Customizing an XML–Haskell data binding with type isomorphism inference in Generic Haskell

Frank Atanassow

Johan Jeuring

Department of Information and Computing Sciences,
Utrecht University

Technical Report UU-CS-2006-025

www.cs.uu.nl

ISSN: 0924-3275

Customizing an XML–Haskell data binding with type isomorphism inference in Generic Haskell

Frank Atanassow and Johan Jeuring

CWI & Utrecht University

Abstract. This paper introduces a type-preserving XML Schema–Haskell data binding (or, translation) UUXML, and shows how to customize it by exploiting the theory of canonical isomorphisms to automatically infer coercions between the machine-generated types and an equivalent, more natural, user-defined set of types. We show how to implement the inference mechanism in Generic Haskell.

1 Introduction

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 UUXML that handles documents typed by the W3C XML Schema standard. Our translation is based on a formal semantics of Schema, and has been proven sound with respect to the semantics. We also show programs in Generic Haskell that construct parsers and printers specialized to a particular Schema type.

Although the translation produces complex datatypes, for generic, or 'schema-aware', applications such as compression, parsing, and unparsing, the translation is adequate because such programs do not depend explicitly on the details of any particular type. But for more conventional uses that depend on a particular schema, the translation is unwieldy and unnatural. To address this, we introduce a form of data binding customization based on the theory of canonical isomorphisms.

Datatypes which differ inessentially in their names and structure are called isomorphic. Between certain isomorphic pairs of types there exists a unique invertible coercion. In this article we describe and implement a program in Generic Haskell which automatically infers this coercion by normalizing types with respect to an algebraic theory. A simple generalization of this technique also enables us to infer some noninvertible coercions such as projections, injections and *ad hoc* coercions between base types.

We explain how this technique has been used to drastically improve the usability of UUXML, and suggest how it might be applied to improve other type-safe language embeddings.

1.1 Outline

This paper is an extended version of our paper UUXML: a type-preserving XML Schema–Haskell data binding [3] presented at the Practical Aspects of Declarative Languages Conference in 2004, and our paper about inferring type isomorphisms generically [4] presented at the Mathematics of Program Construction Conference in 2004.

The remainder of this article is organized as follows. Section 2 introduces UUXML, a type-preserving XML Schema–Haskell data binding. Section 3 gives a formal account of the translation implemented by UUXML. Section 4 introduces generic programming in Generic Haskell, and describes two 'schema-aware' XML applications, parsing and printing, implemented generically. Section 5 introduces the notion of binding customization, discusses the need for such customization *vis-à-vis* UUXML, and shows how to use iso inference to automatically customize the UUXML data binding. Section 6 talks briefly about the theory of isomorphisms, and describes informally the user interface of our inference mechanism. Section 7 shows how our iso inferencer is implemented in Generic Haskell. Finally, Section 8 summarizes our results, and discusses related work and possibilities for future work in this area.

2 An XML Schema–Haskell data binding

XML [45] is the core technology of modern data exchange. An XML document is essentially a tree-based data structure, often, 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 APIs**. 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 XSLT [46], XQuery [52], XDuce [18], CDuce [7], XMλ [29, 40], XStatic [14] *etc.* can be used to transform XML documents.
– **XML data bindings**. XML values can be 'embedded' in an existing programming language (the "target") by finding a suitable translation between XML types and types of the programming language [30]. Examples include JAXB [42] for Java, WASH [44] and HaXml [53] for Haskell, various tools [33] such as EaseXML [34] for Python and X-Prolog [11] for Prolog.

Using a specialized programming language or a data binding has significant advantages over the SAX or DOM approach. For example, parsing to an abstract syntax tree 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 since application logic does not have to be mixed with calls to the parsing APIs. A data binding has the further advantages that existing programming language technology can be leveraged, and that a programmer need not account for XML idiosyncracies (though this may be a disadvantage for some applications). Using Haskell as the target offers the advantages of a typed higher-order programming language with a powerful type system.

In this section we present UUXML, a type-preserving translation of XML documents into Haskell, and more specifically a translation tailored to permit writing programs in Generic Haskell. The documents are assumed to conform to the type system described in the W3C XML Schema [47–49] 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].

2.1 From XML 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 [32], and the W3C's own XML Schema Recommendation [47–49]. 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 analogous to the HaXml translation of DTDs to Haskell [53].

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 datatypes, 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) [9]; that treatment also forms the basis of the W3C's own, more ambitious formal semantics [51].[1] We do not treat all features of Schema, but only the subset covered by MSL (except wildcards). This portion, however, arguably forms a representative subset and suffices for many Schema applications.

2.2 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 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.

[1] At the time of writing, this is currently a W3C Candidate Recommendation.

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 long-winded and irregular. We use an alternative syntax close to that of MSL [9], which is more orthogonal and suited to formal manipulation. In our syntax, the declarations above are written:

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

and the example document above is written:

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

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 displayed in Figure 1.

$$
\begin{array}{lllr}
g ::= & & & \textbf{group} \\
 & & \epsilon & \text{empty sequence} \\
 & | & g \texttt{,}\ g & \text{sequence} \\
 & | & \emptyset & \text{empty choice} \\
 & | & g \ \texttt{|}\ g & \text{choice} \\
 & | & g \ \texttt{\&}\ g & \text{interleaving} \\
 & | & g\{m,n\} & \text{repetition} \\
 & | & \textbf{mix}(g) & \text{mixed content} \\
 & | & x & \text{component name} \\
 & & & \\
m ::= & \langle\text{natural}\rangle & & \textbf{minimum}
\end{array}
\qquad
\begin{array}{lllr}
x ::= & & & \\
 & & @a & \text{attribute name} \\
 & | & \mathsf{e} & \text{element name} \\
 & | & t & \text{type name} \\
 & | & \textbf{anyType} & \\
 & | & \textbf{anyElem} & \\
 & | & \textbf{anySimpleType} & \\
 & | & p & \text{primitive} \\
 & & & \\
n ::= & & & \textbf{maximum} \\
 & & m & \text{bounded} \\
 & | & \infty & \text{unbounded}
\end{array}
$$

Fig. 1. Syntax of model groups.

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 are macros rather than a semantic feature.

Mixed content. Mixed content allows interspersing elements with text. More precisely, a document d matches $\textbf{mix}(g)$ 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`. 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 $[\,\mathsf{String}\ \texttt{:+:}\ [\![\texttt{em}]\!]_G\,]$—that is, a list, each element of which is either a string or a $[\![\texttt{em}]\!]_G$, the translation of `em`. (The subscript G stands for "group"; `:+:` is the binary cartesian sum.) 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, $[\![a\ \texttt{\&}\ b]\!] = a\texttt{,}\ b \texttt{ | } b\texttt{,}\ a$ but

$$[\![a\ \texttt{\&}\ b\ \texttt{\&}\ c]\!] \quad = \quad a\texttt{,}\ (b\texttt{,}\ c \texttt{ | } c\texttt{,}\ b) \quad \texttt{|} \quad b\texttt{,}\ (a\texttt{,}\ c \texttt{ | } c\texttt{,}\ a) \quad \texttt{|} \quad c\texttt{,}\ (a\texttt{,}\ b \texttt{ | } b\texttt{,}\ a)\ .$$

(Note that $[\![a\ \texttt{\&}\ b\ \texttt{\&}\ c]\!] \neq [\![a\ \texttt{\&}\ [\![b\ \texttt{\&}\ c]\!]]\!]$!)

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: a value may be used in any context where a "longer" value is expected. If extension is multiplicative in character (since it deals with projections from products), then the second kind of derivation, called *restriction*, is additive: it lets a value be used in any context where a less restrictive value is expected. For example, if one type involves a choice (a sum), then we can form a restriction-derived type by choosing one alternative or the other.

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 $(@a_1\ \&\ @a_2)\,,\ \mathsf{e}_1\,,\ \mathsf{e}_2$ not $@a_1\,,\mathsf{e}_1\,,@a_2\,,\mathsf{e}_2$.

To illustrate restriction, we declare a type *article* obtained from *publication* by reducing 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.

2.3 An overview of the translation

The objective of the translation from Schema to Haskell is to enable a programmer to write (Generic) 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 $[\![d]\!]_V^{g;u} :: [\![g]\!]_G\ [\![u]\!]_{mix}$.*

A detailed motivation of our translation scheme appears in section 5.2.

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

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

(Here, Void is the empty type; Either is the cartesian sum; and a value of Maybe a is either an a or the unit.)

This is the sort of translation employed by HaXml [53], and we do indeed follow a similar tack. In contrast, WASH [44] 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 None     u       -- no constructors
data Seq g1 g2 u = Seq  (g1 u) (g2 u)
data Or  g1 g2 u = Or1  (g1 u)
                 | Or2  (g2 u) .
```

Suppose G is the set of all Schema groups, and T is the set of Haskell types of kind $\star \to \star$. The type translation takes each group $g \in G$ to a Haskell type $\mathsf{T} \in T$:

$$\llbracket \epsilon \rrbracket_G = \mathsf{Empty} \qquad \llbracket \emptyset \rrbracket_G = \mathsf{None}$$
$$\llbracket g_1 \,,\, g_2 \rrbracket_G = \mathsf{Seq}\ \llbracket g_1 \rrbracket_G\ \llbracket g_2 \rrbracket_G \qquad \llbracket g_1 \,|\, g_2 \rrbracket_G = \mathsf{Or}\ \llbracket g_1 \rrbracket_G\ \llbracket g_2 \rrbracket_G\ .$$

The reason for the kindings and the significance of the type parameter u is explained below. Let us first call attention to the issue which the second half of this paper addresses.

The Schema operators and constants above satisfy the following equational laws for monoids.

$$g \,,\, \epsilon = g \qquad g \,|\, \emptyset = g$$
$$\epsilon \,,\, g = g \qquad \emptyset \,|\, g = g$$
$$(g_1 \,,\, g_2) \,,\, g_3 = g_1 \,,\, (g_2 \,,\, g_3) \qquad (g_1 \,|\, g_2) \,|\, g_3 = g_1 \,|\, (g_2 \,|\, g_3)$$

However, their images—Seq, Or, Empty and None—do not; put differently, the translation "function" $\llbracket - \rrbracket_G$ is not a homomorphism for this theory. For example, depending on how we parenthesize $g_1 \,,\, g_2 \,,\, g_3$, we may obtain:

$$\mathsf{Seq}\ (\mathsf{Seq}\ \llbracket g_1 \rrbracket_G\ \llbracket g_2 \rrbracket_G)\ \llbracket g_3 \rrbracket_G \qquad \text{or} \qquad \mathsf{Seq}\ \llbracket g_1 \rrbracket_G\ (\mathsf{Seq}\ \llbracket g_2 \rrbracket_G\ \llbracket g_3 \rrbracket_G)\ .$$

These types are isomorphic, but not equal; so the translation is not well-defined.

Unfortunately, Haskell does not provide any types which do satisfy the laws above: indeed, Haskell type constructors satisfy no nontrivial equations. A way to addess this problem is to promote the translation function $G \to T$ to a function from Schema types to *sets* of Haskell types, $G \to \mathcal{P}T$; this amounts to a binary relation $G \to T$. Thus, writing $S \ni x$ for $x \in S$, we revise our translation fragment above as follows.

$$\frac{}{\llbracket \epsilon \rrbracket_G \ni \mathsf{Empty}} \qquad \frac{}{\llbracket \emptyset \rrbracket_G \ni \mathsf{None}}$$

$$\frac{\llbracket g_1 \rrbracket_G \ni \mathsf{T}_1 \qquad \llbracket g_2 \rrbracket_G \ni \mathsf{T}_2}{\llbracket g_1 \,,\, g_2 \rrbracket_G \ni \mathsf{Seq}\ \mathsf{T}_1\ \mathsf{T}_2} \qquad \frac{\llbracket g_1 \rrbracket_G \ni \mathsf{T}_1 \qquad \llbracket g_2 \rrbracket_G \ni \mathsf{T}_2}{\llbracket g_1 \,|\, g_2 \rrbracket_G \ni \mathsf{Or}\ \mathsf{T}_1\ \mathsf{T}_2}$$

Since Seq and Or are associative "up to isomorphism", we might now seek to prove a result along the lines of, "if $\llbracket g \rrbracket_G \ni \mathsf{T}_1$ and $\llbracket g \rrbracket_G \ni \mathsf{T}_2$ then T_1 and T_2 are isomorphic." Unfortunately, a detailed proof could not be completed in time for publication; it will appear in the first author's dissertation.

However, it must be admitted that, as an abstract specification of a concrete implementation such as UUXML, this translation is of limited use without the inference mechanism. For example, given a schema type *date* such as

```
day, month, year
```

the user must assume that *date* can be translated as[2]

```
Seq Day (Seq Month Year)
```

in one part of the document, but

```
Seq (Seq Day Month) Year                or
Seq Day (Seq Month (Seq Year Empty))    or
Seq Empty (Seq Day (Seq Month Year))    or even
Seq Day (Seq (Seq Empty Empty) (Seq Month Year))
```

in another part of the document. In fact, the translation need not even be deterministic. Clearly some sort of uniformity is lost here.

An alternative is to normalize all sequences after translation, for example using the list-like form on the second line above. Unfortunately, this solution is little better. Suppose for example that *date* were used in the following two element type declaration contexts

