a generic function involve only one wrapper at a time, but poses a problem for applications which depend on a specific schema.

There are some downsides to the translation. Although the handling of subtyping is straightforward and relatively usable, it does not take advantage of the 1-unambiguity constraint on Schema groups to factor out common prefixes. We will see in the next section that this has an impact on the efficiency of generic applications. Another issue is the use of unary encoding in repetition bounds, though this could be addressed by using a larger radix. Finally, there are some undesirable consequences of the fact that schema types, which obey equational laws, are always translated as abstract datatypes, which satisfy analagous laws only up-to-isomorphism.

Future work may involve extending the translation to cover more Schema features such as facets and wildcards, adopting the semantics described in more recent work [40], which more accurately models Schema's named typing, and exploiting the 1-unambiguity constraint to obtain a more economical translation.

4.3 A formalism for schemas

We now describe the syntax of documents and schemas formally and in a more complete fashion, and give an informal sketch of the semantics.

Documents A *document* is a sequence of *document items*, which may be attributes, elements or primitive textual data. Attribute and element content is assumed to be annotated by their intended type (s and t) in a previous normalization phase of the XML reader. This phase does not check that the content actually matches the indicated type.

d	::=		**document**	di	::=		**document item**
		ϵ	empty sequence			$a[s \ni d]$	attribute
	$\mid$	d , d	sequence		$\mid$	$e[t \ni d]$	element
	$\mid$	di	document item		$\mid$	c	text

The operators , and ϵ obey the equational laws for a monoid.

Model groups A document is typed by a *(model) group*. An overview of the operators which can occur in a group has been given in the previous section.

In fact, groups *per se* are not actually used in XML Schema; instead, only certain restrictions of the grammar g are allowed depending on the content *sort*, that is, whether it belongs to an attribute, element or type. For example, elements and interleaving are not allowed to occur in attribute content, and any attributes occurring in element content must come before all child elements.

Since the validation rules do not depend on the content sort we prefer to treat content in this more uniform fashion as it reduces some inessential complexity in our presentation. This does not entail any loss of "precision": Haskell programs employing our translation cannot violate the additional constraints imposed by content sort restriction because the translator program accepts as input only well-constrained schemas, and consequently the translated datatypes only have values which obey those constraints.

The XML semantics of model groups is given by specifying which documents validate against each group. A formal exposition of the semantics is given by Brown *et al.* [4], which we summarize here informally. (We prefer not to reiterate the formal rules for validation because, as we shall see in section 4.3.1, they are easily read off from the rules for our translation of XML documents into Haskell values.)

ϵ matches the empty document. g_1 , g_2 matches a document matching g_1, followed by a document matching g_2. $\emptyset$ matches no document. g_1 | g_2 matches a document which matches either g_1 or g_2. Unlike in MSL [4], we do *not* stipulate that these operators satisfy any equational laws; for example, sequence , is not held to be associative. Instead, a schema model group such as a , b , c is parsed in a right-associative manner as a , (b , c), and similarly for choice |. Note, however, that in contrast the document (value) operators , and ϵ *are* held to satisfy the laws for a monoid.

$g\{m, n\}$ matches any sequence of documents, each of which match g, provided the sequence is of length at least m, a natural number, and at most n, where n is a "topped natural": it may denote ∞. Arithmetic and ordering on naturals is extended to account for ∞ as follows: $n + \infty = \infty + n = \infty$ and $n \leqslant \infty$ is always true while $\infty < n$ is always false. We sometimes abbreviate repetitions using the syntax ?, * and + with the obvious translations.

An attribute $a[d]$ matches $a[g]$ iff d matches g. An element $e'[d]$ matches $e[g]$ iff d matches g and $e' <: e$ according to the refinement order $<:$. A document d matches $\mathbf{mix}(g)$ iff d' matches g, where d' is obtained from d by deleting all character data not contained in any child elements. A document matches a component name x if it matches the content group bound to the name x by the schema.

g_1 & g_2 behaves like g_1 , g_2 except that the subdocuments may appear in any order. A restriction on the syntax of schemas ensures that either: each g_i is of the form $e[g]$ or $e[g]\{0, 1\}$; or each g_i is of the form $a[g]$ or $a[g]\{0, 1\}$.

Any document d matches against **anyType**; similarly, any element $e[d]$ matches against **anyElem**.

Atomic datatypes p are given by the grammar:

$$p \quad ::= \quad \mathit{boolean} \mid \mathit{integer} \mid \mathit{double} \mid \mathit{string}$$

Only character data can match against an atomic datatype. A document c matches against p iff it matches against the textual representation of a value of that datatype. In particular, every c matches against a *string*. We remain imprecise about the textual representations of the remaining possibilities since they closely resemble the syntax for literals in Haskell.

XML Schema actually includes a much larger repertoire of built-in atomic datatypes and a notion of "facets" which allow implementing further constraints on values, but we do not treat these here.

4.3.1 Translating a schema to a data type

Some semantic functions involved in the translation of a schema to a datatype are given in Figure 1. The function $[\![-]\!]_G$ translates model groups, while $[\![-]\!]_P$, $[\![-]\!]_X$, $[\![-]\!]_B$ and $[\![-]\!]_{mix}$ translate primitive names, component names, repetition bounds and mixities respectively. The free variable Σ, which refers to the schema in question, is global to all these rules and described below. Note that each group is translated as a type of kind $\star \to \star$.

$$\begin{aligned}
[\![\epsilon]\!]_G &= \mathsf{Empty} & [\![g_1\ ,\ g_2]\!]_G &= \mathsf{Seq}\ [\![g_1]\!]_G\ [\![g_2]\!]_G \\
[\![\emptyset]\!]_G &= \mathsf{None} & [\![g_1\ \mathsf{|}\ g_2]\!]_G &= \mathsf{Or}\ [\![g_1]\!]_G\ [\![g_2]\!]_G \\
[\![g_1\ \&\ g_2]\!]_G &= \mathsf{Inter}\ [\![g_1]\!]_G\ [\![g_2]\!]_G & [\![g\{m,n\}]\!]_G &= \mathsf{Rep}\ [\![g]\!]_G\ [\![m,n]\!]_B \\
[\![a]\!]_G &= [\![a[\Sigma(a)]]\!]_G & [\![e]\!]_G &= [\![e[\Sigma(e)]]\!]_G \\
[\![a[s]]\!]_G &= \mathsf{Attr}\ [\![a]\!]_X\ [\![s]\!]_X & [\![e[t]]\!]_G &= \mathsf{Elem}\ [\![e]\!]_X\ [\![t]\!]_X \\
[\![\mathbf{mix}(g)]\!]_G &= \mathsf{Mix}\ [\![g]\!]_G & [\![t]\!]_G &= [\![t]\!]_X
\end{aligned}$$

$$\begin{aligned}
[\![\mathit{boolean}]\!]_P &= \mathsf{T_boolean} & [\![\mathit{integer}]\!]_P &= \mathsf{T_integer} \\
[\![\mathit{double}]\!]_P &= \mathsf{T_double} & [\![\mathit{string}]\!]_P &= \mathsf{T_string}
\end{aligned}$$

$$[\![\mathbf{elem}]\!]_{mix} = () \qquad [\![\mathbf{mix}]\!]_{mix} = \mathsf{String}$$

Figure 1: Formal translation of types of a schema Σ.

