Inferring Type Isomorphisms Generically — With an application to an XML Schema–Haskell data binding

Frank Atanassow

Johan Jeuring

institute of information and computing sciences,
utrecht university

technical report UU-CS-2005-008

www.cs.uu.nl

1 Introduction

Typed functional languages like Haskell [35] and ML [24,33] 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 Date2,

```
data Date2 = Date2 (Int, Int, 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:

```
data Date3 = Date3 (Int, (Int, Int))
data Date4 = Date4 ((Int, Int), Int)
data Date5 = Date5 (Int, (Int, (Int, ()))) .
```

Though these types all represent the same abstract data structure[1], 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. In this case, the amount of code required to define such a conversion function is not so large, but if the declarations are machine-generated, or the number

[1] We will assume all datatypes are strict; otherwise, Haskell's non-strict semantics typically entails that some transformations like nesting add a new value $\perp$ which renders this claim false.

of representations to be simultaneously supported is large, then the size of the conversion code can become unmanageable. In this paper we show how to infer such conversions automatically.

1.1 Isomorphisms

The fact that all these types represent the same abstract type is captured by the notion of *isomorphism*: two types are isomorphic if there exists an invertible function between them, our desired conversion function. Besides invertibility, two basic facts about isomorphisms (isos, for short) are: the identity function is an iso, so every type is isomorphic to itself; and, the composition of two isos is an iso. Considered as a relation, then, isomorphism is an equivalence on types.

Other familiar isos are a consequence of the semantics of base types. For example, the conversion between meters and miles is a non-identity iso between the floating point type Double and itself; if we preserve the origin, the conversion between cartesian and polar coordinates is another example. Finally, some polymorphic isos arise from the structure of types themselves; for example, one often hears that products are associative "up to isomorphism".

It is the last sort, often called *canonical* or *coherence* (iso)morphisms, which are of chief interest to us. Canonical isos are special because they are uniquely determined by the types involved, that is, there is at most one canonical iso between two polymorphic type schemes.

1.1.1 Monoidal isos.

Let Unit and :*: be nullary and binary product constructors, respectively, and Zero and :+: nullary and binary sum constructors, defined as the following datatypes.

```
data Unit     = Unit
data a :*: b = a :*: b
data Zero
data a :+: b = Inl a | Inr b
```

Haskell 98 doesn't support infix type constructors such as :+: and :*:, but we will use these constructors in this paper because it makes type equations easier to read[2]. A few canonical isos of Haskell are summarized by the syntactic

[2] In the Haskell and Generic Haskell source code used to implement the functions defined in this paper we use Sum instead of :+:, and Prod instead of :*:

theory below.

$$\begin{array}{lll} \mathsf{a :\!*\!: Unit} \cong \mathsf{a} & \mathsf{Unit :\!*\!: a} \cong \mathsf{a} & \mathsf{(a :\!*\!: b) :\!*\!: c} \cong \mathsf{a :\!*\!: (b :\!*\!: c)} \\ \mathsf{a :\!+\!: Zero} \cong \mathsf{a} & \mathsf{Zero :\!+\!: a} \cong \mathsf{a} & \mathsf{(a :\!+\!: b) :\!+\!: c} \cong \mathsf{a :\!+\!: (b :\!+\!: c)} \end{array}$$

The isomorphisms which witness these identities are the evident ones. The first two identities in each row express the fact that Unit (resp. Zero) is a right and left unit for :∗: (resp. :+:); the last says 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 canonically commutative:

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

and the currying and the distributivity isos are also canonical:

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

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:

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

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 [23,22,15]. 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; we have not yet attempted to treat the currying iso.

1.1.2 Datatype isos.

In Generic Haskell each datatype declaration effectively induces a canonical iso between the datatype and its underlying "structure type". For example, the declaration

$$\textbf{data}\ \mathsf{List\ a} = \mathit{Nil} \mid \mathit{Cons}\ \mathsf{a}\ (\mathsf{List\ a})$$

induces a canonical isomorphism

$$\mathsf{List\ A} \cong \mathsf{Unit}\ {:}{+}{:}\ (\mathsf{A}\ {:}{*}{:}\ \mathsf{List\ A})\ .$$

for each instantiation type A. We call such isos *datatype isos.*

Note that datatype isos are always strongly regular, as they are identities between pairs of constants. The type parameter in the Haskell declaration does *not* play the role of a variable in the iso theory: our system will not infer that $\mathsf{List\ A}$ is isomorphic to $\mathsf{List\ B}$ even if $\mathsf{A} \cong \mathsf{B}$. However, largely as a side effect of the way Generic Haskell works, it *does* infer the isomorphism between a datatype and its underlying structure type.

1.2 Applying isomorphisms

The ideas of this paper grew out of our attempts to improve UUXML, our XML Schema–Haskell data binding [3]. We think that data bindings are an important application area for inferring type isomorphisms generically. In this paper we will introduce UUXML, a type-preserving XML Schema–Haskell data binding, and we will show how generic isomorphisms can be used to improve this data binding.

1.3 Outline

This paper is an extended version of our paper about inferring type isomorphisms generically [4] presented at the Mathematics of Program Construction Conference in 2004, and it uses much of the material of our paper UUXML: a type-preserving XML Schema–Haskell data binding [3] presented at the Practical Aspects of Declarative Languages Conference in 2004. Furthermore, we have included an introduction to generic programming in Generic Haskell.

The remainder of this article is organized as follows. Section 2 introduces generic programming in Generic Haskell. Section 3 informally describes the user interface to our inference mechanism. Section 4 shows how our iso inferencer is implemented in Generic Haskell. Section 5 introduces UUXML, a type-preserving XML Schema–Haskell data binding. Section 6 shows how to use iso inference to automatically customize this data binding. Finally, Section 7 summarizes our results, and discusses related work and possibilities for future work in this area.

2 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 [16,25].

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 [16] 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* [26,25]. Dependencies both simplify and increase the expressiveness of generic programming.

In order to be able 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 right-associatively if there are more than two constructors), and a sequence of arguments of a constructor by a product, denoted by :*: (nested right-associatively if there are more than two arguments). These structure types have been defined in Section 1. A nullary constructor is replaced by the structure type Unit. The arguments of the constructors are not translated. Section 1 gives the structure type of the datatype List; here 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 *canonically 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.