```
def dateTime[date, time] ;
def timeDate[time, date] ;
```

where, for concreteness, let us assume *time* = hours, mins. In the first example the content would be translated as

```
Seq Day (Seq Month (Seq Year (Seq Hours (Seq Mins Empty))))
```

whereas in the second it would be translated as

```
Seq Hours (Seq Mins (Seq Day (Seq Month (Seq Year Empty)))) .
```

Mixed content. The reason each group g is translated to a higher 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.[3] Accordingly, u is restricted to be either String or (). For example, if $[\![e]\!]_G \ni \mathsf{E}$ and $[\![t]\!]_G \ni \mathsf{T}$, then $[\![\mathsf{e}[t]]\!]_G \ni$ Elem E T () when $\mathsf{e}[t]$ occurs in element-only content, but $[\![\mathsf{e}[t]]\!]_G \ni$ Elem E T String when it occurs in mixed content. The definition of Elem:

```
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' [20], 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
```

[2] UUXML would translate the names Day, Month and Year differently, but for clarity we ignore this detail here.

[3] We use the convention u for mixity because m is used for repetition 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 the value

$\quad$ *Mix* (*Seq* (*Elem* `"one"` *Empty*) (*Elem* `"two"` *Empty*)) `"three"`

whose type is

$\quad$ Mix (Seq (Elem E_1 Empty) (Elem E_2 Empty)) u

if $[\![\mathsf{e}_1]\!]_G = \{\mathsf{E}_1\}$ and $[\![\mathsf{e}_2]\!]_G = \{\mathsf{E}_2\}$ and the contents of e_1 and e_2 are always empty. (Recall that E_1, E_2 are phantom types, hence do not appear at the value level.)

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

$\quad$ **data** Seq g1 g2 u = *Seq* (g1 u) (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.

$\quad$ **data** Inter g1 g2 u = *Inter* (g1 u) (g2 u)

An unfortunate consequence of this choice of translation is that the ordering of the document values is lost. For example, suppose a schema 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:

$\quad$ **def** schedule[speaker & invitedSpeaker & keynoteSpeaker] ; .

A schema processor must know the order in which speakers appeared, but since the translation does not record the permutation an application 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.[4]

Repetition. Repetitions $g\{m, n\}$ are modeled using a datatype Rep $[\![g]\!]_G$ $[\![m, n]\!]_B$ u and a set of datatypes modeling bounds:

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

[4] Although the XML Infoset recommendation [50] (which attempts to provide some additional notion of semantics for XML documents) essentially states that attributes are unordered, the XML standard itself [45] does not.

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:

$$\begin{aligned} [\![\mathsf{e}\{2,4\}]\!]_G &= \{\mathsf{Rep}\ [\![\mathsf{e}]\!]_G\ (\mathsf{SS}\ (\mathsf{SS}\ (\mathsf{ZS}\ (\mathsf{ZS}\ \mathsf{ZZ}))))\} \\ [\![\mathsf{e}\{0,\infty\}]\!]_G &= \{\mathsf{Rep}\ [\![\mathsf{e}]\!]_G\ \mathsf{ZI}\} \\ [\![\mathsf{e}\{2,\infty\}]\!]_G &= \{\mathsf{Rep}\ [\![\mathsf{e}]\!]_G\ (\mathsf{SS}\ (\mathsf{SS}\ \mathsf{ZI}))\}\ . \end{aligned}$$

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 labeled with *LE_* ("Less than or Equal to") indicate the direct subtypes while the *EQ_* alternative ("EQual to") 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:

$$\mathit{fst}(\mathit{split}(t_1)) \mathbin{\&} \mathit{fst}(\mathit{split}(t_2')) \mathbin{,} (\mathit{snd}(\mathit{split}(t_1)) \mathbin{,} \mathit{snd}(\mathit{split}(t_2'))) .$$

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 closer correspondence exists 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 Formal translation

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.

$d ::=$		**document**	$di ::=$		**document item**
	ϵ	empty sequence		$a[d]$	attribute
$\mid$	$d \mathbin{,} d$	sequence	$\mid$	$e[d]$	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.* Figure 1 (Section 2.2) gives an overview of the operators which can occur in a group.

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.

(The *sort* of some content 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.* [9], which we summarize here informally. (We prefer not to reiterate the formal rules for validation because, as we shall see in section 3.1, they are easily read off from the rules for our translation of XML documents into Haskell values.)

ϵ matches the empty document. $g_1 \texttt{,}\ g_2$ matches a document matching g_1, followed by a document matching g_2. $\emptyset$ matches no document. $g_1 \texttt{|}\ g_2$ matches a document which matches either g_1 or g_2. $g\{m, n\}$ 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 $\textbf{mix}(g)$ 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.

$g_1 \,\texttt{\&}\, g_2$ behaves like $g_1 \texttt{,}\ g_2$ 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:

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

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.

3.1 Translating schemas to datatypes

Some semantic functions involved in the translation of a schema to a datatype are given in Figure 2. The function $\llbracket - \rrbracket_G$ translates model groups, while $\llbracket - \rrbracket_P$, $\llbracket - \rrbracket_X$, $\llbracket - \rrbracket_B$ and $\llbracket - \rrbracket_{mix}$ 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. Recall that each group is translated as a type of kind $\star \to \star$. (Of course, we stipulate that, for all groups g, $\llbracket g \rrbracket_G$ is the smallest set satisfying the rules.)

$$\frac{}{\llbracket \epsilon \rrbracket_G \ni \mathsf{Empty}} \qquad \frac{\llbracket g_1 \rrbracket_G \ni \mathsf{T}_1 \qquad \llbracket g_2 \rrbracket_G \ni \mathsf{T}_2}{\llbracket g_1 \, \mathsf{,} \; g_2 \rrbracket_G \ni \mathsf{Seq}\ \mathsf{T}_1\ \mathsf{T}_2}$$

$$\frac{}{\llbracket \emptyset \rrbracket_G \ni \mathsf{None}} \qquad \frac{\llbracket g_1 \rrbracket_G \ni \mathsf{T}_1 \qquad \llbracket g_2 \rrbracket_G \ni \mathsf{T}_2}{\llbracket g_1 \,\mathsf{|}\, g_2 \rrbracket_G \ni \mathsf{Or}\ \mathsf{T}_1\ \mathsf{T}_2}$$

$$\frac{\llbracket g_1 \rrbracket_G \ni \mathsf{T}_1 \qquad \llbracket g_2 \rrbracket_G \ni \mathsf{T}_2}{\llbracket g_1 \ \mathsf{\&}\ g_2 \rrbracket_G \ni \mathsf{Inter}\ \mathsf{T}_1\ \mathsf{T}_2} \qquad \frac{\llbracket g \rrbracket_G \ni \mathsf{T}}{\llbracket g\{m,n\} \rrbracket_G \ni \mathsf{Rep}\ \mathsf{T}\ \llbracket m,n \rrbracket_B}$$

$$\frac{}{\llbracket a \rrbracket_G \ni \llbracket a[\Sigma(a)] \rrbracket_G} \qquad \frac{}{\llbracket e \rrbracket_G \ni \llbracket e[\Sigma(e)] \rrbracket_G}$$

$$\frac{}{\llbracket a[s] \rrbracket_G \ni \mathsf{Attr}\ \llbracket a \rrbracket_X\ \llbracket s \rrbracket_X} \qquad \frac{}{\llbracket e[t] \rrbracket_G \ni \mathsf{Elem}\ \llbracket e \rrbracket_X\ \llbracket t \rrbracket_X}$$

$$\frac{\llbracket g \rrbracket_G \ni \mathsf{T}}{\llbracket \mathbf{mix}(g) \rrbracket_G \ni \mathsf{Mix}\ \mathsf{T}} \qquad \frac{}{\llbracket t \rrbracket_G \ni \llbracket t \rrbracket_X}$$

$$\llbracket \mathit{boolean} \rrbracket_P = \mathsf{T\_boolean} \qquad \llbracket \mathit{integer} \rrbracket_P = \mathsf{T\_integer}$$
$$\llbracket \mathit{double} \rrbracket_P = \mathsf{T\_double} \qquad \llbracket \mathit{string} \rrbracket_P = \mathsf{T\_string}$$

$$\llbracket \mathbf{elem} \rrbracket_{mix} = () \qquad \llbracket \mathbf{mix} \rrbracket_{mix} = \mathsf{String}$$

Fig. 2. Formal translation of types of a schema Σ.

The Haskell types mentioned in Figure 2 are defined in Section 2.3 with the following exceptions.

```
data Attr a g     u = Attr        (g u)
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 $[\![x]\!]_X$ = `"T_"`x; if an element name, then `"E_"`x; if an attribute name, then `"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 generate 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.

For example, the following schema declarations in a schema Σ:

$$\textbf{def}\ \mathsf{e}[g_e]\,;\quad \textbf{def}\ @a[g_a]\,;\quad \textbf{def}\ t = g_t\,;$$

produce bindings:

$$\Sigma(\mathsf{e}) = \mathsf{e}[g_e] \qquad \Sigma(@a) = @a[g_a] \qquad \Sigma(t) = g_t$$

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

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

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 $d, d' \in \mathit{cod}(h)$, $\mathit{dom}(d) \neq \mathit{dom}(d')$ 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 $H = (\{\mathit{Ty}\}, h)$ where $d = (\{C_1, C_2\}, g)$, $H(\mathit{Ty}) = d$, $d(C_1) = F$ and $d(C_2)$ is undefined would be realized as:

import *Def* -- defines Empty, Seq, Or, *etc.*

data Ty u = *C1* (F u) | *C2* .

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. $[\![\Sigma]\!]_S(\textbf{anyType})(\textbf{anyType})$ is undefined
3. $[\![\Sigma]\!]_S(\textbf{anyElem})(\textbf{anyElem})$ is undefined
4. for all x, $[\![\Sigma]\!]_S($"LE_"$x)($"EQ_"$x) = x$
5. for all x, x' s.t. $x <^e_\Sigma x'$ or $x <^r_\Sigma x'$, $[\![\Sigma]\!]_S($"LE_"$x')($"LE_"$x) =$ "LE_"x
6. for all x, x' s.t. $x <^r_\Sigma x'$ $[\![\Sigma]\!]_S(x)(x) \ni [\![\Sigma(x)]\!]_G$
7. for all x, x' s.t. $x <^e_\Sigma x'$ $[\![\Sigma]\!]_S(x)(x) \ni [\![(a_{x'}$ & $a_x)$, $c_{x'}$, $c_x]\!]_G$ 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 3. 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 (group,mixity)-indexed value translation function $[\![-]\!]^{g,u}_V$:

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

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

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

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

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

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

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

Fig. 3. Formal translation of values, part I: Documents.

Judgements of the form $c \overset{\text{str}}{\Rightarrow} v$ are axiomatic, and can be read "character data c translates to value v".

The soundness of this translation, shown in Section 3.2, ensures that $v :: \mathsf{T}\ [\![u]\!]_{mix}$, where $[\![g]\!]_G \ni \mathsf{T}$.

Interleaving The interleaving rule uses a proposition of the form $d \overset{\text{inter}}{\mapsto} d_1; d_2$, defined in Figure 4, which can be read, "document items in d can be permuted to yield a pair of documents d_1 and d_2."

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

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

Fig. 4. Formal translation of values, part II: Interleaving.

Repetition The rules for repetition given in Figure 5 employ propositions of the form $d \in_{g,u} \{m, n\} \overset{\text{rep}}{\Rightarrow} v$, where g is a group and u is a mixity, meaning that d validates against $g\{m, n\}$ producing a value v.

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

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

$$\text{Rep(3)}\frac{d_1 \in_u g \Rightarrow v_1 \qquad d_2 \in_u g\{0, m\} \Rightarrow v_2}{d_1\, \textbf{,}\ d_2 \in_{g,u} \{0, m+1\} \overset{\text{rep}}{\Rightarrow} \mathit{ZS}\ (\mathit{Just}\ v_1)\ v_2}$$

$$\text{Rep(3')}\frac{d_1 \in_u g \Rightarrow v_1 \qquad d_2 \in_{g,u} \{0, \infty\} \overset{\text{rep}}{\Rightarrow} \mathit{ZI}\ v_2}{d_1\, \textbf{,}\ d_2 \in_{g,u} \{0, \infty\} \overset{\text{rep}}{\Rightarrow} \mathit{ZI}\ (v_1 : v_2)}$$

Fig. 5. Formal translation of values, part III: Repetition.

Refinement The rules in Figure 6 define the witnesses to the refinement relation $<:$. If $g <: g' \overset{\text{ref}}{\Rightarrow} f$ then f is a coercion which witnesses the fact that we can upcast a value of type $\mathsf{T}_1\ \mathsf{a}$ to one of type $\mathsf{T}_2\ \mathsf{a}$, for any type a and where $\llbracket g \rrbracket_G \ni \mathsf{T}_1$ and $\llbracket g' \rrbracket_G \ni \mathsf{T}_2$.

$$\text{Reflex}\,\frac{}{g <: g \overset{\text{ref}}{\Rightarrow} id} \qquad \text{Trans}\,\frac{g <: g' \overset{\text{ref}}{\Rightarrow} f \qquad g' <: g'' \Rightarrow f'}{g <: g'' \overset{\text{ref}}{\Rightarrow} f' \circ f}$$

$$\text{Res}\,\frac{x <^{r}_{\Sigma} x'}{x <: x' \overset{\text{ref}}{\Rightarrow} \texttt{"LE\_"}x} \qquad \text{Ext}\,\frac{x <^{e}_{\Sigma} x'}{x <: x' \overset{\text{ref}}{\Rightarrow} \texttt{"LE\_"}x}$$

Fig. 6. Formal translation of values, part IV: refinement.

