UUXML: A Type-Preserving XML Schema–Haskell Data Binding

Frank Atanassow, Dave Clarke* and Johan Jeuring

Institute of Information & Computing Sciences
Utrecht University
The Netherlands
{franka,dave,johanj}@cs.uu.nl

Abstract. An XML data binding is a translation of XML documents into values of some programming language. This paper discusses a type-preserving XML–Haskell data binding that handles documents typed by the W3C XML Schema standard. Our translation is based on a formal semantics of Schema, and has been proved sound with respect to the semantics. We also show a program in Generic Haskell that constructs parsers specialized to a particular Schema type.

1 Introduction

XML [23] 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 type declaration such as a schema. A number of alternative methods of processing XML documents are available:

– **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 [24], XDuce [12], Yatl [5], XMλ [17, 20], SXSLT [14], XStatic [8] etc. can be used to transform 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 [18].

Using a specialized programming language or a data binding has significant advantages over the SAX or DOM approach. For example, parsing comes for free and can be optimized for a specific schema. Also, it is easier to implement, test and maintain software in the target language. A data binding has the further advantages that existing programming language technology can be leveraged, and that a programmer need account for XML idiosyncracies (though this may be a disadvantage for some applications). Programming languages for which

* Now at: CWI, Amsterdam, Netherlands, dave@cwi.nl

XML data bindings have been developed include Java [16] and Python, as well as declarative programming languages such as Prolog [6] and Haskell [22, 29]. Using Haskell as the target for an XML data binding offers the advantages of a typed higher-order programming language with a powerful type system.

Since W3C released XML, thousands of XML tools have been developed, including editors, databases, converters, parsers, validators, search engines, encryptors and compressors [7, 9, 10]. Many XML applications depend on a schema; we call such tools *schema-aware XML tools* [29]. Examples are a validator, which checks that an XML document exhibits the type structure described by a schema, and an XML editor that suggests admissible elements or attributes at the cursor position. Similarly, the performance of search algorithms and compressors improves when the structure of the document is known in advance. Another feature shared by such programs is that they do essentially the same thing for different schemas. In this sense these tools are very similar to generic algorithms such as the equality function, and the map, fold and zip functions. We claim that many XML tools are generic programs, or would benefit from being viewed as such.

In this paper we present UUXML, a translation of XML documents into Haskell, and more specifically a translation tailored to permit writing programs in Generic Haskell [11], a superset of Haskell that allows to define such generic programs. The documents conform to the type system described in the W3C XML Schema [26–28] standard, and the translation preserves typing in a sense we formalize by a type soundness theorem. More details of the translation and a proof of the soundness result are available in a technical report [1].

This paper is organised as follows. Section 2 describes a tool for translating an XML Schema to a set of Haskell data types. Section 3 describes a parser, implemented as a Generic Haskell program, for parsing an XML document into a Haskell value. Section 4 summarizes and discusses related and future work.

2 From Schema to Haskell

XML was introduced with a type formalism called Document Type Declarations (DTDs). Though XML has achieved widespread popularity, DTDs themselves have been deemed 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 [19], and the W3C's own XML Schema Recommendation [26–28]. Both systems include a set of primitive datatypes such as numbers and dates, a way of 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). To write Haskell programs over documents conforming to schemas we require a translation of schemas to Haskell analagous to the HaXml translation of DTDs to Haskell.

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 syntax used here are 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 [25]. 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.