```
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 this definition:

- 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 recursion 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*.

3 Inferring Isomorphisms

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

```
reduce{|t|}   :: t → Univ
expand{|t'|}  :: Univ → t' .
```

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

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

If $\mathsf{t} = \mathsf{t}'$, then $expand\{\!|\mathsf{t}'|\!\}$ and $reduce\{\!|\mathsf{t}|\!\}$ are mutual inverses. If t and t' are merely isomorphic, then expansion *may* fail; it always 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

$$\begin{array}{l}(expand\{\!|(\mathsf{Bool}, \mathsf{Bool}\ \mathsf{:+:}\ (\mathsf{Int}\ \mathsf{:+:}\ \mathsf{String}))|\!\} \circ \\ \quad reduce\{\!|(\mathsf{Bool}, ((), (\mathsf{Bool}\ \mathsf{:+:}\ \mathsf{Int})\ \mathsf{:+:}\ \mathsf{String}))|\!\}) \\ \qquad (True, ((), Inl\ (Inr\ 7)))\ ,\end{array}$$

which evaluates to

$$(True, Inr\ (Inl\ 7))\ .$$

Function $reduce\{\!|\mathsf{t}|\!\}$ 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 $expand\{\!|\mathsf{t}'|\!\}$ 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:

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

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

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

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 6.4 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 7, we discuss some possible future work which may provide static error detection.

4 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::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 2, 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.

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

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

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.

$$\mathbf{data}\ \mathsf{UBase} = \mathit{UInt}\ \mathsf{Int} \mid \mathit{UBool}\ \mathsf{Bool} \mid \mathit{UString}\ \mathsf{String} \mid \cdots$$

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

$$\begin{array}{ll} \mathit{reducebase}\{|\mathsf{t} :: \star|\} & :: \mathsf{t} \to \mathsf{UBase} \\ \mathit{reducebase}\{|\mathsf{Int}|\}\ i & = \mathit{UInt}\ i \\ \mathit{expandbase}\{|\mathsf{t} :: \star|\} & :: \mathsf{UBase} \to \mathsf{t} \\ \mathit{expandbase}\{|\mathsf{Int}|\}\ (\mathit{UInt}\ i) & = i \end{array}$$

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:

$$\mathbf{type}\ \mathsf{Univ} = [\mathsf{UBase}]\ .$$

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 $\mathit{reduce}\{|\mathsf{t}|\}$ is just to prime $\mathit{red}\{|\mathsf{t}|\}$ with an empty list.

$$\begin{array}{lll} \mathit{red}\{|\mathsf{t} :: \star|\} & & :: (\mathit{red}\{|\mathsf{t}|\}) \Rightarrow \mathsf{t} \to \mathsf{Univ} \to \mathsf{Univ} \\ \mathit{red}\{|\mathsf{Int}|\} & i & = (\mathit{reducebase}\{|\mathsf{Int}|\}\ i{:}) \\ \mathit{red}\{|\mathsf{Unit}|\} & \mathit{Unit} & = \mathit{id} \\ \mathit{red}\{|\mathsf{a} :\!*\!: \mathsf{b}|\} & (a :\!*\!: b) & = \mathit{red}\{|\mathsf{a}|\}\ a \circ \mathit{red}\{|\mathsf{b}|\}\ b \\ \mathit{reduce}\{|\mathsf{t} :: \star|\} & & :: (\mathit{red}\{|\mathsf{t}|\}) \Rightarrow \mathsf{t} \to \mathsf{Univ} \\ \mathit{reduce}\{|\mathsf{t}|\}\ x & & = \mathit{red}\{|\mathsf{t}|\}\ x\ [\,] \end{array}$$