The Haskell types mentioned in 1 are given by declarations:

```
data Empty u        =  Empty
data Seq g1 g2 u    =  Seq (g1 u) (g2 u)
data Or g1 g2 u     =  Or1 (g1 u) | Or2 (g2 u)
data None u  {- no constructors -}
data Attr a g u     =  Attr (g u)
data Elem e g u     =  Elem u (g ())
data Mix g u        =  Mix (g String) String
data Inter g1 g2 u  =  Inter (g1 u) (g2 u)
data Rep g b u      =  Rep (b g u)
data T_string u     =  T_string String
data T_boolean u    =  T_boolean Bool
data T_integer u    =  T_integer Integer
data T_double u     =  T_double Double
```

To explain the translation of names, we need an abstract model of schemas and of Haskell modules. For the sake of readability, we prefer to remain a bit informal on some technical points.

The function $[\![-]\!]_X$ converts a schema *name* into a Haskell identifier. If we regard names and identifiers as strings, then this function is the identity except that it prepends a string indicating the name's sort: if x is a type name then $[\![x]\!]_X = \texttt{"T\_"}x$; if an element name, then $\texttt{"E\_"}x$; if an attribute name, then $\texttt{"A\_"}x$. For clarity, in the sequel, we omit the semantic brackets and simply write x for $[\![x]\!]_X$ when no ambiguity can arise.

Let G be the set of all model groups. A *schema* $\Sigma = (X, f, <^e_\Sigma, <^r_\Sigma)$ is a set of component names X paired with a map $f : X \to G$ and two binary predicates $<^e_\Sigma$ and $<^r_\Sigma$ over X which axiomatize the extension and restriction relations respectively. The refinement relation $<:$ is defined as the reflexive-transitive closure of $<^e_\Sigma \cup <^r_\Sigma$. We write $\Sigma(x)$ for $f(x)$. We assume X is disjoint from the primitive names like *string* but includes the distinguished type names **anyType**, **anySimpleType** and **anyElem**. Furthermore, we require that **anyType** $<^r_\Sigma$ **anySimpleType** and **anyType** $<^r_\Sigma$ **anyElem**, and that the schema is *well-formed*: for example, every name except **anyType** is either an extension or restriction of some other name.

By way of example, the following schema declarations in a schema Σ:

```
def    e   [g_e] ;
def   @a   [g_a] ;
def    t   = g_t ;
```

produce bindings:

$$\begin{array}{lcl} \Sigma(\mathsf{e}) & = & \mathsf{e}[g_e] \\ \Sigma(@a) & = & @a[g_a] \\ \Sigma(t) & = & g_t \end{array}$$

We define the function $split : G \to G \times G$ on model groups so that if $(g_1, g_2) = split(g)$ then g_1 is the longest prefix of attribute content of g, and g_2 is the remainder. For example:

$$split(a_1 \;\&\; a_2 \;\&\; \cdots\; a_n \;\texttt{,}\; g_2) = (a_1 \;\&\; a_2 \;\&\; \cdots\; a_n, g_2)$$

If $n = 0$ then $g_1 = \epsilon$.

Now let K be the set of all Haskell type terms of kind $\star \to \star$, and D be the set of datatype declarations. A *datatype* $d = (C, g) \in D$ is a set of constructor names C paired with a *partial* map $g : C \to K$, and we write $d(c)$ for $g(c)$. (If $d(c)$ is undefined, then the constructor is a constant.)

A *Haskell module* $H = (T, h)$ is a set of type names T paired with a map $h : T \to D$, and we write $H(x)$ for $h(x)$. We assume T is disjoint from standard Haskell names and the types declared above like Seq, and that, for all $d, d' \in cod(h)$, $dom(d) \neq dom(d')$ unless $d = d'$, *i.e.*, no

distinct datatypes in H share constructor names. Hence, if $H(\mathsf{t})(c) = F$ then c denotes a function $c :: \forall \mathsf{a} \,.\, F\ \mathsf{a} \to \mathsf{t}$ in module H.

By way of example, a Haskell module $H = (\{Ty\}, h)$ where $d = (\{C_1, C_2\}, g)$, $H(Ty) = d$, $d(C_1) = F$ and $d(C_2)$ is undefined would be realized as:

> **import** *Def* -- defines Empty, Seq, Or, *etc.*
> **data** Ty u = C_1 (F u) | C_2

We now give a set of conditions which describe a function $[\![-]\!]_S$ that, given any schema $\Sigma = (X, f, <^e_\Sigma, <^r_\Sigma)$, produces a well-kinded Haskell module $H = (T, h) = [\![\Sigma]\!]_S$.

1. for all names $x \in X$ and x, "LE_"$x \in T$
2. $[\![\Sigma]\!]_S$(**anyType**)(**anyType**) is undefined
3. $[\![\Sigma]\!]_S$(**anyElem**)(**anyElem**) is undefined
4. forall x, $[\![\Sigma]\!]_S$("LE_"x)("EQ_"x) $= x$
5. forall x, x' s.t. $x <^e_\Sigma x'$ or $x <^r_\Sigma x'$, $[\![\Sigma]\!]_S$("LE_"x')("LE_"x) = "LE_"x
6. forall x, x' s.t. $x <^r_\Sigma x'$ $[\![\Sigma]\!]_S(x)(x) = [\![\Sigma(x)]\!]_G$
7. forall x, x' s.t. $x <^e_\Sigma x'$ $[\![\Sigma]\!]_S(x)(x) = [\![(a_{x'}$ `&` $a_x)$ `,` $c_{x'}$ `,` $c_x]\!]_G$ where $(a_x, c_x) = \mathit{split}(\Sigma(x))$ and $(a_{x'}, c_{x'}) = \mathit{split}(\Sigma(x'))$

The first condition says that each schema name x produces two type names, a type $[\![x]\!]_X$ and a type "LE"$[\![x]\!]_X$; the first (let us call it the 'equational' version) is used to denote values of *exactly* type x, while the second (let us call it the 'down-closed' version) is used to denote values of type x or any of its subtypes. The next two conditions essentially say that the equational versions of **anyType** and **anyElem** carry no interesting information. Condition 4 says that the down-closed version of a type has a constructor which injects the equational version.

Condition 5 says that if x is an immediate subtype of x', then there is a constructor "LE_" x which injects the down-closed version of x into the down-closed version of x'. We call such constructors *axiomatic subtyping witnesses*; note that each instance of the refinement relation $<:$ is witnessed by a function which is expressible as either the identity or a composition of such axiomatic witnesses.