2.1 An overview of XML Schema

A schema describes a set of type declarations which may not only constrain the form of, but also 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. For instance, the following declarations 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.*) A document which validates against `doc` is:

```
<doc>
  <author>James Joyce</author>
  <title>Ulysses</title>
  <year>1922</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:

$$\textbf{def}\ \mathsf{doc}[\,\mathit{document}\,]\,;\quad \textbf{def}\ \mathit{document} = \mathsf{author}^{*}\,\texttt{,}\ \mathsf{title}\,\texttt{,}\ \mathsf{year}?\,;$$

and the example document above is written:

$$\mathsf{doc}[\,\mathsf{author}[\,\texttt{"James Joyce"}\,]\texttt{,}\mathsf{title}[\,\texttt{"Ulysses"}\,]\texttt{,}\mathsf{year}[\,\texttt{"1922"}\,]\,]$$

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

1. Schema defines more 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 of 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 refer to a model group as a *type*. An overview of the syntax of groups is given by the grammar g.

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

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

This grammar is only a rough approximation of 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 forms are familiar from DTDs and regular expressions. Forms $@a$, `e` and t are variables referencing, respectively, attributes, elements and types in the schema. We consider the remaining features in turn.

Primitives. Schema defines some familiar primitives types such as *string*, *boolean* and *integer*, but also more exotic ones (which we do not treat here) such as *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 interspersing elements with text. More precisely, a document d matches $\mathbf{mix}(g)$ if $\mathit{unmix}(d)$ matches g, where $\mathit{unmix}(d)$ is obtained from d by deleting all character text at the top level. An example of mixed content is an XHTML paragraph element with emphasized phrases; in MSL its content would be declared as $\mathbf{mix}(\mathsf{em}^*)$. The opposite of ‘mixed content’ is ‘element-only content.’

DTDs support a similar, but subtly different, notion of mixed content, 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). Groups involving `#PCDATA` can only appear in two forms, either by itself, or in a repeated disjunction involving only element names:

$$\texttt{(\#PCDATA | } \mathsf{e}_1 \texttt{ | } \mathsf{e}_2 \texttt{ | } \cdots\ \mathsf{e}_n \texttt{)* .}$$

To see how Schema’s notion of mixed content differs from DTDs’, observe that a reasonable translation of the DTD content type above is $[\,\mathsf{String}\ \texttt{:+:}\ [\![\texttt{em}]\!]_G\,]$ where $[\![\texttt{em}]\!]_G$ is the translation of `em`. This might lead one to think that we can translate a schema type such as $\mathbf{mix}(g)$ similarly as $[\,\mathsf{String}\ \texttt{:+:}\ [\![g]\!]_G\,]$. However, this translation would not respect the semantics of MSL for at least two reasons. First, it is too generous, because it allows repeated occurrences, yet:

$$\texttt{"hello"}\texttt{,}\ \mathsf{e}[]\texttt{,}\ \texttt{"world"} \in \mathbf{mix}(\mathsf{e}) \quad \text{but} \quad \texttt{"hello"}\texttt{,}\ \mathsf{e}[]\texttt{,}\ \mathsf{e}[]\texttt{,}\ \texttt{"world"} \notin \mathbf{mix}(\mathsf{e})\,.$$

Second, it cannot account for more complex types such as $\mathbf{mix}(\mathsf{e}_1\texttt{,}\ \mathsf{e}_2)$. A document matching the latter 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

$$\mathbf{def}\ \mathit{module} = \mathsf{header}\texttt{,}\ \mathsf{imports}\texttt{,}\ \mathsf{fixityDecl}^*\texttt{,}\ \mathsf{valueDecl}^*\,;$$

with comments: $\mathbf{mix}(\mathit{module})$. An analogous model group is not expressible in the DTD formalism.

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

$$\mathbf{def}\ \mathit{email} = (\mathsf{subject}\ \texttt{\&}\ \mathsf{from}\ \texttt{\&}\ \mathsf{to})\ \texttt{,}\ \mathsf{body}\,;$$

Although interleaving does not really increase the expressiveness of Schema over DTDs, they are a welcome convenience. Interleavings can be expanded to a choice of sequences, but these rapidly become unwieldy. For example, $[\![a\ \&\ b]\!] = a\ ,\ b\ |\ b\ ,\ a$ but $[\![a\ \&\ b\ \&\ c]\!] = a\ ,\ (b\ ,\ c\ |\ c\ ,\ b)\ |\ b\ ,\ (a\ ,\ c\ |\ c\ ,\ a)\ |\ c\ ,\ (a\ ,\ b\ |\ b\ ,\ a)$. (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, which is multiplicative in character. As an example of extension, we declare a type *publication* obtained from *document* by adding fields at the end:

$$\textbf{def}\ \textit{publication}\ \textbf{extends}\ \textit{document} = \mathsf{journal}\ |\ \mathsf{publisher}\,;$$

A *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:

$$\textbf{def}\ t_1 = @a_1\ ,\ \mathsf{e}_1\,; \quad \textbf{def}\ t_2\ \textbf{extends}\ t_1 = @a_2\ ,\ \mathsf{e}_2\,; \tag{1}$$

then the content of t_2 is $(@a_1\ \&\ @a_2)\ ,\ \mathsf{e}_1\ ,\ \mathsf{e}_2$.

To illustrate restriction, we declare a type *article* obtained from *publication* by fixing some of the variability. If an *article* is always from a journal, we write:

$$\textbf{def}\ \textit{article}\ \textbf{restricts}\ \textit{publication} = \mathsf{author}^*\ ,\ \mathsf{title}\ ,\ \mathsf{year}\ ,\ \mathsf{journal}\,;$$

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.

2.2 An overview of the translation

The objective of the translation is to be able to write Haskell programs on data corresponding to schema-conforming documents. At minimum, 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.

Theorem 1. *Let $[\![-]\!]_G$ and $[\![-]\!]_V^{g;u}$ be respectively the type and value translations generated by a schema. Then, for all documents d, groups g and mixities u, if d validates against g in mixity context u, then $[\![d]\!]_V^{g;u} :: [\![g]\!]_G\ [\![u]\!]_{mix}$.*

Let us outline the difficulties posed by features of Schema. As a starting point, consider how we might translate regular patterns into Haskell.

$$\begin{aligned}
[\![\epsilon]\!]_G &= () & [\![\emptyset]\!]_G &= \mathsf{Void} \\
[\![g_1\,,\,g_2]\!]_G &= ([\![g_1]\!]_G, [\![g_2]\!]_G) & [\![g_1 \mid g_2]\!]_G &= \mathsf{Either}\ [\![g_1]\!]_G [\![g_2]\!]_G \\
[\![g^*]\!]_G &= [\,[\![g_1]\!]_G\,] & [\![g^+]\!]_G &= ([\![g]\!]_G, [\![g^*]\!]_G) \\
[\![g?]\!]_G &= \mathsf{Maybe}\ [\![g]\!]_G
\end{aligned}$$

This is the sort of translation employed by HaXml [29], and indeed we follow the same tack. In contrast, WASH [22] 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.

As an example for the reader to refer back to, we present (part of) the translation of the *document* type:

```
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_ indicates that this declaration refers to the type *document*, rather than an element (or attribute) of the same name, which would be indicated by a prefix E_ (A_, respectively). We explain the remaining features in turn.

Primitives. Primitives are translated to the corresponding Haskell types, wrapped by a constructor. For example (the argument u relates to mixed content, discussed below):

```
data T_string u   =   T_string String .
```

User-defined types. Types are translated along the lines of HaXml, using products to model sequences and sums to model choices.

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

The translation takes each group to a Haskell type of kind $\star \to \star$:

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

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, u is restricted to be either String or $()$. For example, $e[t]$ translates 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 Elem:

$$\mathbf{data}\ \mathsf{Elem\ e\ g\ u} = \mathit{Elem}\ \mathsf{u}\ (\mathsf{g}\ ())$$

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' [15], 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.

$$\mathbf{data}\ \mathsf{Mix\ g\ u} = \mathit{Mix}\ (\mathsf{g\ String})\ \mathsf{String}$$

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.

Interleaving. Interleaving is modeled in 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

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

schedule where it is known that exactly three speakers of different types will appear. A part of such a schema may look like:

```
def schedule[speaker & invitedSpeaker & keynoteSpeaker]; .
```

A schema processor must know the order in which speakers appeared, but since we do not record the permutation we cannot recover the document ordering. More commonly, since attribute groups are modeled as interleavings of attributes, this means in particular that schema processors using our translation cannot know 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:

```
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) .
```

The names of datatypes modeling bounds are meant to suggest the familiar unary encoding of naturals, 'Z' for zero and 'S' for successor, while 'I' stands for 'infinity'. Some sample translations are:

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

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 type classes here, though, because one objective of our data representation is to support writing schema-aware programs in Generic Haskell. Such programs operate by recursing over the structure of a type, so encoding the subtyping relation in a non-structural manner such as *via* the type class relation would be counterproductive.

The type **anyType** behaves as the *union* of all types, which suggests an implementation in terms of Haskell datatypes: encode **anyType** as a datatype with one constructor for each type that directly restricts it, the direct subtypes, and one for values that are 'exactly' of type **anyType**.