Here is an example of $\mathit{reduce}\{|\mathsf{t}|\}$ in action:

$$\mathit{reduce}\{|((\mathsf{Int}, (\mathsf{Int}, \mathsf{Int})), ())|\}\ ((2, (3, 4)), ()) = [\,\mathit{UInt}\ 2, \mathit{UInt}\ 3, \mathit{UInt}\ 4]\ .$$

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

$$\begin{aligned}
&\mathit{len}\{\![\mathsf{t} :: \star]\!\} && :: (\mathit{len}\{\![\mathsf{t}]\!\}) \Rightarrow \mathsf{Int} \\
&\mathit{len}\{\![\mathsf{Int}]\!\} && = 1 \\
&\mathit{len}\{\![\mathsf{Unit}]\!\} && = 0 \\
&\mathit{len}\{\![\mathsf{a} :\!*\!: \mathsf{b}]\!\} && = \mathit{len}\{\![\mathsf{a}]\!\} + \mathit{len}\{\![\mathsf{b}]\!\}\ .
\end{aligned}$$

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:

$$\mathit{expand}\{\![((\mathsf{Int},(\mathsf{Int},\mathsf{Int})),())]\!\}\ [\mathit{UInt}\ 2, \mathit{UInt}\ 3, \mathit{UInt}\ 4] = ((2,(3,4)),())\ .$$

4.2 Handling Sums

We now turn to the treatment of associativity and identity laws for sums:

$$\mathsf{a} :\!+\!: \mathsf{Zero} \cong \mathsf{a} \qquad \mathsf{Zero} :\!+\!: \mathsf{a} \cong \mathsf{a} \qquad (\mathsf{a} :\!+\!: \mathsf{b}) :\!+\!: \mathsf{c} \cong \mathsf{a} :\!+\!: (\mathsf{b} :\!+\!: \mathsf{c})\ .$$

We can implement Zero as an abstract type with no (visible) constructors:

```
data Zero .
```

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

$$\textbf{data}\ \mathsf{Univ} = \mathit{UInl}\ \mathsf{UBase} \mid \mathit{UInr}\ \mathsf{Univ}\ .$$

Note that this datatype Univ is isomorphic to:

$$\textbf{data}\ \mathsf{Univ} = \mathit{UIn}\ \mathsf{Int}\ \mathsf{UBase}\ .$$

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

$$\textbf{data}\ \mathsf{Univ} = \mathit{UIn}\ \mathsf{Int}\ \mathsf{Int}\ \mathsf{UBase}\ .$$

If $u = \mathit{UIn}\ r\ a\ b$ 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 $0 \leqslant r < a$. Of course, unlike Unit, Zero has no observable values so there is no representation for it in Univ.

Declarations UBase, $\mathit{reducebase}\{\!|\mathsf{t}|\!\}$ and $\mathit{expandbase}\{\!|\mathsf{t}|\!\}$ are as before.

This time around function $\mathit{reduce}\{\!|\mathsf{t}|\!\}$ represents values by right-associating sums and ignoring choices against Zero. The examples below show some sample inputs and how they are reduced (we write I for Int and u for $\mathit{UInt}\ i$):

$$\begin{array}{lll}
i & :: \mathsf{I} & \mapsto \mathit{UIn}\ 0\ 1\ u \\
\mathit{Inl}\ i & :: \mathsf{I}\ {:}{+}{:}\ \mathsf{Zero} & \mapsto \mathit{UIn}\ 0\ 1\ u \\
\mathit{Inr}\ i & :: \mathsf{Zero}\ {:}{+}{:}\ \mathsf{I} & \mapsto \mathit{UIn}\ 0\ 1\ u \\
\mathit{Inl}\ i & :: \mathsf{I}\ {:}{+}{:}\ \mathsf{I} & \mapsto \mathit{UIn}\ 0\ 2\ u \\
\mathit{Inr}\ i & :: \mathsf{I}\ {:}{+}{:}\ \mathsf{I} & \mapsto \mathit{UIn}\ 1\ 2\ u \\
\mathit{Inl}\ (\mathit{Inl}\ i) & :: (\mathsf{I}\ {:}{+}{:}\ \mathsf{I})\ {:}{+}{:}\ \mathsf{I} & \mapsto \mathit{UIn}\ 0\ 3\ u \\
\mathit{Inl}\ (\mathit{Inr}\ i) & :: (\mathsf{I}\ {:}{+}{:}\ \mathsf{I})\ {:}{+}{:}\ \mathsf{I} & \mapsto \mathit{UIn}\ 1\ 3\ u \\
\mathit{Inr}\ i & :: (\mathsf{I}\ {:}{+}{:}\ \mathsf{I})\ {:}{+}{:}\ \mathsf{I} & \mapsto \mathit{UIn}\ 2\ 3\ u\ .
\end{array}$$