Conditions 6 and 7 express the way subtyping coercions work. Condition 6 says that the equational version of a type $[\![x]\!]_X$ obtained by restriction simply has a constructor which injects the content of x into $[\![x]\!]_X$, *i.e.*, just as in schema specifications, we do not try to factor restrictions to share any parts of a restricted type with its parent. Condition 7 expresses Schema's notion of extension, which reorders the content to bring together attribute content from the parent and child; in contrast to restriction, some simple factoring is done here *via* the *split* function.

Groups The value translation is given by the inference rules of Figure 2. The conclusion of each rule has the form $d \in_u g \Rightarrow v$, which can be read, "document d validates against type g producing Haskell value v" in mixity context u. (The schema and Haskell module(s) in question are left implicit.) This notation is a more readable alternative for describing a type-indexed value translation function $[\![-]\!]^{g,u}_V$:

$$d \in_u g \Rightarrow v \quad \equiv \quad [\![d]\!]^{g,u}_V = v$$

The soundness of this translation, shown in Section 6, ensures that $v :: [\![g]\!]_G\ [\![u]\!]_{mix}$.

Interleaving The interleaving rule uses a proposition of the form $d \overset{\text{inter}}{\mapsto} d_1; d_2$, defined in Figure 3, which can be read, "document items in d can be permuted to yield a pair of documents d_1 and d_2."

$$\text{Empty}\frac{}{\epsilon \in_u \epsilon \Rightarrow \mathit{Empty}} \qquad \text{Seq}\frac{d_1 \in_u g_1 \Rightarrow v_1 \quad d_2 \in_u g_2 \Rightarrow v_2}{d_1\,,\, d_2 \in_u g_1\,,\, g_2 \Rightarrow \mathit{Seq}\ v_1\ v_2}$$

$$\text{Choice(1)}\frac{d \in_u g_1 \Rightarrow v_1}{d \in_u g_1 \mid g_2 \Rightarrow \mathit{Or1}\ v_1} \qquad \text{Choice(2)}\frac{d \in_u g_2 \Rightarrow v_2}{d \in_u g_1 \mid g_2 \Rightarrow \mathit{Or2}\ v_2}$$

$$\text{Inter}\frac{d \overset{\text{inter}}{\mapsto} d_1; d_2 \quad d_i \in_u g_i \Rightarrow v_i}{d \in_u g_1\ \&\ g_2 \Rightarrow \mathit{Inter}\ v_1\ v_2} \qquad \text{Name}\frac{d \in_u g \Rightarrow v \quad g = \Sigma(x)}{d \in_u x \Rightarrow \texttt{"EQ\_"}[\![x]\!]_X\ ([\![x]\!]_X\ v)}$$

$$\text{Rep}\frac{d \in_{g\ u} \{m,n\} \overset{\text{rep}}{\Rightarrow} v}{d \in_u g\{m,n\} \Rightarrow \mathit{Rep}\ v} \qquad \text{Attr}\frac{d \in_{\mathbf{elem}} s \Rightarrow v}{a[d] \in_{\mathbf{elem}} a[s] \Rightarrow \mathit{Attr}\ v}$$

$$\text{Elem}\frac{d \in_u t \Rightarrow v \quad t <: t' \overset{\text{ref}}{\Rightarrow} f}{e[d] \in_{\mathbf{elem}} e[t'] \Rightarrow \mathit{Elem}\ ()\ (f\ v)} \qquad \text{Refine}\frac{d \in_u e \Rightarrow v \quad e <: e' \overset{\text{ref}}{\Rightarrow} f}{d \in_u e' \Rightarrow f\ v}$$

$$\text{Mix(1)}\frac{d \in_{\mathbf{mix}} g \Rightarrow v \quad c \overset{\text{str}}{\Rightarrow} v'}{d, c \in_{\mathbf{elem}} \mathbf{mix}(g) \Rightarrow \mathit{Mix}\ v\ v'} \qquad \text{Mix(2)}\frac{c \overset{\text{str}}{\Rightarrow} v \quad e[d] \in_{\mathbf{elem}} g \Rightarrow \mathit{Elem}\ ()\ v'}{c, e[d] \in_{\mathbf{mix}} g \Rightarrow \mathit{Elem}\ v\ v'}$$

Figure 2: Formal translation of values, part I: Documents.

$$\text{Inter-Empty}\frac{}{\epsilon \overset{\text{inter}}{\mapsto} \epsilon; \epsilon} \qquad \text{Inter-Seq}\frac{d_1 \overset{\text{inter}}{\mapsto} d_1'; d_1'' \quad d_2 \overset{\text{inter}}{\mapsto} d_2'; d_2''}{d_1\,,\, d_2 \overset{\text{inter}}{\mapsto} d_1'\,,\, d_2'; d_1''\,,\, d_2''}$$

$$\text{Inter-Item(1)}\frac{}{di \overset{\text{inter}}{\mapsto} di; \epsilon} \qquad \text{Inter-Item(2)}\frac{}{di \overset{\text{inter}}{\mapsto} \epsilon; di}$$

Figure 3: Formal translation of values, part II: Interleaving.

Repetition The rules for repetition given in Figure 4 employ propositions of the form $d \in_{g,u} \{m,n\} \stackrel{\text{rep}}{\Rightarrow} v :: [\![m,n]\!]_B$, where g is a group and u is a mixity, meaning that d validates against $g\{m,n\}$ producing a value v of type $[\![g\{m,n\}]\!]_G\ [\![u]\!]_{mix}$.

$$\text{Rep(0)}\frac{}{\epsilon \in_{g,u} \{0,0\} \stackrel{\text{rep}}{\Rightarrow} ZZ} \qquad \text{Rep(1)}\frac{\epsilon_u \in g\{0,m\} \Rightarrow v}{\epsilon \in_{g,u} \{0,m+1\} \stackrel{\text{rep}}{\Rightarrow} ZS\ Nothing\ v}$$

$$\text{Rep(1')}\frac{}{\epsilon \in_{g,u} \{0,\infty\} \stackrel{\text{rep}}{\Rightarrow} ZI\ []} \qquad \text{Rep(2)}\frac{d_1 \in_u g \Rightarrow v_1 \quad d_2 \in_u g\{m,n\} \stackrel{\text{rep}}{\Rightarrow} v_2}{d_1\,,\ d_2 \in_{g,u} \{m+1,n+1\} \stackrel{\text{rep}}{\Rightarrow} SS\ v_1\ v_2}$$

$$\text{Rep(3)}\frac{d_1 \in_u g \Rightarrow v_1 \quad d_2 \in_u g\{0,m\} \Rightarrow v_2}{d_1\,,\ d_2 \in_{g,u} \{0,m+1\} \stackrel{\text{rep}}{\Rightarrow} ZS\ (Just\ v_1)\ v_2} \qquad \text{Rep(3')}\frac{d_1 \in_u g \Rightarrow v_1 \quad d_2 \in_{g,u} \{0,\infty\} \stackrel{\text{rep}}{\Rightarrow} ZI\ v_2}{d_1\,,\ d_2 \in_{g,u} \{0,\infty\} \stackrel{\text{rep}}{\Rightarrow} ZI\ (v_1 : v_2)}$$

Figure 4: Formal translation of values, part III: Repetition.