In the case of our bibliographical example, we have:

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

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 variables as references to an LE_ type. 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 t_1 and t_2 are defined as in (1) then $\mathit{split}(t_1) = (@a_1, \mathsf{e}_1)$ and, if t'_2 is the 'extended part' of t_2, then $\mathit{split}(t'_2) = (@a_2, \mathsf{e}_2)$. We then define the translation of t_2 to be:

$$\mathit{fst}(\mathit{split}(t_1)) \;\&\; \mathit{fst}(\mathit{split}(t'_2))\,,\; (\mathit{snd}(\mathit{split}(t_1))\;\;,\;\mathit{snd}(\mathit{split}(t'_2)))\;.$$

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). So 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.

3 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, written in Generic Haskell. To abstract away from details of XML concrete syntax, rather than parse strings, we use a universal data representation *Document* which presents a document as a tree (or rather a forest):

```
type Doc      =  [DocItem]
data DocItem  =  DText String | DAttr String Doc | DElem String Doc
```

We use standard techniques [13] to define a set of monadic parsing combinators operating over Doc. 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. The type of generic parsers is the kind-indexed type GParse{[κ]} t and *gParse*{|t|} denotes a parser which tries to read a document into a value of type t. We now describe its behavior on the various components of Schema.

```
type GParse{[⋆]} t  =  P t
gParse{|t :: κ|}    ::  GParse{[κ]} t
gParse{|String|}    =  pMixed
gParse{|Unit|}      =  pElementOnly
```