Function $\mathit{reduce}\{\!|\mathsf{t}|\!\}$ depends on the generic value $\mathit{arity}\{\!|\mathsf{t}|\!\}$, which counts the number of choices in a sum.

$$\begin{array}{ll}
\mathit{arity}\{\!|\mathsf{t} :: \star|\!\} & :: (\mathit{arity}\{\!|\mathsf{t}|\!\}) \Rightarrow \mathsf{Int} \\
\mathit{arity}\{\!|\mathsf{Int}|\!\} & = 1 \\
\mathit{arity}\{\!|\mathsf{Zero}|\!\} & = 0 \\
\mathit{arity}\{\!|\mathsf{a}\ {:}{+}{:}\ \mathsf{b}|\!\} & = \mathit{arity}\{\!|\mathsf{a}|\!\} + \mathit{arity}\{\!|\mathsf{b}|\!\}
\end{array}$$

Now we can define $\mathit{reduce}\{\!|\mathsf{t}|\!\}$:

$$
\begin{array}{lll}
\mathit{reduce}\{\!|\mathsf{t} :: \star|\!\} & & :: (\mathit{arity}\{\!|\mathsf{t}|\!\}, \mathit{reduce}\{\!|\mathsf{t}|\!\}) \Rightarrow \mathsf{t} \to \mathsf{Univ} \\
\mathit{reduce}\{\!|\mathsf{Int}|\!\} & i & = \mathit{UIn}\ 0\ 1\ (\mathit{reducebase}\{\!|\mathsf{Int}|\!\}\ i) \\
\mathit{reduce}\{\!|\mathsf{Zero}|\!\} & \_ & = \bot \\
\mathit{reduce}\{\!|\mathsf{a :\!+\!: b}|\!\} & (\mathit{Inl}\ x) & = \mathit{UIn}\ r\ (a + \mathit{arity}\{\!|\mathsf{b}|\!\})\ u \\
 & & \qquad \mathbf{where}\ \mathit{UIn}\ r\ a\ u = \mathit{reduce}\{\!|\mathsf{a}|\!\}\ x \\
\mathit{reduce}\{\!|\mathsf{a :\!+\!: b}|\!\} & (\mathit{Inr}\ x) & = \mathit{UIn}\ (r + \mathit{arity}\{\!|\mathsf{a}|\!\})\ (\mathit{arity}\{\!|\mathsf{a}|\!\} + a)\ u \\
 & & \qquad \mathbf{where}\ \mathit{UIn}\ r\ a\ u = \mathit{reduce}\{\!|\mathsf{b}|\!\}\ x\ .
\end{array}
$$

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 $\mathit{expand}\{\!|\mathsf{t}|\!\}$ is defined as follows:

$$
\begin{array}{ll}
\mathit{expand}\{\!|\mathsf{t} :: \star|\!\} & :: (\mathit{arity}\{\!|\mathsf{t}|\!\}, \mathit{expand}\{\!|\mathsf{t}|\!\}) \Rightarrow \mathsf{Univ} \to \mathsf{t} \\
\mathit{expand}\{\!|\mathsf{Int}|\!\}\ (\mathit{UIn}\ 0\ 1\ u) & = \mathit{expandbase}\{\!|\mathsf{Int}|\!\}\ u \\
\mathit{expand}\{\!|\mathsf{Zero}|\!\}\ \_ & = \mathit{error}\ \texttt{"expand"} \\
\mathit{expand}\{\!|\mathsf{a :\!+\!: b}|\!\}\ (\mathit{UIn}\ r\ a\ u) & \\
\quad \mid a \equiv aa + ab \wedge r < aa & = \mathit{Inl}\ (\mathit{expand}\{\!|\mathsf{a}|\!\}\ (\mathit{UIn}\ r\ (a - ab)\ u)) \\
\quad \mid a \equiv aa + ab & = \mathit{Inr}\ (\mathit{expand}\{\!|\mathsf{b}|\!\}\ (\mathit{UIn}\ (r - aa)\ (a - aa)\ u)) \\
\quad \mid \mathit{otherwise} & = \mathit{error}\ \texttt{"expand"} \\
\quad \mathbf{where}\ (aa, ab) = (\mathit{arity}\{\!|\mathsf{a}|\!\}, \mathit{arity}\{\!|\mathsf{b}|\!\})\ . &
\end{array}
$$

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.