3.2 The correctness of the translation

In this section we show that the translation of schemas into Haskell is correct. The two 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 $\llbracket - \rrbracket_{mix}$ by $\llbracket - \rrbracket_m$.

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 $\llbracket - \rrbracket_G$ and $\llbracket - \rrbracket_V$ be respectively the type and value translations generated by a schema. Then, for all documents d, groups g and mixities u, $d \in_u g \implies \llbracket d \rrbracket_V^{g;u}$ and $\llbracket d \rrbracket_V^{g;u} :: \mathsf{T}\ \llbracket u \rrbracket_m$ for some $\mathsf{T} \in \llbracket g \rrbracket_G$.*

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. To cut down on the size of the rules, we write:

1. $d \in_u g \Rightarrow v :: \mathsf{T}$ to mean $d \in_u g \Rightarrow v$ and $v :: \mathsf{T}\ \llbracket u \rrbracket_m$ for some $\mathsf{T} \in \llbracket g \rrbracket_G$;
2. $d \in_{g,u} g\{m,n\} \overset{\text{rep}}{\Rightarrow} v :: \mathsf{T}$ to mean $d \in_{g,u} g\{m,n\} \overset{\text{rep}}{\Rightarrow} v$ and $v :: \mathsf{T}\ \llbracket u \rrbracket_m$ for some $\mathsf{T} \in \llbracket g \rrbracket_G$; and
3. $g <: g' \overset{\text{ref}}{\Rightarrow} f :: \forall \mathsf{a}.\ \mathsf{T}_1\ \mathsf{a} \rightarrow \mathsf{T}_2\ \mathsf{a}$ to mean $g <: g' \overset{\text{ref}}{\Rightarrow} v$ and $f :: \forall \mathsf{a}.\ \mathsf{T}_1\ \mathsf{a} \rightarrow \mathsf{T}_2\ \mathsf{a}$ for some $\mathsf{T}_1 \in \llbracket g \rrbracket_G$ and $\mathsf{T}_2 \in \llbracket g' \rrbracket_G$.

We present the annotated rules below; the proof is by inspection of the rules.

$$\text{Empty}\,\frac{}{\epsilon \in_u \epsilon \Rightarrow \mathit{Empty} :: \mathsf{Empty}}$$

$$\text{Seq}\,\frac{d_1 \in_u g_1 \Rightarrow v_1 :: \mathsf{T}_1 \qquad d_2 \in_u g_2 \Rightarrow v_2 :: \mathsf{T}_2}{d_1\,\mathbf{,}\ d_2 \in_u g_1\,\mathbf{,}\ g_2 \Rightarrow \mathit{Seq}\ v_1\ v_2 :: \mathsf{Seq}\ \mathsf{T}_1\ \mathsf{T}_2}$$

$$\text{Choice(1)}\,\frac{d \in_u g_1 \Rightarrow v_1 :: \mathsf{T}_1}{d \in_u g_1 \ \mathsf{|}\ g_2 \Rightarrow \mathit{Or1}\ \ v_1 :: \mathsf{Or}\ \mathsf{T}_1\ \mathsf{T}_2}$$

$$\text{Choice(2)}\,\frac{d \in_u g_2 \Rightarrow v_2 :: \mathsf{T}_2}{d \in_u g_1 \ \mathsf{|}\ g_2 \Rightarrow \mathit{Or2}\ \ v_2 :: \mathsf{Or}\ \mathsf{T}_1\ \mathsf{T}_2}$$

$$\text{Inter}\,\frac{d \overset{\text{inter}}{\mapsto} d_1; d_2 \qquad d_i \in_u g_i \Rightarrow v_i :: \mathsf{T}_i}{d \in_u g_1\ \&\ g_2 \Rightarrow \mathit{Inter}\ v_1\ v_2 :: \mathsf{Inter}\ \mathsf{T}_1\ \mathsf{T}_2}$$

$$\text{Name}\,\frac{d \in_u g \Rightarrow v :: \mathsf{T} \qquad g = \Sigma(x)}{d \in_u x \Rightarrow \texttt{"EQ\_"}\llbracket x \rrbracket_X\ (\llbracket x \rrbracket_X\ v) :: \texttt{"LE\_"}\llbracket x \rrbracket_X}$$

$$\text{Rep}\,\frac{d \in_{g,u} \{m,n\} \overset{\text{rep}}{\Rightarrow} v :: \llbracket m,n \rrbracket_B\ \mathsf{T}}{d \in_u g\{m,n\} \Rightarrow \mathit{Rep}\ v :: \mathsf{Rep}\ \mathsf{T}\ \llbracket m,n \rrbracket_B}$$

$$\text{Attr}\,\frac{d \in_{\mathbf{elem}} s \Rightarrow v :: \mathsf{T}}{a[d] \in_{\mathbf{elem}} a[s] \Rightarrow \mathit{Attr}\ v :: \mathsf{Attr}\ \llbracket a \rrbracket_X\ \mathsf{T}}$$

$$\text{Elem}\,\frac{d \in_u t \Rightarrow v :: \mathsf{T}_1 \qquad t <: t' \overset{\text{ref}}{\Rightarrow} f :: \forall \mathsf{a}.\ \mathsf{T}_1\ \mathsf{a} \rightarrow \mathsf{T}_2\ \mathsf{a}}{e[d] \in_{\mathbf{elem}} e[t'] \Rightarrow \mathit{Elem}\ ()\ (f\ v) :: \mathsf{Elem}\ \llbracket e \rrbracket_X\ \mathsf{T}_2}$$

$$\text{Refine}\,\frac{d \in_u e \Rightarrow v :: \mathsf{T}_1 \qquad e <: e' \overset{\text{ref}}{\Rightarrow} f :: \forall \mathsf{a}.\ \mathsf{T}_1\ \mathsf{a} \rightarrow \mathsf{T}_2\ \mathsf{a}}{d \in_u e' \Rightarrow f\ v :: \mathsf{T}_2}$$

$$\text{Mix(1)}\,\frac{d \in_{\mathbf{mix}} g \Rightarrow v :: \mathsf{T} \qquad c \overset{\text{str}}{\Rightarrow} v' :: \mathsf{String}}{d\mathbf{,}c \in_{\mathbf{elem}} \mathbf{mix}(g) \Rightarrow \mathit{Mix}\ v\ v' :: \mathsf{Mix}\ \mathsf{T}}$$

$$\text{Mix(2)}\,\frac{c \overset{\text{str}}{\Rightarrow} v :: \mathsf{String} \qquad e[d] \in_{\mathbf{elem}} g \Rightarrow \mathit{Elem}\ ()\ v' :: \mathsf{Elem}\ \llbracket e \rrbracket_X}{c\mathbf{,}e[d] \in_{\mathbf{mix}} g \Rightarrow \mathit{Elem}\ v\ v' :: \mathsf{Elem}\ \llbracket e \rrbracket_X\ \mathsf{T}}$$

$$\text{Rep(0)}\,\frac{}{\epsilon \in_{g,u} \{0,0\} \overset{\text{rep}}{\Rightarrow} \mathit{ZZ} :: \mathsf{ZZ}\ \mathsf{T}}$$

$$\text{Rep(1)}\,\frac{\epsilon_u \in g\{0,m\} \Rightarrow v :: \llbracket 0,m \rrbracket_B\ \mathsf{T}}{\epsilon \in_{g,u} \{0,m+1\} \overset{\text{rep}}{\Rightarrow} \mathit{ZS}\ \mathit{Nothing}\ v :: \mathsf{ZS}\ \llbracket 0,m \rrbracket_B\ \mathsf{T}}$$

$$\text{Rep(1')}\,\frac{}{\epsilon \in_{g,u} \{0,\infty\} \overset{\text{rep}}{\Rightarrow} \mathit{ZI}\ [\,] :: \mathsf{ZI}\ \mathsf{T}}$$

$$\text{Rep(2)}\,\frac{d_1 \in_u g \Rightarrow v_1 :: \mathsf{T} \qquad d_2 \in_{g,u} g\{m,n\} \overset{\text{rep}}{\Rightarrow} v_2 :: \llbracket m,n \rrbracket_B\ \mathsf{T}}{d_1\,\mathbf{,}\ d_2 \in_{g,u} \{m+1,n+1\} \overset{\text{rep}}{\Rightarrow} \mathit{SS}\ v_1\ v_2 :: \mathsf{SS}\ \llbracket m,n \rrbracket_B\ \mathsf{T}}$$

$$\text{Rep(3)}\,\frac{d_1 \in_u g \Rightarrow v_1 :: \mathsf{T} \qquad d_2 \in_u g\{0,m\} \Rightarrow v_2 :: \llbracket 0,m \rrbracket_B\ \mathsf{T}}{d_1\,\mathbf{,}\ d_2 \in_{g,u} \{0,m+1\} \overset{\text{rep}}{\Rightarrow} \mathit{ZS}\ (\mathit{Just}\ v_1)\ v_2 :: \mathsf{ZS}\ \llbracket 0,m \rrbracket_B\ \mathsf{T}}$$

$$\text{Rep(3')}\frac{d_1 \in_u g \Rightarrow v_1 :: \mathsf{T} \qquad d_2 \in_{g,u} \{0,\infty\} \overset{\text{rep}}{\Rightarrow} \mathit{ZI}\ v_2 :: [\![0,\infty]\!]_B\ \mathsf{T}}{d_1\,\textbf{,}\ d_2 \in_{g,u} \{0,\infty\} \overset{\text{rep}}{\Rightarrow} \mathit{ZI}\ (v_1 : v_2) :: \mathsf{ZI}\ [\![0,\infty]\!]_B\ \mathsf{T}}$$

$$\text{Reflex}\frac{}{g <: g \overset{\text{ref}}{\Rightarrow} \mathit{id} :: \forall \mathsf{a}.\ \mathsf{T\ a} \rightarrow \mathsf{T\ a}}$$

$$\text{Trans}\frac{g <: g' \overset{\text{ref}}{\Rightarrow} f :: \forall \mathsf{a}.\ \mathsf{T}_1\ \mathsf{a} \rightarrow \mathsf{T}_2\ \mathsf{a} \qquad g' <: g'' \overset{\text{ref}}{\Rightarrow} f' :: \forall \mathsf{a}.\ \mathsf{T}_2\ \mathsf{a} \rightarrow \mathsf{T}_3\ \mathsf{a}}{g <: g'' \overset{\text{ref}}{\Rightarrow} f' \circ f :: \forall \mathsf{a}.\ \mathsf{T}_1\ \mathsf{a} \rightarrow \mathsf{T}_3\ \mathsf{a}}$$

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

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

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 $[\![d]\!]_V^{t_1,u}$ into a t_2-translation $[\![d]\!]_V^{t_2,u}$. (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 *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 $f :: \forall \mathsf{a}.[\![t_1]\!]_X\ \mathsf{a} \rightarrow [\![t_2]\!]_X\ \mathsf{a}$ satisfying $f\ [\![d]\!]_V^{t_1,u} = [\![d]\!]_V^{t_2,u}$.*
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 $f = \mathit{id}$ 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 $f' :: \forall \mathsf{a}.[\![t_3]\!]_X\ \mathsf{a} \rightarrow [\![t_2]\!]_X\ \mathsf{a}$, and $f = f' \circ \texttt{"LE\_"}t_1$.

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.

4 Schema-aware applications

4.1 Generic programming in Generic Haskell

This section introduces generic programming in Generic Haskell. We give a brief introduction to Generic Haskell; more details can be found in [16, 24].

A generic program is a program that works for a large class of datatypes. A generic program takes a type as argument, and is usually defined by induction on the type structure. Generic Haskell is an extension of Haskell that supports generic programming. In this paper we use the most recent version of Generic Haskell, known as *Dependency-style Generic Haskell* [25, 24]. Dependencies both simplify and increase the expressiveness of generic programming.

In order to apply a program to values of different types, each datatype that appears in a source program is mapped to its structural representation. This representation is expressed in terms of a limited set of datatypes, called structure types. A generic program is defined by induction on these structure types. Whenever a generic program is applied to a user-defined datatype, the Generic Haskell compiler takes care of the mapping between the user-defined datatype and its corresponding structural representation. Furthermore, a generic program may also be directly defined on a user-defined datatype, in which case this definition takes precedence over the definitions generated for the structure type of the user-defined datatype. A definition of a generic function on a user-defined datatype is called a default case.