Refinement The rules in Figure 5 define the witnesses to the refinement relation $<:$. If $g <: g' \Rightarrow f$ then f is a coercion which witnesses the fact that we can upcast a value of type $[\![g]\!]_G$ a to one of type $[\![g']\!]_G$ a, for any type a.

$$\text{Reflex}\frac{}{g <: g \Rightarrow id} \qquad \text{Trans}\frac{g <: g' \Rightarrow f \quad g' <: g'' \Rightarrow f'}{g <: g'' \Rightarrow f' \,.\, f}$$

$$\text{Res}\frac{x <^r_\Sigma x'}{x <: x' \Rightarrow \texttt{"LE\_"}x} \qquad \text{Ext}\frac{x <^e_\Sigma x'}{x <: x' \Rightarrow \texttt{"LE\_"}x}$$

Figure 5: Formal translation of values, part IV: refinement.

5 From XML documents to Haskell data

In this section we describe an implementation of the translation outlined in the previous section as a generic parser for XML documents. If t is a Haskell type corresponding to a Schema type t, then the generic value $gParse\{|\mathsf{t}|\}$ denotes a parser that accepts all and only documents validating against t. Consequently, generic programming is not only useful for implementing XML tools, it is already useful when constructing a data binding.

Rather than parse strings, we use a universal data representation Document which presents a document as a tree (or rather a forest):

```
type Document  =  [DocItem]
data DocItem   =  DText String
               |  DAttribute String Document
               |  DElement String Document
```

It is a simple matter to parse an XML document into this representation.

We use standard techniques [25] to define a set of monadic parsing combinators operating over Document. P a is the type of parsers that parse a value of type a. We omit the definitions here because they are straightfoward generalizations of string parsers.

```
return   ::   ∀a . a → P a
(>>=)    ::   ∀a b . P a → (a → P b) → P b
fmap     ::   ∀a b . (a → b) → P a → P b
runP     ::   ∀a . P a → Document → a
```

return and >>= are the unit and bind operations of the P monad; *runP p d* is the result of running parser p on a document d.

The type of generic parsers is given by the kind-indexed type GParse{|t|}:

```
type GParse{|*|} t   =   P t
```

The generic value *gParse*{|t|} denotes a parser which tries to read a document into a value of type t. We now describe its functionality on the various components of Schema.

```
gParse{|t :: κ|}      ::   GParse{|κ|} t
gParse{|String|}      =    pMixed
gParse{|Unit|}        =    pElementOnly
```

The first two cases handle mixities: *pMixed* optionally matches any *DText* chunk(s), while parser *pElementOnly* always succeeds without consuming input. Note that no schema type actually translates to Unit or String (by themselves), but these cases are used indirectly by the other cases.

```
gParse{|Empty u|}        =   return Empty
gParse{|Seq g1 g2 u|}    =   do doc1 ← gParse{|g1 u|}
                                doc2 ← gParse{|g2 u|}
                                return (Seq doc1 doc2)
gParse{|None u|}         =   mzero
gParse{|Or g1 g2 u|}     =   fmap Or1 gParse{|g1 u|}
                             <|> fmap Or2 gParse{|g2 u|}
```

Sequences and choices map closely onto the corresponding monad operators. p <|> q tries parser p on the input first, and if p fails attempts again with q, and *mzero* is the identity element for <|>.

```
gParse{|T_string|}     =   fmap T_string pText
gParse{|T_integer|}    =   fmap T_integer pReadableText
gParse{|T_double|}     =   fmap T_double pReadableText
gParse{|T_boolean|}    =   fmap T_boolean pReadableText
```

String primitives are handled by a parser *pText*, which matches any *DText* chunk(s). The function *pReadableText* parses integers, doubles, and booleans using the standard Haskell *read* function, since we defined our alternative schema syntax to use Haskell syntax for the primitives.

```
gParse{|Elem e g u|}     =   do mixity ← gParse{|u|}
                                let p = gParse{|g|} pElementOnly
                                elemt gName{|e|} (fmap (Elem mixity) p)
gParse{|Attr a g u|}     =   let p = gParse{|g|} pElementOnly
                             in attr gName{|a|} (fmap Attr p)
```

An element is parsed by first using the mixity parser corresponding to u to read any preceding mixity content, then by using the parser function *elemt* to read in the actual element. *elemt s p* checks for a document item *DElement s d*, where the parser p is used to (recursively) parse the

subdocument d. We always pass in $gParse\{\!|\mathsf{g}|\!\}\ pElementOnly$ for p because mixed content is 'canceled' when we descend down to the children of an element. Parsing of attributes is very similar.

This code uses an auxiliary type-indexed function $gName\{\!|\mathsf{e}|\!\}$ to acquire the name of an element; we omit its full definition here, since it has only one interesting case:

$$gName\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\} = drop\ 5\ (conName\ c)$$

This case makes use of the special Generic Haskell syntax $\mathsf{Con}\ c\ \mathsf{a}$, which binds c to a record containing syntactic information about a datatype. The right-hand side just returns the name of the constructor, minus the first five characters (something like $\mathsf{LE\_T\_}$), thus giving the correct attribute or element name as a string.

$$\begin{array}{lcl} gParse\{\!|\mathsf{Mix\ g\ u}|\!\} & = & \mathbf{do}\ doc \leftarrow gParse\{\!|\mathsf{g}|\!\}\ pMixed \\ & & \quad mixity \leftarrow pMixed \\ & & \quad return\ (Mix\ doc\ mixity) \end{array}$$

When descending through a Mix type constructor, we perform the opposite of the procedure for elements above: we ignore the mixity parser corresponding to u and substitute *pMixed* instead. *pMixed* is then called again to pick up the trailing mixity content.

$$\begin{array}{lcl} gParse\{\!|\mathsf{Rep\ g\ b\ u}|\!\} & = & fmap\ Rep\ gParse\{\!|\mathsf{b\ g\ u}|\!\} \\ gParse\{\!|\mathsf{ZZ\ g\ u}|\!\} & = & return\ ZZ \\ gParse\{\!|\mathsf{ZI\ g\ u}|\!\} & = & fmap\ ZI\ \$\ many\ gParse\{\!|\mathsf{g\ u}|\!\} \\ gParse\{\!|\mathsf{ZS\ g\ b\ u}|\!\} & = & \mathbf{do}\ x \leftarrow option\ gParse\{\!|\mathsf{g\ u}|\!\} \\ & & \quad y \leftarrow gParse\{\!|\mathsf{b\ g\ u}|\!\} \\ & & \quad return\ (ZS\ x\ (Rep\ y)) \\ gParse\{\!|\mathsf{SS\ g\ b\ u}|\!\} & = & \mathbf{do}\ x \leftarrow gParse\{\!|\mathsf{g\ u}|\!\} \\ & & \quad y \leftarrow gParse\{\!|\mathsf{b\ g\ u}|\!\} \\ & & \quad return\ (SS\ x\ (\mathsf{Rep}\ y)) \end{array}$$

Repetitions are handled using the familiar parser combinators *many* p and *option* p, which parse, respectively, a sequence of documents matching p and an optional p.