4.3 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.

$$
\begin{array}{l}
\mathbf{data}\ \mathsf{UnivS}\ = \mathit{UIn}\ \mathsf{Integer}\ \mathsf{UnivP} \\
\mathbf{data}\ \mathsf{UnivP}\ = \mathit{UNil} \mid \mathit{UCons}\ \mathsf{UBase}\ \mathsf{UnivP} \\
\mathbf{data}\ \mathsf{UBase} = \mathit{UInt}\ \mathsf{Int} \mid \mathit{USum}\ \mathsf{UnivS}
\end{array}
$$

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

4.4 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:

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

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.

5 An XML Schema–Haskell data binding

XML [44] is the core technology of modern data exchange. An XML document is essentially a tree-based data structure, usually, but not necessarily, structured according to a type declaration such as a schema. A number of alternative methods of processing XML documents are available:

- **XML API's**. A conventional API such as SAX or the W3C's DOM can be used, together with a programming language such as Java or VBScript, to access the components of a document after it has been parsed.
- **XML programming languages**. A specialized programming language such as XSLT [45], XDuce [18], Yatl [10], XMλ [31,39], SXSLT [20], XStatic [14] etc. can be used to transform XML documents.
- **XML data bindings**. XML values can be 'embedded' in an existing programming language by finding a suitable mapping between XML types and types of the programming language [32].

Using a specialized programming language or a data binding has significant advantages over the SAX or DOM approach. For example, parsing comes for free and can be optimized for a specific schema. Also, it is easier to implement, test and maintain software in the target language. A data binding has the further advantages that existing programming language technology can be leveraged, and that a programmer need not account for XML idiosyncracies (though this may be a disadvantage for some applications). Programming languages for which XML data bindings have been developed include Java [30] and Python, as well as declarative programming languages such as Prolog [11] and Haskell [43,50]. Using Haskell as the target for an XML data binding offers the advantages of a typed higher-order programming language with a powerful type system.

In this section we present UUXML, a 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].

5.1 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 [34], 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 analagous to the HaXml translation of DTDs to Haskell [50].

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) [8]; that treatment also forms the basis of the W3C's own, more ambitious but as yet unfinished, formal semantics [46]. 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.

5.2 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.

5.2.1 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 [8], which is more orthogonal and suited to formal manipulation. In our syntax, the declarations above are written:

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

and the example document above is written:

```
doc[author["James Joyce"],title["Ulysses"],year["1922"]] .
```

5.2.2 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.

5.2.3 Overview.

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

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

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

This grammar is only a rough approximation 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.

5.2.4 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**.

5.2.5 User-defined types.

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

5.2.6 Mixed content.

Mixed content allows interspersing elements with text. More precisely, a document d matches **mix**(g) if *unmix*(d) matches g, where *unmix*(d) is obtained from d by deleting all character text at the top level. An example of mixed content is an XHTML paragraph element with emphasized phrases; in MSL its content would be declared as **mix**(em*). The opposite of 'mixed content' is 'element-only content.'

DTDs support a similar, but subtly different, notion of mixed content, specified by a declaration such as:

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

This allows `em` elements to be interspersed with character data when appearing as the children of `text` (but not as descendants of children). Groups involving `#PCDATA` can only appear in two forms, either by itself, or in a repeated disjunction involving only element names:

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

To see how Schema's notion of mixed content differs from DTDs', observe that a reasonable translation of the DTD content type above is $[\,\textsf{String :+: } [\![\texttt{em}]\!]_G\,]$ where $[\![\texttt{em}]\!]_G$ is the translation of `em`. This might lead one to think that we can translate a schema type such as $\mathbf{mix}(g)$ similarly as $[\,\textsf{String :+: } [\![g]\!]_G\,]$. 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:

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

Second, it cannot account for more complex types such as $\mathbf{mix}(\textsf{e}_1 \texttt{,} \ \textsf{e}_2)$. A document matching the latter type consists of two elements $\textsf{e}_1$ and $\textsf{e}_2$, 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

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

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

5.2.7 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.

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

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

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

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

5.2.8 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.

5.2.9 Derivation.

XML Schema also supports two kinds of *derivation* (which we sometimes also call *refinement*) by which new types can be obtained from old. The first kind, called *extension*, is quite similar to the notion of inheritance in object-oriented languages. The second kind, called *restriction*, is an 'additive' sort of subtyping, roughly dual to extension, which is multiplicative in character. As an example of extension, we declare a type *publication* obtained from *document* by adding fields at the end:

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

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:

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

then the content of t_2 is $(@a_1\ \texttt{\&}\ @a_2)\texttt{,}\ \mathsf{e}_1\texttt{,}\ \mathsf{e}_2$ not $@a_1\texttt{,}\mathsf{e}_1\texttt{,}@a_2\texttt{,}\mathsf{e}_2$.

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:

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

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.