The translation of a datatype to a structure type replaces a choice between constructors by a sum, denoted by :+: (nested to the right if there are more than two constructors), and a sequence of arguments of a constructor by a product, denoted by :*: (nested to the right if there are more than two arguments). A nullary constructor is replaced by the structure type Unit. The arguments of the constructors are not translated. We give the structure type for the datatype of binary trees with integers in the leaves, and strings in the internal nodes.

```
data Tree        = Leaf Int | Node Tree String Tree
type Str (Tree) = Int :+: (Tree :*: (String :*: Tree)) .
```

Here *Str* is a meta function that given an argument type generates a new type name. The structural representation of a datatype only depends on the top level structure of a datatype. The arguments of the constructors, including recursive calls to the original datatype, appear in the representation type without modification. A type and its structural representation are isomorphic (ignoring undefined values). The isomorphism is witnessed by a so-called *embedding-projection pair*: a value of the datatype

```
data EP (a :: ∗) (b :: ∗) = EP (a → b) (b → a) .
```

The Generic Haskell compiler generates the translation of a type to its structural representation, together with the corresponding embedding projection pair.

From this translation, it follows that it suffices to define a generic function on sums (:+:), products (:*: and Unit) and on base types such as Int and String. To be able to inspect constructor names, we will also encode the constructors in the structure type of a datatype, using the structure type Con defined by

data Con a = *Con* ConDescr a

where the type ConDescr is used for constructor descriptions. The structure type for Tree now becomes

type *Str* (Tree) = Con Int :+: Con (Tree :*: (String :*: Tree)) .

For example, we define a very simple generic function *content* that extracts the strings and integers (shown as strings) that appear in a value of a datatype t. The instance of *content* on the type Tree returns the following strings when applied to *Node* (*Leaf* 3) "Bla" (*Leaf* 7): ["3", "Bla", "7"].

The generic function *content* returns the document's content for any datatype[5]:

```
content{|t :: ⋆|}                :: (content{|t|}) ⇒ t → [String]
content{|Unit|}     Unit         = []
content{|Int|}      int          = [show int]
content{|String|}   str          = [str]
content{|a :+: b|}  (Inl a)      = content{|a|} a
content{|a :+: b|}  (Inr b)      = content{|b|} b
content{|a :*: b|}  (a :*: b)    = content{|a|} a ++ content{|b|} b
content{|Con c a|}  (Con a)      = content{|a|} a
```

There are a couple of things to note about generic function definitions:

– Function *content*{|t|} is a type-indexed function. The type argument appears in between special parentheses {|, |}. An instance of *content* is obtained by applying *content* a type.
– The constraint *content*{|t|} that appears in the type of function *content* says that function *content depends on* itself. A generic function *f* depends on a generic function *g* if there is an arm in the definition of *f*, for example that for *f*{|a :+: b|}, which uses *g* on a variable in the type argument, for example *g*{|a|}. If a generic function depends on itself it is defined by induction over the type structure.
– The type of function *content* is given for a type t of kind $\star$. This does not mean that *content* can only be applied to types of kind $\star$; it only gives the type information for types of kind $\star$. The type of function *content* on types with kinds other than $\star$ can automatically be derived from this base type.
– Using an accumulating parameter we can obtain a more efficient version of function *content*.

4.2 From XML documents to Haskell data

In this section we describe an implementation of the translation outlined in the previous subsection 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 Doc which presents a document as a tree (or rather a forest):

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

[5] The (infix) ++ function returns the concatenation of its two input strings.

We use standard techniques [19] 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. *gParse*{|t|} denotes a parser which tries to read a document into a value of type t. The type of generic parsers is:

```
gParse{|t :: ⋆|} :: (gParse{|t|}, gName{|t|}, gInter{|t|}, empty{|t|}) ⇒ P t .
```

This function depends on several auxiliary values which are defined below. We now describe its behavior on the various components of Schema.

```
gParse{|String|} = pMixed
gParse{|Unit|}   = pElementOnly
```

The first two cases handle mixities: *pMixed* matches a sequence of *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 then it tries again with *q*, and *mzero* is the identity element for <|>. *fmap* is the action of a functor on a function.

```
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 a sequence of documents matching *p* and an optional *p*, respectively.

```
gParse{|T_string  u|} = fmap T_string pText
gParse{|T_integer u|} = fmap T_integer pReadableText
```

String primitives are handled by a parser *pText*, which matches a sequence of *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.

```
gParse{|Elem e g u|} = do mixity ← gParse{|u|}
                          let p = gParse{|g|} pElementOnly
                          elemt gName{|e|} (fmap (Elem mixity) 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 *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.

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

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, $gInter\{\!|\mathsf{t}|\!\}$, where t is in our case always instantiated to a component group. The function has the following type:

$$gInter\{\!|\mathsf{t} :: \star|\!\} :: (gInter\{\!|\mathsf{t}|\!\}, empty\{\!|\mathsf{t}|\!\}) \Rightarrow \forall \mathsf{a}.\ \mathsf{PermP}\ (\mathsf{t} \rightarrow \mathsf{a}) \rightarrow \mathsf{PermP\ a}\ .$$

Interleaving is handled using these permutation phrase combinators [5]:

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

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 $<\|>$ or, if optional, $<|?>$; for example, $<\|>$ takes a permutation parser returning continuations for a-values with b-answers, and an ordinary parser returning an a-value, and produces a permutation parser returning a b-value. *mapPerms* is the standard map function for the PermP type. *permute q* converts a permutation parser q into a normal parser.

$$\begin{aligned} & gParse\{\!|\mathsf{Inter\ g1\ g2\ u}|\!\} = \\ & \quad permute\ \$\ (gInter\{\!|\mathsf{g2\ u}|\!\} \circ gInter\{\!|\mathsf{g1\ u}|\!\})\ (newperm\ Inter) \end{aligned}$$

To see how the above code works, observe that:

$$\begin{aligned} & f1 = gInter\{\!|\mathsf{g1\ u}|\!\} :: \forall \mathsf{u\ b}.\ \mathsf{PermP}\ (\mathsf{g1\ u} \rightarrow \mathsf{b}) \rightarrow \mathsf{PermP\ b} \\ & f2 = gInter\{\!|\mathsf{g2\ u}|\!\} :: \forall \mathsf{u\ c}.\ \mathsf{PermP}\ (\mathsf{g2\ u} \rightarrow \mathsf{c}) \rightarrow \mathsf{PermP\ c} \\ & hence \\ & f2 \circ f1 \qquad\qquad\quad :: \forall \mathsf{u\ c}.\ \mathsf{PermP}\ (\mathsf{g1\ u} \rightarrow \mathsf{g2\ u} \rightarrow \mathsf{c}) \rightarrow \mathsf{PermP\ c}\ . \end{aligned}$$

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.

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

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

```
gInter{|Con c a|}           = (<||> fmap Con gParse{|a|})
gInter{|Inter g1 g2 u|}     = gInter{|g1 u|} ∘ gInter{|g2 u|}
                                ∘ mapPerms (λf x y → f (Inter x y))
gInter{|Rep g (ZS ZZ) u|} = (<|?> (Rep (empty{|(ZS ZZ) g u|})
                                      ,fmap Rep gParse{|(ZS ZZ) g u|}))
```

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 ZS ZZ: this corresponds to a Schema type like $\mathsf{e}\{0,1\}$, that is, an optional element (or attribute).[6] *empty*{|t|} is a simple auxiliary function which provides a value of the argument type when the optional value is omitted; its definition can be found in Löh's dissertation [24].

4.3 From Haskell data to XML documents

We now describe a generic printer for the XML translation types we have introduced. Printing to XML is actually easier than parsing it; as before, rather than producing strings, we use the Doc type as a more abstract representation of XML.

The generic function *gPrint*{|t|} accepts a value of type t (which is assumed to be in the range of the translation) and produces a Doc-transformer, that is, a function of type Doc → Doc. *gPrint*{|t|} depends on itself and the auxiliary function *gName*{|t|}, which is described above in Section 4.2.

```
gPrint{|t :: ⋆|} :: (gParse{|t|}, gName{|t|}) ⇒ t → Doc → Doc
```

To produce an actual Doc, one merely applies the result to the empty list. This formulation follows a familiar idiom in Haskell, which avoids the quadratic complexity of the list concatenation operator.

```
gPrint{|Unit|} Unit = id
gPrint{|String|} s    = (DText s:)
```

These first two cases handle mixities. In an element-only context, we simply pass the input document through; otherwise, we cons on a *DText* chunk.

```
gPrint{|a :*: b|} (x1 :*: x2)          = gPrint{|a|} x1 ∘ gPrint{|b|} x2
gPrint{|a :+: b|} (Inl x1)             = gPrint{|a|} x1
gPrint{|a :+: b|} (Inr x2)             = gPrint{|b|} x2
gPrint{|Empty|} Empty                  = id
```

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

$$\begin{aligned}
&gPrint\{\!|\mathsf{Seq\ g1\ g2\ u}|\!\}\ (Seq\ x1\ x2) &&= gPrint\{\!|\mathsf{g1\ u}|\!\} \circ gPrint\{\!|\mathsf{g2\ u}|\!\}\\
&gPrint\{\!|\mathsf{Inter\ g1\ g2\ u}|\!\}\ (Inter\ x1\ x2) &&= gPrint\{\!|\mathsf{g1\ u}|\!\} \circ gPrint\{\!|\mathsf{g2\ u}|\!\}\\
&gPrint\{\!|\mathsf{None}|\!\} &&= \bot\\
&gPrint\{\!|\mathsf{Or\ g1\ g2\ u}|\!\}\ (Or1\ x1) &&= gPrint\{\!|\mathsf{g1\ u}|\!\}\ x1\\
&gPrint\{\!|\mathsf{Or\ g1\ g2\ u}|\!\}\ (Or2\ x2) &&= gPrint\{\!|\mathsf{g2\ u}|\!\}\ x2
\end{aligned}$$

These cases handle the model groups. Sequences and interleavings produce transformers which are just composed, as this corresponds to appending.

$$\begin{aligned}
&gPrint\{\!|\mathsf{Elem\ e\ g\ u}|\!\}\ (Elem\ m\ x) = gPrint\{\!|\mathsf{u}|\!\}\ m \circ\\
&\qquad\qquad (DElem\ gName\{\!|\mathsf{e}|\!\}\ (gPrint\{\!|\mathsf{g}\ ()|\!\}\ x\ [\,]):)
\end{aligned}$$

To handle elements, we first output the mixity value preceding it, then cons on a *DElem* block. The recursive call descends into the descendants of the element, terminating it with an empty list to produce a Doc.

$$gPrint\{\!|\mathsf{Mix\ g\ u}|\!\}\ (Mix\ x\ m) = gPrint\{\!|\mathsf{g\ String}|\!\}\ x \circ (DText\ m{:})$$

To handle Mix, we first output the XML x, then any trailing text.

$$\begin{aligned}
&gPrint\{\!|\mathsf{T\_string\ u}|\!\}\ (T\_string\ x) &&= (DText\ x{:})\\
&gPrint\{\!|\mathsf{T\_integer\ u}|\!\}\ (T\_integer\ x) &&= (DText\ (show\ x){:})
\end{aligned}$$

Values of the primitive types are simply output as strings, using *DText* chunks. *show* is a Haskell Prelude function which converts a value of any type to a string.

$$\begin{aligned}
&gPrint\{\!|\mathsf{Rep\ g\ b\ u}|\!\}\ (Rep\ x) &&= gPrint\{\!|\mathsf{b\ g\ u}|\!\}\ x\\
&gPrint\{\!|\mathsf{ZZ\ g\ u}|\!\}\ ZZ &&= id\\
&gPrint\{\!|\mathsf{ZI\ g\ u}|\!\}\ (ZI\ x) &&= foldr\ (\circ)\ id\ (map\ gPrint\{\!|\mathsf{g\ u}|\!\}\ x)\\
&gPrint\{\!|\mathsf{ZS\ b\ g\ u}|\!\}\ (ZS\ x\ rep) &&= maybe\ id\ gPrint\{\!|\mathsf{g\ u}|\!\}\ x \circ\\
& &&\quad gPrint\{\!|\mathsf{Rep\ g\ b\ u}|\!\}\ rep\\
&gPrint\{\!|\mathsf{SS\ b\ g\ u}|\!\}\ (SS\ x\ rep) &&= gPrint\{\!|\mathsf{g\ u}|\!\}\ x \circ gPrint\{\!|\mathsf{Rep\ g\ b\ u}|\!\}\ rep
\end{aligned}$$

Repetition is mostly self-explanatory. Here we use the Haskell Prelude function *foldr*, the familiar list induction operator, and *map*, the list functor action. *maybe* is another standard function, defined by:

$$\begin{aligned}
&maybe &&:: b \to (a \to b) \to \mathsf{Maybe}\ a \to b\\
&maybe\ x\ f\ Nothing &&= x\\
&maybe\ x\ f\ (Just\ a) &&= f\ a
\end{aligned}$$

5 Improving UUXML

The default translation scheme of a data binding may produce unwieldy, convoluted and redundant types and values. Our own Haskell–XML Schema binding UUXML suffers from this problem.

In this section we use UUXML as a case study of the problem of overwhelmingly complex data representation which tends to accompany type-safe language embeddings. We outline the proble and explain how the design criteria gave rise to it. In Section 6, we show how to address the problem using iso inference. The same technique might be used in other situations, for example, compilers and similar language processors which are designed to exploit type-safe data representations.

5.1 The Problem with UUXML

Let us briefly give the reader a sense of the magnitude of the problem.