Most of the code handling interleaving is part of another auxiliary function, $gInter\{\!|\mathsf{t}|\!\}$, which has the kind-indexed type:

$$\mathbf{type}\ \mathsf{GInter}\{\!|\star|\!\} = \forall \mathsf{a}\,.\,\mathsf{PermP}\ (\mathsf{t} \to \mathsf{a}) \to \mathsf{PermP\ a}$$

Interleaving is handled using these permutation phrase combinators [1]:

$$\begin{array}{lcl} (<\|>) & :: & \forall \mathsf{a\ b}\,.\,\mathsf{PermP}\ (a \to b) \to \mathsf{P}\ a \to \mathsf{PermP}\ b \\ (<|?>) & :: & \forall \mathsf{a\ b}\,.\,\mathsf{PermP}\ (a \to b) \to (a, \mathsf{P}\ a) \to \mathsf{PermP}\ b \\ mapPerms & :: & \forall \mathsf{a\ b}\,.\,(a \to b) \to \mathsf{PermP}\ a \to \mathsf{PermP}\ b \\ permute & :: & \forall \mathsf{a}\,.\,\mathsf{PermP}\ a \to \mathsf{P}\ a \\ newperm & :: & \forall \mathsf{a\ b}\,.\,(a \to b) \to \mathsf{PermP}\ (a \to b) \end{array}$$

Briefly, a permutation parser $q :: \mathsf{PermP\ a}$ reads a sequence of (possibly optional) documents in any order, returning a semantic value a. Permutation parsers are created using *newperm* and chained together using <||> and <|?> (if optional). *mapPerms* is the standard map function for the PermP type. *permute* q converts a permutation parser q into a normal parser.

$$gParse\{\!|\mathsf{Inter\ g1\ g2\ u}|\!\} = permute\ \$\ (gInter\{\!|\mathsf{g2\ u}|\!\}\ .\ gInter\{\!|\mathsf{g1\ u}|\!\})\ (newperm\ Inter)$$

To see how the above code works, observe that:

$$\begin{array}{l} f1 = gInter\{\!|\mathsf{g1\ u}|\!\} :: \forall \mathsf{g1\ u\ b}\,.\,\mathsf{PermP\ (g1\ u \to b) \to PermP\ b} \\ f2 = gInter\{\!|\mathsf{g2\ u}|\!\} :: \forall \mathsf{g2\ u\ c}\,.\,\mathsf{PermP\ (g2\ u \to c) \to PermP\ c} \end{array}$$

Hence:

$$f2 \,.\, f1 :: \forall \mathsf{g1\ g2\ u\ c}\,.\,\mathsf{PermP\ (g1\ u \to g2\ u \to c) \to PermP\ c}$$

Note that if c is instantiated to Inter g1 g2 u, then the function type appearing in the domain becomes the type of the data constructor *Inter*, so we need only apply it to *newperm Inter* to get a permutation parser of the right type.

$$(f1 \,.\, f2)\ (newperm\ Inter) :: \forall \mathsf{g1\ g2\ u}\,.\,\mathsf{PermP\ (Inter\ g1\ g2\ u)}$$

Many cases of function *gInter* need not be defined because the syntax of interleavings in Schema is so restricted.

$$\begin{array}{lcl}
gInter\{\!|\mathsf{t} :: \kappa|\!\} & :: & \mathsf{GInter}\{\![\kappa]\!\}\ \mathsf{t} \\
gInter\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\} & = & (<\!\|\!> fmap\ Con\ gParse\{\!|\mathsf{a}|\!\}) \\
gInter\{\!|\mathsf{Inter\ g1\ g2\ u}|\!\} & = & gInter\{\!|\mathsf{g1\ u}|\!\}\,.\,gInter\{\!|\mathsf{g2\ u}|\!\} \\
 & & .\,mapPerms\ (\lambda f\ x\ y \to f\ (Inter\ x\ y)) \\
gInter\{\!|\mathsf{Rep\ g\ (ZS\ ZZ)\ u}|\!\} & = & (<\!|?\!>\ (Rep\ gDefault\{\!|\mathsf{(ZS\ ZZ)\ g\ u}|\!\} \\
 & & \quad ,fmap\ Rep\ gParse\{\!|\mathsf{(ZS\ ZZ)\ g\ u}|\!\}))
\end{array}$$

The key to understanding this declaration is the Con case. We see that an atomic type (an element or attribute name) produces a permutation parser transformer of the form $(<\!\|\!> q)$. The Inter case composes such parsers, so more generally we obtain parser transformers of the form:

$$(<\!\|\!> q_1 <\!\|\!> q_2 <\!\|\!> q_3 <\!\|\!> ...)$$

The Rep case is only ever called when g is atomic and the bounds are of the form ZS ZZ: this corresponds to a Schema type like $\mathsf{e}\{0,1\}$, that is, an optional element (or attribute).[3]

Discussion. Schema types correspond to tree languages satisfying a '1-unambiguity' rule analagous to the LL(1) restriction on string languages. But a glance at the representation of parsers:

$$\textbf{data}\ \mathsf{P\ a} = \mathsf{P}\{\,unP :: \mathsf{Document} \to \mathsf{Maybe\ (Document, a)}\,\}$$

shows that we have not used any of the well-known techniques [42, 31] that exploit determinism to optimize combinator parsing of LL(1) languages. The reason we cannot use these techniques is that the grammar defined by our translated schema types is not left-factored, *i.e.*, the grammar may include alternatives which share common prefixes.

For example, all three of the example types *document*, *publication* and *article* start with an (optional) author. The generic parser $gParse\{\!|\mathsf{LE\_T\_document\ u}|\!\}$ can be thought of as a sum (some alternatives are omitted):

$$gParse\{\!|\mathsf{T\_article\ u}|\!\} <\!\|\!> gParse\{\!|\mathsf{T\_publication\ u}|\!\} <\!\|\!> gParse\{\!|\mathsf{T\_document\ u}|\!\}$$

If we use a deterministic parser to parse a *document*, the parser would commit to the first branch as soon as it encountered an author element, because parsing an author consumes a prefix of the input. But the difference between these three types becomes evident only *after* that prefix, so if the input is actually a non-*article document*, we would get a parser error.

Our solution is to use a backtracking parser which tries the next alternative when the current alternative fails, and to arrange our translation so that more-specific types occur earlier in a sum than less-specific types. (This is always possible because every type has at most one supertype.)