5.3 An overview of the translation

The objective of the translation is to enable a programmer to write (Generic) Haskell programs on data corresponding to schema-conforming documents. (A detailed motivation of our translation scheme appears in section 6.2.) At minimum, we expect the translation to satisfy a type-soundness result which ensures that, if a document validates against a particular schema type, then the translated value is typeable in Haskell by the translated type.

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

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

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

This is the sort of translation employed by HaXml [50], and indeed we follow the same tack. In contrast, WASH [43] takes a decidedly different approach, encoding the state automaton corresponding to a regular pattern at the type level, and makes extensive use of type classes to express the transition relation.

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.

5.3.1 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 .
```

5.3.2 User-defined types.

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

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

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

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

5.3.3 Mixed content.

The reason each group g is translated to a first-order type $\mathsf{t} :: \star \to \star$ rather than a ground type is that the argument, which we call the 'mixity', indicates whether a document occurs in a mixed or element-only context.[3] Accordingly, u is restricted to be either String or (). For example, $e[t]$ translates as Elem $[\![e]\!]_G$ $[\![t]\!]_G$ () when it occurs in element-only content, and Elem $[\![e]\!]_G$ $[\![t]\!]_G$ 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' [21], 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
```

[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

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

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

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

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

5.3.4 Interleaving.

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

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

An unfortunate consequence of this is that we lose the ordering of the document values. For example, suppose we have a schema which describes a conference schedule where it is known that exactly three speakers of different types will appear. A part of such a schema may look like:

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

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

5.3.5 *Repetition.*

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

$$
\begin{array}{rlrl}
[\![0, 0]\!]_B = \mathsf{ZZ} & & [\![0, m+1]\!]_B = \mathsf{ZS}\ [\![0, m]\!]_B \\
[\![0, \infty]\!]_B = \mathsf{ZI} & & [\![m+1, n+1]\!]_B = \mathsf{SS}\ [\![m, n]\!]_B
\end{array}
$$

defined by:

$$
\begin{array}{ll}
\textbf{data}\ \mathsf{Rep\ g\ b\ u} & = \mathit{Rep}\ (\mathsf{b\ g\ u}) \\
\textbf{data}\ \mathsf{ZZ\ g\ u} & = \mathit{ZZ} \\
\textbf{data}\ \mathsf{ZI\ g\ u} & = \mathit{ZI}\ [\mathsf{g\ u}] \\
\textbf{data}\ \mathsf{ZS\ b\ g\ u} & = \mathit{ZS}\ (\mathsf{Maybe}\ (\mathsf{g\ u}))\ (\mathsf{Rep\ g\ b\ u}) \\
\textbf{data}\ \mathsf{SS\ b\ g\ u} & = \mathit{SS}\ (\mathsf{g\ u})\ (\mathsf{Rep\ g\ b\ u})\ .
\end{array}
$$

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:

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

5.3.6 *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:

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{T\_anyType\ u} & = & \mathit{T\_anyType} \\
\textbf{data}\ \mathsf{LE\_T\_anyType\ u} & = & \mathit{EQ\_T\_anyType}\ (\mathsf{T\_anyType\ u}) \\
& \mid & \mathit{LE\_T\_anySimpleType}\ (\mathsf{LE\_T\_anySimpleType\ u}) \\
& \mid & \mathit{LE\_T\_anyElem}\ (\mathsf{LE\_T\_anyElem\ u}) \\
& \mid & \mathit{LE\_T\_document}\ (\mathsf{LE\_T\_document\ u})\ .
\end{array}
$$

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

$$fst(split(t_1)) \;\&\; fst(split(t_2'))\;,\;(snd(split(t_1))\;\;,\;snd(split(t_2')))\;.$$

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

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

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

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

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

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

5.4 From XML documents to Haskell data

In this subsection 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 .
```

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. The type of generic parsers is the kind-indexed type GParse{[κ]} t and *gParse*{|t|} denotes a parser which tries to read a document into a value of type t. We now describe its behavior on the various components of Schema.

```
gParse{|t :: ⋆|}   :: (gParse{|t|}) ⇒ P t
gParse{|String|} = pMixed
gParse{|Unit|}   = pElementOnly
```

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

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

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

```
gParse{|Rep g b u|} = fmap Rep gParse{|b g u|}
gParse{|ZZ g u|} =    return ZZ
gParse{|ZI g u|} =    fmap ZI $ many gParse{|g u|}
gParse{|ZS g b u|} =  do x ← option gParse{|g u|}
                         y ← gParse{|b g u|}
                         return (ZS x (Rep y))
gParse{|SS g b u|} =  do x ← gParse{|g u|}
                         y ← gParse{|b g u|}
                         return (SS x (Rep y))
```

Repetitions are handled using the familiar combinators *many p* and *option p*, which parse 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 any *DText* chunk(s). Function *pReadableText* parses integers (also doubles and booleans—here omitted) using the standard Haskell *read* function, since we defined our alternative schema syntax to use Haskell syntax for the primitives.