Consider the following XML schema, which describes a simple bibliographic record `doc` including a sequence of authors, a title and an optional publication date, which is a year followed by a month. It is a variant of the schema introduced in Section 2.2.

```
<element name="doc" type="docType"/>
<complexType name="docType">
  <sequence>
    <element ref="author" minOccurs="0" maxOccurs="unbounded"/>
    <element ref="title"/>
    <element ref="pubDate" minOccurs="0"/>
  </sequence>
  <attribute name="key" type="string"/>
</complexType>
<element name="author" type="string"/>
<element name="title"  type="string"/>
<complexType name="pubDateType">
  <sequence>
    <element ref="year"/>
    <element ref="month"/>
  </sequence>
</complexType>
<element name="pubDate" type="pubDateType"/>
<element name="year"    type="int"/>
<element name="month"   type="int"/>
```

An example document which validates against this schema is:

```
<doc key="homer-odyss">
  <author>Homer</author>
  <title>The Odyssey</title>
</doc> .
```

UUXML translates each of the types `doc` and `docType` into a pair of types,

```
data E_doc u          = E_doc (Elem LE_E_doc LE_T_docType u)
data LE_E_doc u       = EQ_E_doc (E_doc u)
data T_docType u      = T_docType (Seq A_key (Seq (Rep LE_E_author ZI)
                                  (Seq LE_E_title (Rep LE_E_pubDate
                                  (ZS ZZ)))) u)
data LE_T_docType u   = EQ_E_docType (T_docType u)
                      | LE_T_publicationType (LE_T_publicationType u)
```

and the example document above into the value:

EQ_E_doc (*E_doc* (*Elem* () (*EQ_T_docType* (*T_docType* (*Seq* (*A_key* (*Attr* (*EQ_T_string* (*T_string* `"homer-odyss"`))))(*Seq* (*Rep* (*ZI* [*EQ_E_author* (*E_author* (*Elem* () (*EQ_T_string* (*T_string* `"Homer"`))))])) (*Seq* (*EQ_E_title*

$(E\_title\ (Elem\ ()\ (EQ\_T\_string\ (T\_string$ `"The Odyssey"`$)))))\ (Rep$
$(ZS\ Nothing\ (Rep\ ZZ)))))))))$
which has type `LE_E_doc` ().

The problem is clear: if a user wants to, say, retrieve the content of the `author` field, he or she must pattern-match against no less than ten constructors before reaching `"Homer"`. For larger, more complex documents or document types, the problem can be even worse.

5.2 Conflicting Issues in UUXML

UUXML's usability issues are a side effect of its design goals. We discuss these here in some depth, and close by suggesting why similar issues may plague other applications which process typed languages.

First, UUXML is type-safe and preserves as much static type information as possible to eliminate the possibility of constructing invalid documents. In contrast, Java–XML bindings tend to ignore a great deal of type information, such as the types of repeated elements (only partly because of the limitations of Java collections).

Second, UUXML translates (a sublanguage of) XML Schema types rather than the less expressive DTDs. This entails additional complexity compared with bindings such as HaXML [53] that merely target DTDs. For example, XML Schema supports not just one but two distinct notions of subtyping and a more general treatment of mixed content than DTDs.

Third, the UUXML translation closely follows the Model Schema Language (MSL) formal semantics [9], even going so far as to replicate that formalism's abstract syntax as closely as Haskell's type syntax allows. This has advantages: we have been able to prove the soundness of the translation, that is, that valid documents translate to typeable values, and the translator is relatively easy to correctly implement and maintain. However, our strict adherence to MSL has introduced a number of 'dummy constructors' and 'wrappers' which could otherwise be eliminated.

Fourth, since Haskell does not directly support subtyping and XML Schema does, our binding tool emits a *pair* of Haskell datatypes for each schema type `t`: an 'equational' variant which represents documents which validate *exactly* against `t`, and a 'down-closed' variant, which represents all documents which validate against all subtypes of `t`. Our expectation was that a typical Haskell user would read a document into the down-closed variant, pattern-match against it to determine which exact/equational type was used, and do the bulk of their computation using that.

Finally, UUXML was intended, first and foremost, to support the development of 'schema-aware' XML applications using Generic Haskell. This moniker describes programs, such as our XML compressor XComprez [2], which operate on documents of any schema, but not necessarily parametrically. XComprez, for example, exploits the type information of a schema to improve compression ratios.

Because Generic Haskell works by traversing the structure of datatypes, we could not employ methods, such as those in WASH [44], which encode schema information in non-structural channels such as Haskell's type class system. Such information is instead necessarily expressed in the structure of UUXML's types, and makes them more complex.

For schema-aware applications this complexity is not such an issue, since generic functions typically need not pattern-match deeply into a datatype; for example, the parser *gParse*{|t|} of Section 4.2 is a schema-aware application, and so does not depend explicitly on the details of a particular DTD. But if we aim to use UUXML for more conventional applications it can become an overwhelming problem; for example, a function that needs to extract the bibliographical `key` attribute from our Homer example needs to dig past ten constructors!

In closing, we emphasize that many similar issues are likely to arise, not only with other data bindings and machine-generated programs, but also with any type-safe representation of a typed object language in a metalanguage such as Haskell. Preserving the type information necessarily complicates the representation. If the overall 'style' of the object language is to be preserved, as was our desire in staying close to MSL, then the representation is further complicated. If subtyping is involved, even more so. If the representation is intended to support generic programming, then it must express as much information as possible structurally, which also entails some complexity.

For reasons such as these, one might be tempted to eschew type-safe embeddings entirely, but then what is the point of programming in a statically typed language if not to exploit the type system? Arguably, the complexity problem arises not from static typing itself, but rather the insistence on using only a *single* data representation. In the next section, we show how iso inference drastically simplifies dealing with *multiple* data representations.

6 Inferring Isomorphisms

Typed functional languages like Haskell and ML [23, 31] typically support the declaration of user-defined, polymorphic algebraic datatypes. In Haskell, for example, we might define a datatype representing dates in a number of ways. The most straightforward and conventional definition is probably the one given by Date below,

```
data Date = Date Int Int Int
```

but a more conscientious Dutch programmer might prefer Date_NL:

```
data Date_NL = Date_NL Day Month Year
data Day     = Day     Int
data Month   = Month   Int
data Year    = Year    Int .
```

An American programmer, on the other hand, might opt for Date_US, which follows the US date format:

```
data Date_US = Date_US Month Day Year .
```

If the programmer has access to an existing library which can compute with dates given as Int-triples, though, they may prefer $\mathsf{Date2}$,

$\textbf{data}\ \mathsf{Date2} = \mathit{Date2}\ (\mathsf{Int}, \mathsf{Int}, \mathsf{Int})$,

for the sake of simplifying data conversion between his application and the library. In some cases, for example when the datatype declarations are machine-generated, a programmer might even have to deal with more unusual declarations such as:

$\textbf{data}\ \mathsf{Date3} = \mathit{Date3}\ (\mathsf{Int}, (\mathsf{Int}, \mathsf{Int}))$
$\textbf{data}\ \mathsf{Date4} = \mathit{Date4}\ ((\mathsf{Int}, \mathsf{Int}), \mathsf{Int})$
$\textbf{data}\ \mathsf{Date5} = \mathit{Date5}\ (\mathsf{Int}, (\mathsf{Int}, (\mathsf{Int}, ())))$.

Although these types all represent the same abstract data structure[7], they represent it differently; they are certainly all unequal, firstly because they have different names, but more fundamentally because they exhibit different surface structures. Consequently, programs which use two or more of these types together must be peppered with applications of conversion functions.

Medium-size and large programs typically use many client libraries from disparate sources. In programs of this size, it becomes inevitable that two or more libraries export distinct datatypes that share the same (or subsume another's) semantics. In order for the clients to interoperate, the host program must resolve these differences by converting between representations.

In our example above, the amount of code required to define such conversion functions is not so large, but if the declarations are machine-generated, or the number of representations to be simultaneously supported is large, then the size of the conversion code can become unmanageable.

6.1 Isomorphisms

The fact that all these types represent the same abstract type is captured by the relation $\cong$ of *isomorphy*: types A and B are isomorphic, $A \cong B$, iff there exists an invertible function $f : A \to B$. The witness f is called an *isomorphism* (or *iso*) and serves as our desired conversion function. The identity function is an iso, as is the composition of two isos; thus, using invertibility, one easily obtains that isomorphy is an equivalence relation.

Some familiar isos arise from the semantics of base types. For example, modulo numeric precision issues, the conversion between meters and miles is an isomorphism between the floating-point type Double and itself; if we preserve the origin, the conversion between cartesian and polar coordinates is another example.

Some polymorphic isos arise from the structure of types themselves; for example, one says that products are commutative "up to isomorphism," meaning that there exists an iso $\gamma : \forall a, b.\, a \times b \to b \times a$. Mac Lane was the first to

[7] We will assume all datatypes are strict; otherwise, Haskell's non-strict semantics entails that some transformations such as nesting add a new value $\perp$ which renders this claim false. In practice, though, only programs which depend essentially on laziness are sensitive to this issue.

show [26], in the setting of category theory, that a stronger result holds: γ is the *only* polymorphic function of this type, among a certain class of polymorphic functions. Moreover, he showed that γ can be constructed, entirely by composition and functor application, from a basis of canonical functions. (A more recent treatment of this result appears in Mac Lane's book [27].) Such theorems are called *coherence theorems*, and Mac Lane called functions like γ *canonical*.

In other words, canonical functions are special because they are uniquely determined by their type; since there is exactly one canonical function at certain types, they serve as a natural form of coercion. Furthermore, since canonicality is preserved by composition and certain functors, such coercions do not suffer from the incoherence issues that can plague programming in languages with *ad hoc* coercion disciplines.

Pierce [36] gives the following example of incoherence. Suppose there exist the following implicit coercions:

$$\begin{aligned}
[\![\mathsf{Bool} \leqslant \mathsf{Int}]\!] &= \lambda b \to \textbf{if}\ b\ \textbf{then}\ 1\ \textbf{else}\ 0\\
[\![\mathsf{Int} \leqslant \mathsf{String}]\!] &= \mathit{int2String}\\
[\![\mathsf{Bool} \leqslant \mathsf{Double}]\!] &= \lambda b \to \textbf{if}\ b\ \textbf{then}\ 1.0\ \textbf{else}\ 0.0\\
[\![\mathsf{Double} \leqslant \mathsf{String}]\!] &= \mathit{double2String}\ ,
\end{aligned}$$

where, for the purpose of this example, we suppose that

$$\mathit{int2String}\ 1 = \texttt{"1"} \qquad\qquad \mathit{float2String}\ 1.0 = \texttt{"1.0"}\ .$$

Now consider the semantics of the term $t = \mathit{putStr}\ \mathit{True}$, where $\mathit{putStr}\ s$ denotes the I/O action which prints the string s. The term t is typeable because $\mathit{True} :: \mathsf{Bool}$ can be coerced to a value of type String; however, the rules provide two possible coercions:

$$\begin{tikzcd}
 & \mathsf{Int} \arrow[dr, "\mathit{int2String}"] & \\
\mathsf{Bool} \arrow[ur, "\lambda b \to \textbf{if}\ b\ \textbf{then}\ 1\ \textbf{else}\ 0"] \arrow[dr, "\lambda b \to \textbf{if}\ b\ \textbf{then}\ 1.0\ \textbf{else}\ 0.0"'] & & \mathsf{String} \\
 & \mathsf{Double} \arrow[ur, "\mathit{double2String}"'] &
\end{tikzcd}$$

which, however, also produce two different values for t: the upper composition maps *True* to an action printing `"1"`, the lower to one printing `"1.0"`.[8]

Haskell's system of type classes itself suffers from this problem. A classic example involves the overloaded functions *read* and *show*, which convert any value from and to their external representations as strings. But consider the program *read* (*show True*); according to the Haskell 98 Report [35] this program

[8] After adding the *ad hoc* coercions described at the end of Section 7.4 to our implementation, we encountered this very problem and spent half an hour searching for its cause!

should raise a static error because, without a proper type annotation, its dynamic semantics is ambiguous: they depend on the type assigned to it, yet any type will do. For example, *read* (*show True*) :: Int fails while *read* (*show True*) :: Bool does not. In fact, the example without the type annotation is accepted by the Glasgow Haskell Compiler and fails, though another implementation (that ignores the Report's admonition to raise an error in ambiguous cases) may well evaluate this term to *True*.

6.2 Monoidal isos.

A few canonical isos of (Generic) Haskell are summarized by the syntactic theory below.

$$\mathsf{a}\ {:}{*}{:}\ \mathsf{Unit} \cong \mathsf{a} \qquad \mathsf{Unit}\ {:}{*}{:}\ \mathsf{a} \cong \mathsf{a} \qquad (\mathsf{a}\ {:}{*}{:}\ \mathsf{b})\ {:}{*}{:}\ \mathsf{c} \cong \mathsf{a}\ {:}{*}{:}\ (\mathsf{b}\ {:}{*}{:}\ \mathsf{c})$$
$$(\mathsf{a}\ {:}{+}{:}\ \mathsf{b})\ {:}{+}{:}\ \mathsf{c} \cong \mathsf{a}\ {:}{+}{:}\ (\mathsf{b}\ {:}{+}{:}\ \mathsf{c})$$

The isomorphisms which witness these identities are the evident ones. The first two identities in each row express the fact that Unit is a right and left unit for :∗:; the last two say that :∗: (resp. :+:) is associative. We call these isos collectively the *monoidal isos*.