The first two cases handle mixities: *pMixed* optionally matches *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{|Rep g b u|}  =  fmap Rep gParse{|b g u|}
gParse{|ZZ g u|}     =  return ZZ
gParse{|ZI g u|}     =  fmap ZI $ many gParse{|g u|}
gParse{|ZS g b u|}   =  do x ← option gParse{|g u|}
                           y ← gParse{|b g u|}
                           return (ZS x (Rep y))
gParse{|SS g b u|}   =  do x ← gParse{|g u|}
                           y ← gParse{|b g u|}
                           return (SS x (Rep y))
```

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

$$\begin{aligned} gParse\{\!|\mathsf{T\_string}|\!\} &= fmap\ T\_string\ pText \\ gParse\{\!|\mathsf{T\_integer}|\!\} &= fmap\ T\_integer\ pReadableText \end{aligned}$$

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

$$\begin{aligned} gParse\{\!|\mathsf{Elem\ e\ g\ u}|\!\} = \ &\mathbf{do}\ mixity \leftarrow gParse\{\!|\mathsf{u}|\!\} \\ &\quad \mathbf{let}\ p = gParse\{\!|\mathsf{g}|\!\}\ pElementOnly \\ &\quad elemt\ gName\{\!|\mathsf{e}|\!\}\ (fmap\ (Elem\ mixity)\ p) \end{aligned}$$

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 *DElem 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 similar.

This code uses an auxiliary type-indexed function $gName\{\!|\mathsf{e}|\!\}$ to acquire the name of an element; 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 (say, LE_T_), thus giving the attribute or element name as a string.

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

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.

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

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

Interleaving is handled using these permutation phrase combinators [3]:

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

Briefly, a permutation parser $q :: \mathsf{PermP}\ \mathsf{a}$ 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:

$$\begin{array}{lll} f1 = gInter\{\!|\mathsf{g1\ u}|\!\} & :: & \forall \mathsf{g1\ u\ b}\,.\,\mathsf{PermP\ (g1\ u} \rightarrow \mathsf{b)} \rightarrow \mathsf{PermP\ b} \\ f2 = gInter\{\!|\mathsf{g2\ u}|\!\} & :: & \forall \mathsf{g2\ u\ c}\,.\,\mathsf{PermP\ (g2\ u} \rightarrow \mathsf{c)} \rightarrow \mathsf{PermP\ c} \qquad \text{-- hence} \\ f2\ .\ f1 & :: & \forall \mathsf{g1\ g2\ u\ c}\,.\,\mathsf{PermP\ (g1\ u} \rightarrow \mathsf{g2\ u} \rightarrow \mathsf{c)} \rightarrow \mathsf{PermP\ c}\ . \end{array}$$

Note that if c is instantiated to $\mathsf{Inter\ g1\ g2\ u}$, 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.

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

In 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 $\mathsf{ZS\ ZZ}$: this corresponds to a Schema type like $\mathsf{e}\{0, 1\}$, that is, an optional element (or attribute).[2]

4 Conclusions

XML Schema has several features not available natively in Haskell, including mixed content, two forms of subtyping and a generalized form of repetition. Nevertheless, we have shown that these features can be accomodated by Haskell's datatype mechanism alone. The existence of a simple formal semantics

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

for Schema such as MSL's was a great help to both the design and implementation of our work, and essential for the proof of type soundness.

Though the translation is cumbersome for Haskell programs which process documents of a single schema, for schema-aware programs such as the parser of Section 3 this defect is not so noticeable because Generic Haskell programs usually do not need to pattern-match deeply into values of a datatype. In a companion paper [2] we show how to use Generic Haskell to automatically infer type isomorphisms to effectively customize our translation and make writing non-schema-aware XML software far simpler.

Besides its verbosity, 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. This has a negative impact on the efficiency of generic applications such as our parser. Another issue is the use of unary encoding in repetition bounds, though this could be addressed by using a larger radix. Finally, schema types, which obey equational laws, are always translated as abstract datatypes, which satisfy analagous laws only up to isomorphism; this lack of coherence means that users must know some operational details of our translator. Our work on isomorphism inference can help address this problem.

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 presented here and a generic pretty-printer. 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 [21], which more accurately models Schema's named typing, and exploiting the 1-unambiguity constraint to obtain a more economical translation.

References

1. Frank Atanassow, Dave Clarke, and Johan Jeuring. Scripting XML with Generic Haskell. Technical Report UU-CS-2003, Utrecht University, 2003.
2. Frank Atanassow and Johan Jeuring. Inferring type isomorphisms generically. To appear in Proc. MPC '04.
3. 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.
4. Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler. MSL: A model for W3C XML Schema. In *Proc. WWW10*, May 2001.
5. Sophie Cluet and Jérôme Siméon. YATL: a functional and declarative language for XML, 2000.
6. Jorge Coelho and Mário Florido. Type-based XML processing in logic programming. In *PADL 2003*, pages 273–285, 2003.
7. Peter Flynn. *Understanding SGML and XML Tools.* Kluwer Academic Publishers, 1998.
8. Vladimir Gapeyev and Benjamin C. Pierce. Regular object types. In *European Conference on Object-oriented Programming (ECOOP 2003)*, 2003.

9. Lars M. Garshol. Free XML tools and software. Available from `http://www.garshol.priv.no/download/xmltools/`.
10. Google. Web Directory on XML tools. `http://www.google.com/`.
11. Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory, 2003. To appear.
12. Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Third International Workshop on the Web and Databases (WebDB), volume 1997 of Lecture Notes in Computer Science*, pages 226–244, 2000.
13. Graham Hutton and Erik Meijer. Monadic parser combinators. *Journal of Functional Programming*, 8(4):437–444, 1996.
14. Oleg Kiselyov and Shriram Krishnamurti. SXSLT: manipulation language for XML. In *PADL 2003*, pages 226–272, 2003.
15. Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Second USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, October 1999. USENIX Association. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).
16. Brett McLaughlin. *Java & XML data binding*. O'Reilly, 2003.
17. Erik Meijer and Mark Shields. XMLambda: A functional language for constructing and manipulating XML documents. Available from `http://www.cse.ogi.edu/~mbs/`, 1999.
18. Eldon Metz and Allen Brookes. XML data binding. *Dr. Dobb's Journal*, pages 26–36, March 2003.
19. OASIS. RELAX NG. `http://www.relaxng.org`, 2001.
20. 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/`.
21. Jérôme Siméon and Philip Wadler. The essence of XML. In *Proc. POPL 2003*, 2003.
22. Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4&5):435–468, July 2002.
23. W3C. XML 1.0. `http://www.w3.org/XML/`, 1998.
24. W3C. XSL Transformations 1.0. `http://www.w3.org/TR/xslt`, 1999.
25. W3C. XML Schema: Formal description. `http://www.w3.org/TR/xmlschema-formal`, 2001.
26. W3C. XML Schema part 0: Primer. `http://www.w3.org/TR/xmlschema-0`, 2001.
27. W3C. XML Schema part 1: Structures. `http://www.w3.org/TR/xmlschema-1`, 2001.
28. W3C. XML Schema part 2: Datatypes. `http://www.w3.org/TR/xmlschema-2`, 2001.
29. Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *International Conference on Functional Programming*, pages 148–159, 1999.