```
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*{|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*{|e|} to acquire the name of an element; it has only one interesting case:

```
gName{|Con c a|} = drop 5 (conName c)
```

This case makes use of the special Generic Haskell syntax Con *c* 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.

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

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, $\mathit{gInter}\{\!|\mathsf{t}|\!\}$, which has the following type:

$$\mathit{gInter}\{\!|\mathsf{t} :: \star|\!\} :: (\mathit{gInter}\{\!|\mathsf{t}|\!\}) \Rightarrow \forall \mathsf{a}.\ \mathsf{PermP}\ (\mathsf{t} \to \mathsf{a}) \to \mathsf{PermP\ a}\ .$$

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

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

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, $<\!|?\!>$. *mapPerms* is the standard map function for the PermP type. *permute* q converts a permutation parser q into a normal parser.

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

To see how the above code works, observe that:

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

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.

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

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

$$
\begin{array}{ll}
\mathit{gInter}\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\} & = (<\!|\!|\!>\ \mathit{fmap}\ \mathit{Con}\ \mathit{gParse}\{\!|\mathsf{a}|\!\}) \\
\mathit{gInter}\{\!|\mathsf{Inter\ g1\ g2\ u}|\!\} & = \mathit{gInter}\{\!|\mathsf{g1\ u}|\!\} \circ \mathit{gInter}\{\!|\mathsf{g2\ u}|\!\} \\
 & \quad \circ\ \mathit{mapPerms}\ (\lambda f\ x\ y \to f\ (\mathit{Inter}\ x\ y)) \\
\mathit{gInter}\{\!|\mathsf{Rep\ g\ (ZS\ ZZ)\ u}|\!\} & = (<\!|?\!>\ (\mathit{Rep}\ \mathit{gDefault}\{\!|\mathsf{(ZS\ ZZ)\ g\ u}|\!\} \\
 & \qquad\qquad ,\mathit{fmap}\ \mathit{Rep}\ \mathit{gParse}\{\!|\mathsf{(ZS\ ZZ)\ g\ u}|\!\}))
\end{array}
$$

In the Con case, we see that an atomic type (an element or attribute name) produces a permutation parser transformer of the form $(<\!|\!|\!>\ q)$. The Inter case composes such parsers, so more generally we obtain parser transformers of the form $(<\!|\!>\ q_1 <\!|\!>\ q_2 <\!|\!>\ q_3 <\!|\!>\ ...)$. The Rep case is only ever called when g is atomic and the bounds are of the form ZS ZZ: this corresponds to a Schema type like $\mathsf{e}\{0, 1\}$, that is, an optional element (or attribute).[4] $\mathit{gDefault}\{\!|\mathsf{t}|\!\}$ is a simple auxiliary declaration which provides a value of the appropriate type when the optional value is omitted.

6 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, presented in the previous section, suffers from this problem.

In this section we use UUXML as a case study, to show how iso inference can be used to address a practical problem, the problem of overwhelmingly complex data representation which tends to accompany type-safe language embeddings. We outline the problem, explain how the design criteria gave rise to it, and finally show how to attack it.

In essence, our strategy will be to define a customized datatype, one chosen by the client programmer especially for the application. We use our mechanism to automatically infer the functions which convert to and from the customized representation by bracketing the core of the program with $\mathit{reduce}\{\!|\mathsf{t}|\!\}$ and $\mathit{expand}\{\!|\mathsf{t}|\!\}$. Generic Haskell does the rest, and the programmer is largely relieved from the burden imposed by the UUXML data representation.

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.

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

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.

```
<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:

> *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*))))))))) .

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.

6.2 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 [50] 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 [8], 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 [43], 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. But if we aim to use UUXML for more conventional applications, as we have demonstrated, it can become an overwhelming problem.

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 one is obliged to express as much information as possible structurally, and this too 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.

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`:

```
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.