This list is not exhaustive. For example, binary product and sum are also commutative up to canonical iso:

$$\mathsf{a}\ {:}{*}{:}\ \mathsf{b} \cong \mathsf{b}\ {:}{*}{:}\ \mathsf{a} \qquad\qquad \mathsf{a}\ {:}{+}{:}\ \mathsf{b} \cong \mathsf{b}\ {:}{+}{:}\ \mathsf{a}$$

and the distributivity iso is also canonical:

$$\mathsf{a}\ {:}{*}{:}\ (\mathsf{b}\ {:}{+}{:}\ \mathsf{c}) \cong (\mathsf{a}\ {:}{*}{:}\ \mathsf{b})\ {:}{+}{:}\ (\mathsf{a}\ {:}{*}{:}\ \mathsf{c})\ .$$

There is a subtle but important difference between the monoidal isos and the other isos mentioned above. Although all are canonical, and so possess unique polymorphic witnesses determined by the *type schemes* involved, only in the case of the monoidal isos does the uniqueness property transfer unconditionally to the setting of *types*.

To see this, consider instantiating the product-commutativity iso scheme to obtain:

$$\mathsf{Int}\ {:}{*}{:}\ \mathsf{Int} \cong \mathsf{Int}\ {:}{*}{:}\ \mathsf{Int}\ .$$

This identity has two witnesses: one is the intended twist map, but the other is the identity function.

This distinction is in part attributable to the form of the identities involved; the monoidal isos are all *strongly regular*, that is:

1. each variable that occurs on the left-hand side of an identity occurs exactly once on the right-hand side, and *vice versa*, and
2. variables occur in the same order on both sides.

The strong regularity condition is adapted from work on generalized multicategories [22, 21, 15]. It is a sufficient—but not necessary—condition to ensure that a pair of types determines a unique canonical iso witness.

Thanks to the canonicality and strong regularity properties, given two types we can determine if a unique iso between them exists, and if so can generate it automatically. Thus our program infers all the monoidal isos, but not the commutativity or distributivity isos.

6.3 Datatype isos.

As explained in Section 4.1, in Generic Haskell each datatype declaration induces a canonical iso between the datatype t and an underlying structure type. For example, the declaration

$\quad\textbf{data}\ \mathit{ListInt} = \mathit{Nil} \mid \mathit{Cons}\ \mathsf{Int}\ \mathit{ListInt}$

induces a canonical isomorphism

$$\mathit{ListInt} \cong \mathsf{Unit} \mathbin{:+:} (\mathsf{Int} \mathbin{:*:} \mathit{ListInt})\ .$$

for each instantiation type A. We call such isos *datatype isos*; their canonical status can be seen as arising from the universal property characterizing initial algebras.

6.4 The reduce/expand paradigm

From a Generic Haskell user's point of view, iso inference is a simple matter of applying two generic functions,

$\quad \mathit{reduce}\{\!|\mathsf{t}|\!\}\ \ :: \mathsf{t} \to \mathsf{Univ}$
$\quad \mathit{expand}\{\!|\mathsf{t'}|\!\} :: \mathsf{Univ} \to \mathsf{t'}\ .$

$\mathit{reduce}\{\!|\mathsf{t}|\!\}$ takes a value of any type and converts it to a value of a universal, normalized representation denoted by the type Univ; $\mathit{expand}\{\!|\mathsf{t'}|\!\}$, its dual, converts such a universal value back to a 'regular' value, if possible. The iso which converts from t to $\mathsf{t'}$ is thus expressed as:

$\quad \mathit{expand}\{\!|\mathsf{t'}|\!\} \circ \mathit{reduce}\{\!|\mathsf{t}|\!\}\ .$

If $\mathsf{t} = \mathsf{t'}$, then $\mathit{expand}\{\!|\mathsf{t'}|\!\}$ and $\mathit{reduce}\{\!|\mathsf{t}|\!\}$ are mutual inverses. If t and $\mathsf{t'}$ are merely isomorphic, then expansion *may* fail at run-time; it succeeds if the two types are *canonically* isomorphic, $\mathsf{t} \cong \mathsf{t'}$, according to the monoidal and datatype iso theories.

As an example, consider the expression

$\quad (\mathit{expand}\{\!|(\mathsf{Bool}, \mathsf{Bool} \mathbin{:+:} (\mathsf{Int} \mathbin{:+:} \mathsf{String}))|\!\} \circ$
$\quad\quad \mathit{reduce}\{\!|(\mathsf{Bool}, ((), (\mathsf{Bool} \mathbin{:+:} \mathsf{Int}) \mathbin{:+:} \mathsf{String}))|\!\})$
$\quad\quad\quad (\mathit{True}, ((), \mathit{Inl}\ (\mathit{Inr}\ 7)))\ ,$

which evaluates to

$\quad (\mathit{True}, \mathit{Inr}\ (\mathit{Inl}\ 7))\ .$

Function $\mathit{reduce}\{\!|\mathsf{t}|\!\}$ picks a type in each isomorphism class which serves as a normal form, and uses the canonical witness to convert values of t to that form. Normalized values are represented in a special way in the abstract type Univ; a typical user need not understand the internals of Univ unless $\mathit{expand}\{\!|\mathsf{t}'|\!\}$ fails. If t and t′ are 'essentially' the same yet structurally different then this automatic conversion can save the user a substantial amount of typing, time and effort.

Our functions also infer two coercions which are not invertible:

$$\mathsf{a} \;{:}{*}{:}\; \mathsf{b} \leqslant \mathsf{a} \qquad\qquad \mathsf{a} \leqslant \mathsf{a} \;{:}{+}{:}\; \mathsf{b}\ .$$

The canonical witnesses here are the first projection of a product and the left injection of a sum. Thanks to these reductions, the expression

$(\mathit{expand}\{\!|\mathsf{Either\ Bool\ Int}|\!\} \circ \mathit{reduce}\{\!|(\mathsf{Bool}, \mathsf{Int})|\!\})\ (\mathit{True}, 4)$

evaluates to *Left True*; note that it cannot evaluate to *Right* 4 because such a reduction would involve projecting a suffix and injecting into the right whereas we infer only prefix projections and left injections. Of course, we would prefer our theory to include the dual pair of coercions as well, but doing so would break the property that each pair of types determines a unique canonical witness. Despite this limitation, we will see in Section 6.6 how these coercions, when used with a cleverly laid out datatype, can be used to simulate single inheritance.

Now let us look at some examples which fail.

1. The conversion
   $\mathit{expand}\{\!|(\mathsf{Bool}, \mathsf{Int})|\!\} \circ \mathit{reduce}\{\!|(\mathsf{Int}, \mathsf{Bool})|\!\}$
   fails because our theory does not model commutativity of :*:.
2. The conversion
   $\mathit{expand}\{\!|\mathsf{Bool}|\!\} \circ \mathit{reduce}\{\!|\mathsf{Int}|\!\}$
   fails because the types are neither isomorphic nor coercible.
3. The conversion
   $\mathit{expand}\{\!|\mathsf{Bool}|\!\} \circ \mathit{reduce}\{\!|\mathsf{Either}\ ()\ ()|\!\}$
   fails because we chose to represent certain base types like Bool as "abstract": they are not destructured when reducing.

Currently, because our implementation depends on the "universal" type Univ, failure occurs at run-time and a message helpful for pinpointing the error's source is printed. In section 8, we discuss some possible future work which may provide static error detection.

6.5 Exploiting Isomorphisms

Datatypes produced by UUXML are unquestionably complicated. Let us consider instead what our ideal translation target might look like. Here is an obvious, very conventional, Haskell-style translation image of doc using records types:

```
module Doc where
data Doc     = Doc     { key     :: String,
                         authors :: [String],
                         title   :: String,
                         pubDate :: Maybe PubDate }
data PubDate = PubDate { year    :: Integer,
                         month   :: Integer }
```

Observe in particular that:

– the target types Doc and PubDate have conventional, Haskellish names which do not look machine-generated;
– the fields are typed by conventional Haskell datatypes like String, lists and Maybe;
– the attribute *key* is treated just like other elements; and
– intermediate 'wrapper' elements like *title* and *year* have been elided and do not generate new types;
– the positional information encoded in wrappers is available in the field projection names;
– the field name *authors* has been changed from the element name *author*, which is natural since *authors* projects a list whereas each *author* tag wraps a single author.

Achieving an analogous result in Java with a data binding like JAXB would require annotating (editing) the source schema directly, or writing a 'binding customization file' which is substantially longer than the two datatype declarations above. Both methods also require learning another XML vocabulary and some details of the translation process, and the latter uses XPath syntax to indicate the parts which require customization—a maintenance hazard since the schema structure may change.

With our iso inference system, provided that the document is known to be exactly of type `doc` and not a proper subtype, all that is required is the above Haskell declaration plus the following modest incantation.

$\mathit{expand}\{\!|\mathsf{Doc}|\!\} \circ \mathit{reduce}\{\!|\mathsf{E\_doc}|\!\}$

This expression denotes a function of type $\mathsf{E\_doc} \rightarrow \mathsf{Doc}$ which converts the unwieldy UUXML representation of `doc` into the idealized form above.

For example, the code in Figure 7 is a complete Generic Haskell program that reads in a `doc`-conforming document from standard input, deletes all authors named "De Sade", and writes the result to standard output.

```
module Censor where
import UUXML   -- our framework
import XDoc    -- automatically translated XML Schema
import Doc     -- the custom declarations Doc & PubDate

main     = interact work
work     = toE_doc ∘ censor ∘ toDoc
censor d = d{ authors = filter (≢ "De Sade") (authors d) }
toE_doc  = unparse{|E_doc|} ∘ expand{|E_doc|} ∘ reduce{|Doc|}
toDoc    = expand{|Doc|} ∘ reduce{|E_doc|} ∘ parse{|E_doc|}
```

Fig. 7. An example XML application.

6.6 The Role of Coercions

Recall that our system infers two non-invertible coercions:

$$\mathsf{a :\!*\!: b} \leqslant \mathsf{a} \qquad\qquad \mathsf{a} \leqslant \mathsf{a :\!+\!: b}\ .$$

Of course, this is only half the story we would like to hear! Though we could easily implement the dual pair of coercions, we cannot implement them both together except in an *ad hoc* fashion (and hence refrain from doing so). This limitation exists partly because, in reducing to a universal type, we have thrown away the type information. But, even if we knew the types involved, it is not clear, for example, whether $\mathsf{a} \rightarrow \mathsf{a :\!+\!: a}$ should be witnessed by the left or the right injection.

Fortunately, even this 'biased' form of subtyping proves quite useful. In particular, XML Schema's so-called 'extension' subtyping exactly matches the form of the first projection coercion, as it only allows documents validating against a type `t` to be used in contexts of type `s` if `s` matches a prefix of `t`: so `t` is an extension of `s`.

Schema's other form of subtyping, called 'restriction', allows documents validating against type `t` to be used in contexts of type `s` if every document validating against `t` also validates against `s`: so `t` is a restriction of `s`. This can only happen if `s`, regarded as a grammar, can be reformulated as a disjunction of productions, one of which is `t`, so it appears our left injection coercion can capture part of this subtyping relation as well.

Actually, due to a combination of circumstances, the situation is better than might be expected. First, subtyping in Schema is *manifest* or *nominal*, rather than purely *structural*: consequently, restriction only holds between types assigned a name in the schema. Second, our translation models subtyping by generating a Haskell datatype declaration for the down-closure of each named schema type. For example, the 'colored point' example familiar from the object-oriented literature would be expressed thus:

```
data Point     = Point ...
data CPoint    = CPoint ...
data LE_Point  = EQ_Point Point
               | LE_CPoint LE_CPoint
data LE_CPoint = EQ_CPoint CPoint
               | ...
```

Third, we have arranged our translator so that the *EQ_*... constructors always appear in the leftmost summand. This means that the injection from the 'equational' variant of a translated type to its down-closed variant is always the leftmost injection, and consequently picked out by our expansion mechanism.

```
EQ_Point  :: Point → LE_Point
EQ_CPoint :: CPoint → LE_CPoint
```

Since Haskell is, in itself, not well equipped for dealing with subtyping, when *reading* an XML document we would rather have the coercion the other way around, that is, we should like to read an LE_Point into a Point, but of course this is unsafe. However, when *writing* a value to a document these coercions save us some work inserting constructors.

Of course, since, unlike Schema itself, our coercion mechanism is structural, we can employ this capability in other ways. For instance, when writing a value to a document, we can use the fact that *Nothing* is the leftmost injection into the Maybe a type to omit optional elements.

7 Generic Isomorphisms

In this section, we describe how to automatically generate isomorphisms between pairs of datatypes.

We address the problem in four parts, treating first the product and sum isos in isolation, then showing how to merge those implementations. Finally, we describe a simple modification of the resulting program which implements the non-invertible coercions.