The fact that we cannot exploit more efficient parser representations is a limitation of our translation. We could attempt to use deterministic parsers by altering the translation so that

[3]The Generic Haskell compiler does not accept the syntax $gInter\{\!|\mathsf{Rep\ g\ (ZS\ ZZ)\ u}|\!\}$. We actually define this case using $gInter\{\!|\mathsf{Rep\ g\ b\ u}|\!\}$, where b is used consistently instead of (ZS ZZ), but the function is only ever called when $\mathsf{b} = \mathsf{ZS\ ZZ}$.

the grammars are left-factored; this is not so hard to do for types derived by extension, since (ignoring the possibility of additional attributes) the declarations mention only a new suffix, but much harder for types derived by restriction, where essentially the entire body of the supertype is repeated. Even if we factor restrictions suitably, the resulting datatypes and datatype hierarchy would bear little resemblance to the original schema, so we have opted to use the less-efficient backtracking approach for now.

6 The correctness of the transformation

In this section we show that the translation of schemas into Haskell is correct. The two major results are that the translation is sound w.r.t. typing and that each instance of the subtyping relation is witnessed by a coercion. In this section we abbreviate the mixity translation function $[\![-]\!]_{mix}$ as $[\![-]\!]_m$.

The type soundness property says that if a document validates against a schema type, then the translation of the document is typeable against the translation of the schema type. More formally we have:

Proposition 1 (Type soundness) *Let $[\![-]\!]_G$ and $[\![-]\!]_V$ be respectively the type and value translations generated by a schema. Then, for all documents d, groups g and mixities u, $d \in_u g \implies [\![d]\!]_V^{g,u} \wedge [\![d]\!]_V^{g,u} :: [\![g]\!]_G\ [\![u]\!]_m$.*

Proof: By structural induction we show that the value translation rules preserve the type translation. Doing structural induction over the values amounts to annotating the value translation rules with explicit types. Thus we write $d \in_u g \Rightarrow v :: t$ to mean $d \in_u g \implies [\![d]\!]_V^{g,u} \wedge [\![d]\!]_V^{g,u} :: [\![g]\!]_G\ [\![u]\!]_m$. By writing

$$\frac{A_1 \quad \cdots \quad A_n}{A_0}$$

we mean: if A_1, ...and A_n, then A_0. Thus our compact format neatly presents the required structural induction proofs.

We present the annotated rules below.

$$\text{Empty}\ \frac{}{\epsilon \in_u \epsilon \Rightarrow \mathit{Empty} :: \mathsf{Empty}\ [\![u]\!]_m}$$

$$\text{Seq}\ \frac{d_1 \in_u g_1 \Rightarrow v_1 :: [\![g_1]\!]_G\ [\![u]\!]_m \qquad d_2 \in_u g_2 \Rightarrow v_2 :: [\![g_2]\!]_G\ [\![u]\!]_m}{d_1\,,\, d_2 \in_u g_1\,,\, g_2 \Rightarrow \mathit{Seq}\ v_1\ v_2 :: \mathsf{Seq}\ [\![g_1]\!]_G\ [\![g_2]\!]_G\ [\![u]\!]_m}$$

$$\text{Choice(1)}\ \frac{d \in_u g_1 \Rightarrow v_1 :: [\![g_1]\!]_G\ [\![u]\!]_m}{d \in_u g_1\ \mathsf{|}\ g_2 \Rightarrow \mathit{Or1}\ v_1 :: \mathsf{Or}\ [\![g_1]\!]_G\ [\![g_2]\!]_G\ [\![u]\!]_m}$$

$$\text{Choice(2)}\ \frac{d \in_u g_2 \Rightarrow v_2 :: [\![g_2]\!]_G\ [\![u]\!]_m}{d \in_u g_1\ \mathsf{|}\ g_2 \Rightarrow \mathit{Or2}\ v_2 :: \mathsf{Or}\ [\![g_1]\!]_G\ [\![g_2]\!]_G\ [\![u]\!]_m}$$

$$\text{Inter}\ \frac{d \overset{\text{inter}}{\mapsto} d_1; d_2 \qquad d_i \in_u g_i \Rightarrow v_i :: [\![g_i]\!]_G\ [\![u]\!]_m}{d \in_u g_1\ \mathsf{\&}\ g_2 \Rightarrow \mathit{Inter}\ v_1\ v_2 :: \mathsf{Inter}\ [\![g_1]\!]_G\ [\![g_2]\!]_G\ [\![u]\!]_m}$$

$$\text{Name}\ \frac{d \in_u g \Rightarrow v :: [\![g]\!]_G\ [\![u]\!]_m \qquad g = \Sigma(x)}{d \in_u x \Rightarrow \texttt{"EQ\_"}[\![x]\!]_X\ ([\![x]\!]_X\ v) :: \texttt{"LE\_"}[\![x]\!]_X\ [\![u]\!]_m}$$

$$\text{Rep}\ \frac{d \in_{g,u} \{m,n\} \overset{\text{rep}}{\Rightarrow} v :: [\![m,n]\!]_B\ [\![g]\!]_G\ [\![u]\!]_m}{d \in_u g\{m,n\} \Rightarrow \mathit{Rep}\ v :: \mathsf{Rep}\ [\![g]\!]_G\ [\![m,n]\!]_B\ [\![u]\!]_m}$$

$$\text{Attr}\ \frac{d \in_{\mathbf{elem}} s \Rightarrow v :: [\![s]\!]_G}{a[d] \in_{\mathbf{elem}} a[s] \Rightarrow \mathit{Attr}\ v :: \mathsf{Attr}\ [\![a]\!]_X\ [\![s]\!]_G}$$

$$\text{Elem}\ \frac{d \in_u t \Rightarrow v :: [\![t]\!]_G \qquad t <: t' \overset{\text{ref}}{\Rightarrow} f :: \forall \mathsf{a}\,.\,[\![t]\!]_G\ \mathsf{a} \to [\![t']\!]_G\ \mathsf{a}}{e[d] \in_{\mathbf{elem}} e[t'] \Rightarrow \mathit{Elem}\ ()\ (f\ v) :: \mathsf{Elem}\ [\![e]\!]_X\ [\![t']\!]_G}$$

$$\text{Refine}\ \frac{d \in_u e \Rightarrow v :: [\![e]\!]_G \qquad e <: e' \overset{\text{ref}}{\Rightarrow} f :: \forall \mathsf{a}\,.\,[\![e]\!]_G\ \mathsf{a} \to [\![e']\!]_G\ \mathsf{a}}{d \in_u e' \Rightarrow f\ v :: [\![e']\!]_G}$$

$$\text{Mix(1)}\ \frac{d \in_{\mathbf{mix}} g \Rightarrow v :: [\![g]\!]_G\ \mathsf{String} \qquad c \overset{\text{str}}{\Rightarrow} v' :: \mathsf{String}}{d\mathbf{,}c \in_{\mathbf{elem}} \mathbf{mix}(g) \Rightarrow \mathit{Mix}\ v\ v' :: \mathsf{Mix}\ [\![g]\!]_G\ ()}$$