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

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

For example, the following 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 above
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|}
```

6.4 The Role of Coercions

Recall that our system infers two non-invertible coercions:

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

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 is only partly because, in reducing to a universal type, we have thrown away the type information. Even if we knew the types involved, it is not clear, for example, whether the coercion a → a :+: a should determine 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 so well-equipped at dealing 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.

6.5 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.

7 Conclusions

This paper describes:

- a simple, powerful and general 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 isomorophisms solves some usability problems stemming from the complexity of UUXML.

Our inference mechanism leverages the power of an existing tool, Generic Haskell, and the established and growing 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.

7.1 Related & Future Work

7.1.1 XML frameworks and languages.

We have already mentioned the HaXML [50] and WASH [43] XML data bindings for Haskell. The Model Schema Language semantics [8] is now superseded by newer work [40]; we are investigating how to adapt UUXML to the more recent treatment. Special-purpose languages, such as XSLT [45], XDuce [18], Yatl [10], XMλ [31,39], SXSLT [20] and Xtatic [14], take a different approach to XML problems.

7.1.2 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 [37]. 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. [9]. More recent work has focused on linear type isomorphisms [6,41,38,29].

In category theory, Mac Lane initiated the study of coherence in a seminal paper [27]; his book [28] treats the case for monoidal categories. Beylin and Dybjer's use [7] 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 [23,22] 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.

7.1.3 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 [36] 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.

7.1.4 Implicit coercions.

Thatte introduced a declaration construct for introducing user-defined, *implicit* conversions between types [42], 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.

7.1.5 The Generic Haskell implementation.

We see several ways to improve our current implementation of iso inference.

- We would like to detect inference errors statically rather than dynamically (see below).
- Inferring more isomorphisms (such as the linear currying isos) and more powerful kinds of isomorphisms (such as commutativity of products and sums, and distributivity of one over the other) is also attractive.
- Currently, adding new *ad hoc* coercions requires editing the source code; since such coercions typically depend on the domain of application, a better approach would be to somehow parametrize the code by them, perhaps using *first-class generic functions*.
- We could exploit the fact that Generic Haskell allows to define type cases on the $\rightarrow$ type constructor: instead of providing two generic functions $\mathit{reduce}\{\!|\mathsf{t}|\!\}$ and $\mathit{expand}\{\!|\mathsf{t}|\!\}$, we would provide only a single generic function:

$$\mathit{coerce}\{\!|\mathsf{t} \rightarrow \mathsf{t}'|\!\} = \mathit{expand}\{\!|\mathsf{t}'|\!\} \circ \mathit{reduce}\{\!|\mathsf{t}|\!\} .$$

- The fact that the unique witness property does not readily transfer from type schemes to types might be circumvented by inferring first-class polymorphic functions which can then be instantiated at suitable types. Generic Haskell does not currently allow to do so, but if we could write expressions like $\mathit{coerce}\{\!|\forall \mathsf{a}\ \mathsf{b}.\ (\mathsf{a},\mathsf{b}) \to (\mathsf{b},\mathsf{a})|\!\}$ we could infer all canonical isos, without restriction, and perhaps handle examples like Date_NL and Date_US from section 1.

7.1.6 UUXML.

We have so far developed a prototype implementation of the translation and checked its correctness with a few simple examples and some slightly larger ones, such as the generic parser presented here and a generic pretty-printer. Future work may involve extending the translation to cover more Schema features such as facets and wildcards, adopting the semantics described in more recent work [40], which more accurately models Schema's named typing, and exploiting the 1-unambiguity constraint to obtain a more economical translation.

7.1.7 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 expand and reduce 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 NF{|t|} could be defined as follows.

```
type NF{|t|}              = Norm{|t|} Unit
data Norm{|Unit|} t       = NUnit t
data Norm{|a :*: b|} t    = NProd (a :*: (b :*: t))
data Norm{|Int|}  t       = NBase (Int :*: t)
```

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 NF{|Int|} and NF{|(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 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 NF{|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 and Andres Löh for their comments on this paper, and Eelco Dolstra and Fermin Reig for comments on an earlier version. 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.

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] 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.

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

[9] Kim B. Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.

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

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

[21] 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).

[22] Thomas S.H. Leinster. *Operads in Higher-Dimensional Category Theory.* PhD thesis, Trinity College and St John's College, Cambridge, 2000.

[23] Tom Leinster. *Higher Operads, Higher Categories.* Cambridge University Press, 2003.

[24] 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/`.

[25] Andres Löh. *Exploring Generic Haskell.* PhD thesis, Utrecht University, 2004.

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

[27] Saunders Mac Lane. Natural associativity and commutativity. *Rice University Studies*, 49:28–46, 1963.

[28] 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).

[29] Bruce McAdam. How to repair type errors automatically. In *Trends in Functional Programming (Proc. Scottish Functional Programming Workshop)*, volume 3, 2001.

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

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

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

[33] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, May 1997.

[34] OASIS. RELAX NG. `http://www.relaxng.org`, 2001.

[35] Simon Peyton Jones, John Hughes, et al. Haskell 98 — A non-strict, purely functional language. Available from `http://haskell.org`, Feb 1999.

[36] 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.

[37] Mikael Rittri. Retrieving library identifiers via equational matching of types. In *Conference on Automated Deduction*, pages 603–617, 1990.

[38] Mikael Rittri. Retrieving library functions by unifying types modulo linear isomorphism. *Informatique Theorique et Applications*, 27(6):523–540, 1993.

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

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

[41] 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] 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.

[43] Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4&5):435–468, July 2002.

[44] W3C. XML 1.0. `http://www.w3.org/XML/`, 1998.

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

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

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