In each case, we build the requisite morphism by reducing a value $v :: \mathsf{t}$ to a value of a universal datatype $u = \mathit{reduce}\{\!|\mathsf{t}|\!\}\ v :: \mathsf{Univ}$. The type Univ plays the role of a normal form from which we can then expand to a value $\mathit{expand}\{\!|\mathsf{t}'|\!\}\ u :: \mathsf{t}'$ of the desired type, where $\mathsf{t} \leqslant \mathsf{t}'$ canonically, or $\mathsf{t} \cong \mathsf{t}'$ for the isos. Function *reduce* is similar to function *content* defined in Section 4.1, except that it returns a single value instead of a list, and the returned value contains information about its type, instead of being a string.

7.1 Handling Products

We define the functions $\mathit{reduce}\{\!|\mathsf{t}|\!\}$ and $\mathit{expand}\{\!|\mathsf{t}|\!\}$ which infer the isomorphisms expressing associativity and identities of binary products:

$$\mathsf{a :\!*\!: Unit} \cong \mathsf{a} \qquad \mathsf{Unit :\!*\!: a} \cong \mathsf{a} \qquad \mathsf{(a :\!*\!: b) :\!*\!: c} \cong \mathsf{a :\!*\!: (b :\!*\!: c)}\ .$$

We assume a set of base types, which may include integers, booleans, strings and so on. For brevity's sake, we mention only integers in our code.

```
    data UBase = UInt Int | UBool Bool | UString String | ···
```

The following two functions merely serve to convert back and forth between the larger world and our little universe of base types.

```
    reducebase{|t :: ⋆|}          :: t → UBase
    reducebase{|Int|} i           = UInt i
    expandbase{|t :: ⋆|}          :: UBase → t
    expandbase{|Int|} (UInt i) = i
```

Now, as Schemers well know, if we ignore the types and remove all occurrences of Unit, a right-associated tuple is simply a cons-list, hence our representation, Univ is defined as:

```
    type Univ = [UBase] .
```

Our implementation of $\mathit{reduce}\{\!|\mathsf{t}|\!\}$ depends on an auxiliary function $\mathit{red}\{\!|\mathsf{t}|\!\}$, which accepts a value of t along with an accumulating argument of type Univ; it returns the normal form of the t-value with respect to the laws above. The role of *reduce*{|t|} is just to prime *red*{|t|} with an empty list.

```
red{|t :: ⋆|}                  :: (red{|t|}) ⇒ t → Univ → Univ
red{|Int|}       i             = (reducebase{|Int|} i:)
red{|Unit|}      Unit          = id
red{|a :*: b|}   (a :*: b)     = red{|a|} a ∘ red{|b|} b

reduce{|t :: ⋆|}               :: (red{|t|}) ⇒ t → Univ
reduce{|t|} x                  = red{|t|} x [ ]
```

Here is an example of *reduce*{|t|} in action:

```
reduce{|((Int, (Int, Int)), ())|} ((2, (3, 4)), ()) = [UInt 2, UInt 3, UInt 4] .
```

Function *expand*{|t|} takes a value of the universal data type, and returns a value of type t. It depends on the generic function *len*{|t|}, which computes the length of a product, that is, the number of components of a tuple:

```
len{|t :: ⋆|}    :: (len{|t|}) ⇒ Int
len{|Int|}       = 1
len{|Unit|}      = 0
len{|a :*: b|}   = len{|a|} + len{|b|} .
```

Observe that the nullary product, Unit, is assigned length zero.

Now we can write *expand*{|t|}; like *reduce*{|t|}, it is defined in terms of a helper function *exp*{|t|}, this time in a dual fashion with the 'unparsed' remainder appearing as an output.

```
exp{|t :: ⋆|}          :: (exp{|t|}) ⇒ Univ → (t, Univ)
exp{|Int|} (u : us)    = (expandbase{|Int|} u, us)
exp{|Int|} [ ]         = error "exp"
exp{|Unit|} us         = (Unit, us)
exp{|a :*: b|} us      = let (u, us′)  = exp{|a|} us
                             (v, us″)  = exp{|b|} us′
                         in  (u :*: v, us″)

expand{|t :: ⋆|}       :: (exp{|t|}) ⇒ Univ → t
expand{|t|} u          = case exp{|t|} u of
                           (v, [ ]) → v
                           (v, _)   → error "expand"
```

In the last case of function *exp*, we compute the lengths of each factor of the product to determine how many values to project there—remember that a need not be a base type. This information tells us how to split the list between recursive calls.

Here is an example of *expand*{|t|} in action:

```
expand{|((Int, (Int, Int)), ())|} [UInt 2, UInt 3, UInt 4] = ((2, (3, 4)), ()) .
```

7.2 Handling Sums

We now turn to the treatment of associativity for sums:

```
(a :+: b) :+: c ≅ a :+: (b :+: c) .
```

As we will be handling sums alone in this section, we redefine the universal type as a right-associated sum of values:

```
data Univ = UInl UBase | UInr Univ .
```

Note that this datatype Univ can also be represented by[9]:

```
data Univ = UIn Integer UBase .
```

We prefer the latter representation as it simplifies some definitions. We also add a second integer field:

```
data Univ = UIn Integer Integer UBase .
```

If $u = UIn\ r\ a\ b$ then we call a the *arity* of u—it remembers the "width" of the sum value we reduced; we call r the *rank* of u—it denotes a zero-indexed position within the arity, the choice which was made. We guarantee, then, that $1 \leqslant r < a$.

Declarations UBase, *reducebase*{|t|} and *expandbase*{|t|} are as before.

This time around function *reduce*{|t|} represents values by right-associating sums. The examples below show some sample inputs and how they are reduced:

```
i             :: Int                  ↦ UIn 0 1 (UInt i)
Inl i         :: Int :+: Int          ↦ UIn 0 2 (UInt i)
Inr i         :: Int :+: Int          ↦ UIn 1 2 (UInt i)
Inl (Inl i)   :: (Int :+: Int) :+: Int ↦ UIn 0 3 (UInt i)
Inl (Inr i)   :: (Int :+: Int) :+: Int ↦ UIn 1 3 (UInt i)
Inr i         :: (Int :+: Int) :+: Int ↦ UIn 2 3 (UInt i) .
```

Function *reduce*{|t|} depends on the generic value *arity*{|t|}, which counts the number of choices in a sum.

```
arity{|t :: ⋆|}     :: (arity{|t|}) ⇒ Integer
arity{|Int|}        = 1
arity{|a :+: b|}    = arity{|a|} + arity{|b|}
```

Now we can define *reduce*{|t|}:

```
reduce{|t :: ⋆|}             :: (arity{|t|}, reduce{|t|}) ⇒ t → Univ
reduce{|Int|}      i          = UIn 0 1 (reducebase{|Int|} i)
reduce{|a :+: b|} (Inl x)    = UIn r (a + arity{|b|}) u
                                  where UIn r a u = reduce{|a|} x
reduce{|a :+: b|} (Inr x)    = UIn (r + arity{|a|}) (arity{|a|} + a) u
                                  where UIn r a u = reduce{|b|} x .
```

This treats base types as unary sums, and computes the rank of a value by examining the arities of each summand, effectively flattening the sum.

The function *expand*{|t|} is defined as follows:

```
expand{|t :: ⋆|}              :: (arity{|t|}, expand{|t|}) ⇒ Univ → t
expand{|Int|} (UIn 0 1 u)     = expandbase{|Int|} u
expand{|a :+: b|} (UIn r a u)
    | a ≡ aa + ab ∧ r < aa    = Inl (expand{|a|} (UIn r (a − ab) u))
    | a ≡ aa + ab             = Inr (expand{|b|} (UIn (r − aa) (a − aa) u))
    | otherwise               = error "expand"
  where (aa, ab) = (arity{|a|}, arity{|b|}) .
```

[9] The type Integer is unbounded, while Int is a "machine integer".

The logic in the last case checks that the universal value 'fits' in the sum type a :+: b, and injects it into the appropriate summand by comparing the value's rank with the arity of a, being sure to adjust the rank and arity on recursive calls.

7.3 Sums and Products Together

It may seem that a difficulty in handling sums and products simultaneously arises in designing the type Univ, as a naïve amalgamation of the sum Univ (call it UnivS) and the product Univ (call it UnivP) permits multiple representations of values identified by the canonical isomorphism relation. However, since the rules of our isomorphism theory do not interact—in particular, we do not account for any sort of distributivity—, a simpler solution exists: we can nest our two representations and add the top layer as a new base type. For example, we can use UnivP in place of UBase in UnivS and add a new constructor to UBase to encapsulate sums.

```
data UnivS  = UIn Integer Integer UnivP
data UnivP  = UNil | UCons UBase UnivP
data UBase = UInt Int | USum UnivS
```

We omit the details, as the changes to our code examples are straightforward.

7.4 Handling Coercions

The reader may already have noticed that our expansion functions impose some unnecessary limitations. In particular:

- when we expand to a product, we require that the length of our universal value equals the number computed by *len*{|t|}, and
- when we expand to a sum, we require that the arity of our universal value equals the number computed by *arity*{|t|}.

If we lift these restrictions, replacing equality by inequality, we can project a prefix of a universal value onto a tuple of smaller length, and inject a universal value into a choice of larger arity. The modified definitions are shown below for products:

```
expand{|t|} u = case exp{|t|} u of
                    (v, _) → v
```

and for sums:

```
expand{|a :+: b|} (UIn r a u)
    | a ⩽ aa + ab ∧ r < aa = Inl (expand{|a|} (UIn r (a − ab) u))
    | a ⩽ aa + ab          = Inr (expand{|b|} (UIn (r − aa) (a − aa) u))
    | otherwise            = error "expand"
  where (aa, ab) = (arity{|a|}, arity{|b|}) .
```

These changes implement our canonical coercions, the first projection of a product and left injection of a sum:

$$\mathsf{a} :\!*\!: \mathsf{b} \leqslant \mathsf{a} \qquad\qquad \mathsf{a} \leqslant \mathsf{a} :\!+\!: \mathsf{b} \ .$$

Ad hoc coercions can be handled using our approach as well. Many conventional languages define a subtyping relation between primitive types. For example, in XML Schema `int` (bounded integers) is a subtype of `integer` (unbounded integers) which is a subtype of `decimal` (reals representable by decimal numerals) [49]. We can easily model this by (adding some more base types and) modifying the functions which convert base types.

$$
\begin{array}{lll}
\mathit{expandbase}\{|\mathsf{Decimal}|\} & (\mathit{UDecimal}\ x) & = x \\
\mathit{expandbase}\{|\mathsf{Decimal}|\} & (\mathit{UInteger}\ x) & = \mathit{integer2dec}\ x \\
\mathit{expandbase}\{|\mathsf{Decimal}|\} & (\mathit{UInt}\ x) & = \mathit{int2dec}\ x \\
\mathit{expandbase}\{|\mathsf{Integer}|\} & (\mathit{UInteger}\ x) & = x \\
\mathit{expandbase}\{|\mathsf{Integer}|\} & (\mathit{UInt}\ x) & = \mathit{int2integer}\ x \\
\mathit{expandbase}\{|\mathsf{Int}|\} & (\mathit{UInt}\ x) & = x
\end{array}
$$

Such primitive coercions are easy to handle, but without due care are likely to break the coherence properties of inference, so that the inferred coercion depends on operational details of the inference algorithm.

7.5 Conclusion

Let us summarize the main points of this case study.

We demonstrated first by example that UUXML-translated datatypes are overwhelmingly complex and redundant. To address complaints that this problem stems merely from a bad choice of representation, we enumerated some of UUXML's design criteria, and explained why they necessitate that representation. We also suggested why other translations and type-safe embeddings might suffer from the same problem. Finally, we described how to exploit our iso inference mechanism to address this problem, and how coercion inference can also be used to simplify the treatment of object language features such as subtyping and optional values which the metalanguage does not inherently support.

8 Conclusions

This paper describes:

– a simple and powerful mechanism for automatically inferring a well-behaved class of isomorphisms.
– UUXML, an XML Schema–Haskell data binding. 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 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.
– how the automatic inference of isomorphisms solves some usability problems stemming from the complexity of UUXML.

Our inference mechanism leverages the power of an existing tool, Generic Haskell, and makes some use of the established theory of type isomorphisms. UUXML uses Generic Haskell in its generic parser.

We believe that both the general idea of exploiting isomorphisms and our implementation technique have application beyond UUXML. For example, when libraries written by distinct developers are used in the same application, they often include different representations of what amounts to the same datatype. When passing data from one library to the other the data must be converted to conform to each library's internal conventions. Our technique could be used to simplify this conversion task; to make this sort of application practical, though, iso inference should probably be integrated with type inference, and the class of isos inferred should be enlarged. We discuss such possibilities for future work below.

8.1 Related & Future Work

Isomorphisms and coherence. In computer science, the use of type isomorphisms seem to have been popularized first by Rittri who demonstrated their value in software retrieval tasks, such as searching a software library for functions matching a query type [38]. Since then the area has ballooned; good places to start on the theory of type isomorphisms is Di Cosmo's book [12] and the paper by Bruce et al. [10]. More recent work has focused on linear type isomorphisms [6, 41, 39, 28].

In category theory, Mac Lane initiated the study of coherence in a seminal paper [26]; his book [27] treats the case for monoidal categories. Beylin and Dybjer's use [8] of Mac Lane's coherence theorem influenced our technique here. The strong regularity condition is sufficient for ensuring that an algebraic theory is *cartesian*; cartesian monads have been used by Leinster [22, 21] and Hermida [15] to formalize the notion of generalized multicategory, which generalizes a usual category by imposing an algebraic theory on the objects, and letting the domain of an arrow be a term of that theory.