$$\text{Mix(2)}\ \frac{c \overset{\text{str}}{\Rightarrow} v :: \mathsf{String} \qquad e[d] \in_{\mathbf{elem}} g \Rightarrow \mathit{Elem}\ ()\ v' :: \mathsf{Elem}\ [\![e]\!]_X\ ()}{c\mathbf{,}e[d] \in_{\mathbf{mix}} g \Rightarrow \mathit{Elem}\ v\ v' :: \mathsf{Elem}\ [\![e]\!]_X\ [\![g]\!]_G\ \mathsf{String}}$$

$$\text{Rep(0)}\ \frac{}{\epsilon \in_{g,u} \{0,0\} \overset{\text{rep}}{\Rightarrow} \mathit{ZZ} :: \mathsf{ZZ}\ [\![g]\!]_G\ [\![u]\!]_m}$$

$$\text{Rep(1)}\ \frac{\epsilon_u \in g\{0,m\} \Rightarrow v :: [\![0,m]\!]_B\ [\![g]\!]_G\ [\![u]\!]_m}{\epsilon \in_{g,u} \{0,m+1\} \overset{\text{rep}}{\Rightarrow} \mathit{ZS}\ \mathit{Nothing}\ v :: \mathsf{ZS}\ [\![0,m]\!]_B\ [\![g]\!]_G\ [\![u]\!]_m}$$

$$\text{Rep(1')}\ \frac{}{\epsilon \in_{g,u} \{0,\infty\} \overset{\text{rep}}{\Rightarrow} \mathit{ZI}\ [\,] :: \mathsf{ZI}\ [\![g]\!]_G\ [\![u]\!]_m}$$

$$\text{Rep(2)}\ \frac{d_1 \in_u g \Rightarrow v_1 :: [\![g]\!]_G\ [\![u]\!]_m \qquad d_2 \in_u g\{m,n\} \overset{\text{rep}}{\Rightarrow} v_2 :: [\![m,n]\!]_B\ [\![g]\!]_G\ [\![u]\!]_m}{d_1\mathbf{,}\ d_2 \in_{g,u} \{m+1,n+1\} \overset{\text{rep}}{\Rightarrow} \mathit{SS}\ v_1\ v_2 :: \mathsf{SS}\ [\![m,n]\!]_B\ [\![g]\!]_G\ [\![u]\!]_m}$$

$$\text{Rep(3)}\ \frac{d_1 \in_u g \Rightarrow v_1 :: [\![g]\!]_G\ [\![u]\!]_m \qquad d_2 \in_u g\{0,m\} \Rightarrow v_2 :: [\![0,m]\!]_B\ [\![g]\!]_G\ [\![u]\!]_m}{d_1\mathbf{,}\ d_2 \in_{g,u} \{0,m+1\} \overset{\text{rep}}{\Rightarrow} \mathit{ZS}\ (\mathit{Just}\ v_1)\ v_2 :: \mathsf{ZS}\ [\![0,m]\!]_B\ [\![g]\!]_G\ [\![u]\!]_m}$$

$$\text{Rep(3')}\ \frac{d_1 \in_u g \Rightarrow v_1 :: [\![g]\!]_G\ [\![u]\!]_m \qquad d_2 \in_{g,u} \{0,\infty\} \overset{\text{rep}}{\Rightarrow} \mathit{ZI}\ v_2 :: [\![0,\infty]\!]_B\ [\![g]\!]_G\ [\![u]\!]_m}{d_1\mathbf{,}\ d_2 \in_{g,u} \{0,\infty\} \overset{\text{rep}}{\Rightarrow} \mathit{ZI}\ (v_1 : v_2) :: \mathsf{ZI}\ [\![0,\infty]\!]_B\ [\![g]\!]_G\ [\![u]\!]_m}$$

$$\text{Reflex}\ \frac{}{g <: g \Rightarrow \mathit{id} :: \forall \mathsf{a}\,.\,[\![g]\!]_G\ \mathsf{a} \to [\![g]\!]_G\ \mathsf{a}}$$

$$\text{Trans}\ \frac{g <: g' \Rightarrow f :: \forall \mathsf{a}\,.\,[\![g]\!]_G\ \mathsf{a} \to [\![g']\!]_G\ \mathsf{a} \qquad g' <: g'' \Rightarrow f' :: \forall \mathsf{a}\,.\,[\![g']\!]_G\ \mathsf{a} \to [\![g'']\!]_G\ \mathsf{a}}{g <: g'' \Rightarrow f'\,.\,f :: \forall \mathsf{a}\,.\,[\![g]\!]_G\ \mathsf{a} \to [\![g'']\!]_G\ \mathsf{a}}$$

$$\text{Res}\ \frac{x <^r_\Sigma x'}{x <: x' \Rightarrow \texttt{"LE\_"}x :: \forall \mathsf{a}\,.\,\texttt{"LE\_"}x\ \mathsf{a} \to \texttt{"LE\_"}x'\ \mathsf{a}}$$

$$\text{Ext}\ \frac{x <^e_\Sigma x'}{x <: x' \Rightarrow \texttt{"LE\_"}x :: \forall \mathsf{a}\,.\,\texttt{"LE\_"}x\ \mathsf{a} \to \texttt{"LE\_"}x'\ \mathsf{a}}$$

End of proof.

The next proposition says that if a document d appears as the content of an element and validates against two types t_1 and t_2 such that $t_1 <: t_2$, then there exists a suitable function which coerces the t_1-translation $[\![d]\!]^{t_1,u}_V$ into a t_2-translation $[\![d]\!]^{t_2,u}_V$. (The restriction that d must appear as the content of an element arises because this is the only place we can apply the subsumption rule.) For example, in an element content context, if d validates against *publication* as v then it validates against *document* as $\mathit{LE\_T\_publication}\ v$.

Proposition 2 (Existence of coercions) *For all elements e of a schema, if $e[d] \in_u e[t_1] \ \wedge\ e[d] \in_u e[t_2] \ \wedge\ t_1 <: t_2$ then there exists a function $f :: \forall \mathsf{a}\,.\,[\![t_1]\!]_X\ \mathsf{a} \to [\![t_2]\!]_X\ \mathsf{a}$ satisfying $f\ [\![d]\!]^{t_1,u}_V = [\![d]\!]^{t_2,u}_V$.*

Proof: Recall that $<:$ is generated as the reflexive-transitive closure of $<^e_\Sigma \cup <^r_\Sigma$. It is easy to see from the value translation rules for refinement that this implies the witness f is either the identity or a composition of axiomatic subtyping witnesses.

If $t_1 = t_2$, then $f = id$ and we are done.

Otherwise, $\exists t_3.(t_1 <^r_\Sigma t_3 \vee t_1 <^e_\Sigma t_3) \wedge t_3 <: t_2$. So by induction there is an $f' :: \forall \mathtt{a} \,.\, [\![t_3]\!]_X\ \mathtt{a} \to [\![t_2]\!]_X\ \mathtt{a}$, and $f = f' \,.\, \mathtt{"LE\_"}t_1$.