Schema matching. In areas like database management and electronic commerce, the plethora of data representation standards—formally, 'schemas'—used to transmit and store data can hinder reuse and data exchange. To deal with this growing problem, 'schema matching', the problem of how to construct a mapping between elements of two schemas, has become an active research area. Because the size, complexity and number of schemas is only increasing, finding ways to accurately and efficiently automate this task has become more and more important; see Rahm and Bernstein [37] for a survey of approaches.

Erwig [13] has suggested a technique for automatic schema matching similar to ours in the sense that it exploits 'information-preserving and -approximating' functions between DTDs. Although that approach can automatically infer some more sophisticated transformations than ours (such as one related to the sum-product distributivity iso), it is based on a home-grown semantics for XML

DTDs, not W3C Schemas, and does not guarantee that the transformations are canonical, thus requiring some interaction (with a user or external data source) to select a mapping appropriate to the task at hand.

We believe that our approach, which exploits not only the syntax but semantics of types, could provide new insights into schema matching. In particular, the notion of canonical (iso)morphism could help clarify when a mapping's semantics is forced entirely by structural considerations, and when additional information (linguistic, descriptive, *etc.*) is provably required to disambiguate a mapping.

Implicit coercions. Thatte introduced a declaration construct for introducing user-defined, *implicit* conversions between types [43], using, like us, an equational theory on types. Thatte also presents a principal type inference algorithm for his language, which requires that the equational theory is *unitary*, that is, every unifiable pair of types has a unique most general unifier. To ensure theories be unitary, Thatte demands they be *finite* and *acyclic*, and uses a syntactic condition related to, but different from, strong regularity to ensure finiteness. In Thatte's system, coherence seems to hold if and only if the user-supplied conversions are true inverses.

The relationship between Thatte's system and ours requires further investigation. In some ways Thatte's system is more liberal, allowing for example distributive theories. On the other hand, the unitariness requirement rules out associative theories, which are infinitary. The acyclicity condition also rules out commutative theories, which are not strongly regular, but also the currying iso, which is. Another difference between Thatte's system and ours is that his catches errors at compile-time, while the implementation we presented here does so at run-time. A final difference is that, although the finite acyclicity condition is decidable, the requirement that conversions be invertible is not; consequently, users may introduce declarations which break the coherence property (produce ambiguous programs). In our system, any user-defined conversions are obtained structurally, as datatype isos from datatype declarations, which cannot fail to be canonical; hence it is not possible to break coherence.

Inference failure. Because our implementation depends on the "universal" type Univ, failure occurs dynamically and a message helpful for pinpointing the error's source is printed. This situation is unsatisfactory, though, since every invocation of the $\mathit{expand}\{\!|\cdot|\!\}$ and $\mathit{reduce}\{\!|\cdot|\!\}$ functions together mentions the types involved; in principle, we could detect failures statically, thus increasing program reliability.

Such early detection could also enable new optimizations. For example, if the types involved are not only isomorphic but equal, then the conversion is the identity and a compiler could omit it altogether. But even if the types are only isomorphic, the reduction might not unreasonably be done at compile-time, as our isos are all known to be terminating; this just amounts to adjusting the data representation 'at one end' or the other to match exactly.

We have investigated, but not tested, an approach for static failure detection based on an extension of Generic Haskell's *type-indexed datatypes* [17]. The idea is to introduce a type-indexed datatype $\mathsf{NF}\{\![\mathsf{t}]\!\}$ which denotes the normal form of type t w.r.t. to the iso theory, and then reformulate our functions so that they are assigned types:

$$\begin{aligned} &\mathit{reduce}\{\!|\mathsf{t}|\!\} \;\; :: \mathsf{t} \to \mathsf{NF}\{\![\mathsf{t}]\!\} \\ &\mathit{expand}\{\!|\mathsf{t}|\!\} :: \mathsf{NF}\{\![\mathsf{t}]\!\} \to \mathsf{t}\ . \end{aligned}$$

For example, considering only products, the type $\mathsf{NF}\{\![\mathsf{t}]\!\}$ could be defined as follows.

$$\begin{aligned} &\mathbf{type}\ \mathsf{NF}\{\![\mathsf{t}]\!\} &&= \mathsf{Norm}\{\![\mathsf{t}]\!\}\ \mathsf{Unit} \\ &\mathbf{data}\ \mathsf{Norm}\{\![\mathsf{Unit}]\!\}\ \mathsf{t} &&= \mathit{NUnit}\ \mathsf{t} \\ &\mathbf{data}\ \mathsf{Norm}\{\![\mathsf{a} :\!*\!: \mathsf{b}]\!\}\ \mathsf{t} &&= \mathit{NProd}\ (\mathsf{a} :\!*\!: (\mathsf{b} :\!*\!: \mathsf{t})) \\ &\mathbf{data}\ \mathsf{Norm}\{\![\mathsf{Int}]\!\}\quad \mathsf{t} &&= \mathit{NBase}\ (\mathsf{Int} :\!*\!: \mathsf{t}) \end{aligned}$$

This would give the GH compiler enough information to reject bad conversions at compile-time.

Unfortunately, the semantics of GH's type-indexed datatypes is too "generative" for this approach to work. The problem is apparent if we try to compile the expression:

$$\mathit{expand}\{\!|\mathsf{Int}|\!\} \circ \mathit{reduce}\{\!|(\mathsf{Int}, ())|\!\}\ .$$

GH flags this as a type error, because it treats $\mathsf{NF}\{\![\mathsf{Int}]\!\}$ and $\mathsf{NF}\{\![(\mathsf{Int}, ())]\!\}$ as distinct (unequal), though structurally identical, datatypes.

A possible solution to this issue may be a recently considered GH extension called *type-indexed types* (as opposed to *type-indexed datatypes*). If $\mathsf{NF}\{\![t]\!\}$ is implemented as a type-indexed type, then, like Haskell's type synonyms, structurally identical instances like the ones above will actually be forced to be equal, and the expression above should compile. However, type-indexed types—as currently envisioned—also share the limitations of Haskell's type synonyms w.r.t. recursion; a type-indexed type like $\mathsf{NF}\{\![\mathsf{List\ Int}]\!\}$ is likely to cause the compiler to loop as it tries to expand recursive occurrences while traversing the datatype body. Nevertheless, of the several approaches we have considered to addressing the problem of static error detection, type-indexed types seems the most promising.

Acknowledgements. The authors thank Dave Clarke, Bastiaan Heeren, Andres Löh, Eelco Dolstra, Fermin Reig and Alexey Rodriguez Yakushev and several anonymous referees for comments on various versions of this article. Tom Leinster kindly clarified some of our questions about cartesian monads and the strong regularity condition. Wouter Swierstra helped investigate approaches to static detection of inference failure. This research has been funded by the Netherlands Organisation for Scientific Research (NWO) (project Mondrian, nr. 612.060.007).

References

1. Frank Atanassow, Dave Clarke, and Johan Jeuring. Scripting XML with Generic Haskell. Technical Report UU-CS-2003-023, Utrecht University, 2003.

2. Frank Atanassow, Dave Clarke, and Johan Jeuring. Scripting XML with Generic Haskell. In *Proc. 7th Brazilian Symposium on Programming Languages*, 2003.
3. Frank Atanassow, Dave Clarke, and Johan Jeuring. UUXML: A type-preserving XML Schema-Haskell data binding. In Bharat Jayaraman, editor, *Proceedings 6th International Symposium on Practical Aspects of Declarative Languages, PADL'04*, volume 3057 of *LNCS*, pages 71–85. Springer-Verlag, 2004.
4. Frank Atanassow and Johan Jeuring. Inferring type isomorphisms generically. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC'04*, volume 3125 of *LNCS*, pages 32–53. Springer-Verlag, 2004.
5. 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.
6. Vincent Balat and Roberto Di Cosmo. A linear logical view of linear type isomorphisms. In *CSL*, pages 250–265, 1999.
7. V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of 2003 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2003.
8. Ilya Beylin and Peter Dybjer. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In *TYPES*, pages 47–61, 1995.
9. Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler. MSL: A model for W3C XML Schema. In *Proc. WWW10*, May 2001.
10. Kim B. Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.
11. Jorge Coelho and Mário Florido. Type-based XML processing in logic programming. In *PADL 2003*, pages 273–285, 2003.
12. Roberto Di Cosmo. *Isomorphisms of Types: From lambda-calculus to Information Retrieval and Language Design*. Birkhäuser, 1995.
13. Martin Erwig. Toward the automatic derivation of XML transformations. In *1st Int. Workshop on XML Schema and Data Management (XSDM '03)*, volume 2814 of *LNCS*, pages 342–354, 2003.
14. Vladimir Gapeyev and Benjamin C. Pierce. Regular object types. In *European Conference on Object-oriented Programming (ECOOP 2003)*, 2003.
15. C. Hermida. Representable multicategories. *Advances in Mathematics*, 151:164–225, 2000.
16. Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory. In *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 1–56. Springer-Verlag, 2003.
17. 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.
18. 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.
19. Graham Hutton and Erik Meijer. Monadic parser combinators. *Journal of Functional Programming*, 8(4):437–444, 1996.
20. Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Second USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, 1999. USENIX Association. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).

21. Thomas S.H. Leinster. *Operads in Higher-Dimensional Category Theory.* PhD thesis, Trinity College and St John's College, Cambridge, 2000.
22. Tom Leinster. *Higher Operads, Higher Categories.* Cambridge University Press, 2003.
23. Xavier Leroy et al. *The Objective Caml system release 3.07, Documentation and user's manual*, December 2003. Available from `http://caml.inria.fr/ocaml/htmlman/`.
24. Andres Löh. *Exploring Generic Haskell.* PhD thesis, Utrecht University, 2004.
25. Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In *Proceedings of the International Conference on Functional Programming (ICFP '03)*, August 2003.
26. Saunders Mac Lane. Natural associativity and commutativity. *Rice University Studies*, 49:28–46, 1963.
27. Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics.* Springer-Verlag, New York, 2nd edition, 1997. (1st ed., 1971).
28. Bruce McAdam. How to repair type errors automatically. In *Trends in Functional Programming (Proc. Scottish Functional Programming Workshop)*, volume 3, 2001.
29. Erik Meijer and Mark Shields. XMLambda: A functional language for constructing and manipulating XML documents. Available from `http://www.cse.ogi.edu/~mbs/`, 1999.
30. Eldon Metz and Allen Brookes. XML data binding. *Dr. Dobb's Journal*, pages 26–36, March 2003.
31. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, May 1997.
32. OASIS. RELAX NG. `http://www.relaxng.org`, 2001.
33. Uche Ogbuji. Xml data bindings in python, parts 1 & 2. *xml.com*, 2003. `http://www.xml.com/pub/a/2003/06/11/py-xml.html`.
34. Uche Ogbuji. EaseXML: A Python data-binding tool. *xml.com*, 2005. `http://www.xml.com/pub/a/2003/06/11/py-xml.html`.
35. Simon Peyton Jones et al. *Haskell 98, Language and Libraries. The Revised Report.* Cambridge University Press, 2003. A special issue of the Journal of Functional Programming.
36. Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.
37. Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal: Very Large Data Bases*, 10(4):334–350, 2001.
38. Mikael Rittri. Retrieving library identifiers via equational matching of types. In *Conference on Automated Deduction*, pages 603–617, 1990.
39. Mikael Rittri. Retrieving library functions by unifying types modulo linear isomorphism. *Informatique Theorique et Applications*, 27(6):523–540, 1993.
40. 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/`.
41. Sergei Soloviev. A complete axiom system for isomorphism of types in closed categories. In A. Voronkov, editor, *Proceedings 4th Int. Conf. on Logic Programming and Automated Reasoning, LPAR'93, St. Petersburg, Russia, 13–20 July 1993*, volume 698, pages 360–371. Springer-Verlag, Berlin, 1993.
42. Sun Microsystems. Java Architecture for XML Binding (JAXB). `http://java.sun.com/xml/jaxb/`, 2003.

43. Satish R. Thatte. Coercive type isomorphism. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, volume 523 of *LNCS*, pages 29–49. Springer-Verlag New York, Inc., 1991.
44. Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4&5):435–468, July 2002.
45. W3C. XML 1.0. `http://www.w3.org/XML/`, 1998.
46. W3C. XSL Transformations 1.0. `http://www.w3.org/TR/xslt`, 1999.
47. W3C. XML Schema part 0: Primer. `http://www.w3.org/TR/xmlschema-0`, 2001.
48. W3C. XML Schema part 1: Structures. `http://www.w3.org/TR/xmlschema-1`, 2001.
49. W3C. XML Schema part 2: Datatypes. `http://www.w3.org/TR/xmlschema-2`, 2001.
50. W3C. XML information set (second edition). `http://www.w3.org/TR/xml-infoset/`, 2004.
51. W3C. XML Schema: Formal description. `http://www.w3.org/TR/xmlschema-formal`, 2005.
52. W3C. XQuery 1.0: An XML query language. `http://www.w3.org/TR/xquery/`, 2005.
53. Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *International Conference on Functional Programming*, pages 148–159, 1999.