It remains to show that $f\ [\![d]\!]_V^{t_1,u} = [\![d]\!]_V^{t_2,u}$. Observe that, by expanding our alternate notation, the Elem rule can be rewritten:

$$\text{Elem}\frac{[\![d]\!]_V^{t,u} = v \qquad [\![t <: t']\!] = f}{[\![e[d]]\!]_V^{e[t'],u} = \mathit{Elem}\ ()\ (f\ v)}$$

Let $t' := t_2$. Taking $t := t_1$, we obtain $[\![e[d]]\!]_V^{e[t_2],u} = \mathit{Elem}\ ()\ (f\ [\![d]\!]_V^{t_1,u})$ and again, taking $t := t_2$, $[\![e[d]]\!]_V^{e[t_2],u} = \mathit{Elem}\ ()\ (\mathit{id}\ [\![d]\!]_V^{t_2,u}) = \mathit{Elem}\ ()\ [\![d]\!]_V^{t_2,u}$. Therefore, by congruence of equality, $f\ [\![d]\!]_V^{t_1,u} = [\![d]\!]_V^{t_2,u}$.

End of proof.

7 Conclusions

We have achieved two things in this paper. Firstly, we argued that generic programming is appropriate for XML processing, and, as evidence of this claim, we described the generic implementation of an XML compressor. Our compressor, XCompress, compares favourably with XMill, because it uses information about an XML document present in its DTD. Our second contribution was to present an encoding of Schema in terms of Haskell data types and describe a generic program which parses and validates XML documents with respect to their purported Schema.

Several other classes of XML tools can be implemented as generic programs and would benefit from such an implementation [21]. The combination of HaXml and generic programming in Generic Haskell is very useful for implementing the kind of XML tools for which DTDs play an important rôle. Using generic programming, such tools become easier to write, because a lot of the code pertaining to DTD handling and optimisation is generated by the Generic Haskell compiler. The resulting tools are more effective, because they can take advantage of the DTD's structure. For example, a DTD-aware XML compressor, such as XCompress described in this paper, compresses better than XML compressors that don't take the DTD into account, such as XMill. Furthermore, our compressor is much smaller than XMill. Now that we have a translation of Schema into Haskell, we can continue the development of XML tools for the more expressive Schema formalism.

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 [23] 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 [8] for generic programming on DTDs or Schemas is a promising approach.

Acknowledgements. Andres Löh, Paul Hagg, and Jeroen Snijders helped with implementing XCompress.

References

[1] A.I. Baars, A. Löh, and S.D. Swierstra. Parsing permutation phrases. In R. Hinze, editor, *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 171–182. Elsevier, 2001.

[2] 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.

[3] Richard Bird and Jeremy Gibbons. Arithmetic coding with folds and unfolds. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming, 4th International Summer School, Oxford, UK*, volume 2638 of *LNCS*. Springer-Verlag, 2003.

[4] Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler. MSL: A model for W3C XML Schema. In *Proc. WWW10*, May 2001.

[5] Robert D. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, 1988.

[6] Mario Cannataro, Gianluca Carelli, Andrea Pugliese, and Domenico Sacca. Semantic lossy compression of XML data. In *Knowledge Representation Meets Databases*, 2001.

[7] James Cheney. Compressing XML with multiplexed hierarchical models. In *Proceedings of the 2001 IEEE Data Compression Conference, DCC'01*, pages 163–172, 2001.

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

[9] 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/`.

[10] Dave Clarke and Andres Löh. Generic Haskell, specifically. In Jeremy Gibbons and Johan Jeuring, editors, *Generic Programming*, volume 243 of *IFIP*, pages 21–48. Kluwer Academic Publishers, January 2003.

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

[12] Jorge Coelho and Mário Florido. Type-based XML processing in logic programming. In *PADL 2003*, pages 273–285, 2003.

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

[14] William S. Evans and Christopher W. Fraser. Bytecode compression via profiled grammar rewriting. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 148–155, 2001.

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

[16] Michael Franz. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile object systems. In *Mobile Object Systems: Towards the Programmable Internet*, pages 263–276. Springer-Verlag: Heidelberg, Germany, 1997.

[17] Vladimir Gapeyev and Benjamin C. Pierce. Regular object types. In *European Conference on Object-oriented Programming (ECOOP 2003)*, July 2003. to appear.

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

[19] 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.

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

[21] Paul Hagg. A framework for developing generic XML Tools. Master's thesis, Department of Information and Computing Sciences, Utrecht University, 2002.

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

[23] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *Proceedings of the 6th Mathematics of Program Construction Conference, MPC'02*, volume 2386 of *LNCS*, pages 148–174, 2002.

[24] 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.

[25] Graham Hutton and Erik Meijer. Monadic parser combinators. *Journal of Functional Programming*, 8(4):437–444, 1996.

[26] INC Intelligent Compression Technologies. XML-Xpress. Whitepaper available from `http://www.ictcompress.com/products_xmlxpress.html`, 2001.

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

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

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

[30] Oleg Kiselyov and Shriram Krishnamurti. SXSLT: manipulation language for XML. In *PADL 2003*, pages 226–272, 2003.

[31] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Utrecht University, 2001.

[32] 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.

[33] Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In *Proceedings of the International Conference on Functional Programming (ICFP'03)*, August 2003. to appear.

[34] Brett McLaughlin. *Java & XML data binding.* O'Reilly, 2003.

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

[36] Eldon Metz and Allen Brookes. XML data binding. *Dr. Dobb's Journal*, pages 26–36, 2003.

[37] OASIS. RELAX NG. Available from `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`, 2001.

[38] 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/`, February 1999.

[39] 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/`.

[40] Jérôme Siméon and Philip Wadler. The essence of XML. In *Proc. POPL 2003*, 2003.

[41] C.H. Stork, V. V. Haldar, and M. Franz. Generic adaptive syntax-directed compression for mobile code. Technical Report 00-42, Department of Information and Computer Science, University of California, Irvine, 2000.

[42] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and Joao Sariava. Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206, 1998.

[43] Peter Thiemann. A typed representation for HTML and XML documents in haskell. Available from `http://citeseer.nj.nec.com/thiemann01typed.html`, 2001.

[44] Pankaj Tolani and Jayant R. Haritsa. XGRIND: A query-friendly XML compressor. In *ICDE*, 2002.

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

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

[47] W3C. XML Schema: Formal description. Available from `http://www.w3.org/TR/xmlschema-formal`, 2001.

[48] W3C. XML Schema part 0: Primer. Available from `http://www.w3.org/TR/xmlschema-0`, 2001.

[49] W3C. XML Schema part 1: Structures. Available from `http://www.w3.org/TR/xmlschema-1`, 2001.

[50] W3C. XML Schema part 2: Datatypes. Available from `http://www.w3.org/TR/xmlschema-2`, 2001.

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

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