Inferring Type Isomorphisms Generically

Frank Atanassow and Johan Jeuring

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

Abstract. Datatypes which differ inessentially in their names and structure are said to be isomorphic; for example, a ternary product is isomorphic to a nested pair of binary products. In some canonical cases, the conversion function is uniquely determined solely by the two types involved. In this article we describe and implement a program in Generic Haskell which automatically infers this function by normalizing types w.r.t. an algebraic theory of canonical isomorphisms. A simple generalization of this technique also allows to infer some non-invertible coercions such as projections, injections and *ad hoc* coercions between base types. We explain how this technique has been used to drastically improve the usability of a Haskell–XML Schema data binding, and suggest how it might be applied to improve other type-safe language embeddings.

1 Introduction

Typed functional languages like Haskell [27] and ML [16, 25] 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, he or she 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:

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

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 of representations to be simultaneously supported is large, then the size of the conversion code might 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] 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.

Monoidal isos. A few canonical isos of Haskell are summarized by the syntactic theory below.[2]

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

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. they occur in the same order on both sides.

The strong regularity condition is adapted from work on generalized multicategories [15, 14, 10]. We claim, but have not yet proved, that strong regularity 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.

[2] We use the type syntax familiar from the Generic Haskell literature, i.e., Unit and :∗: are respectively nullary and binary product, and Zero and :+: are respectively nullary and binary sum constructors.

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

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

induces the canonical iso

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

We call such isos *datatype isos*.

Note that datatype isos are *not* strongly regular in general; for example the List identity mentions a twice on the right-hand side. Intuitively, though, there is only one witness to a datatype iso: the constructor(s). Again, we claim, and hope in the future to prove, that isos of this sort uniquely determine a canonical witness. Largely as a side effect of the way Generic Haskell works, our inference mechanism *does* infer datatype isos.

1.2 Outline

The remainder of this article is organized as follows. In section 2 we give an informal description of the user interface to our inference mechanism. Section 3 discusses a significant application of iso inference, a way of automatically customizing a Haskell–XML Schema data binding. In section 4 we examine the Generic Haskell implementation of our iso inferencer. Finally, in section 5 we summarize our results, and discuss related work and possibilities for future work in this area.

2 Inferring Isomorphisms

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

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

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

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

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

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

which evaluates to

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

Function $\mathit{reduce}\{\!|\mathsf{t}|\!\}$ picks a type in each isomorphism class which serves as a normal form, and uses the canonical witness to convert values of t to that form. Normalized values are represented in a special way in the abstract type Univ; a typical user need not understand the internals of Univ unless $\mathit{expand}\{\!|\mathsf{t}'|\!\}$ fails. If t and t' are 'essentially' the same, yet structurally substantially 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:

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

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

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

evaluates to *Left True*; note that it cannot evaluate to *Right* 4 because such a reduction would involve projecting a suffix and injecting into the right whereas we infer only prefix projections and left injections. Of course, we would prefer our theory to include the dual pair of coercions as well, but doing so would break the property that each pair of types determines a unique canonical witness. Nevertheless, we will see in section 3.4 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 include commutativity of $\mathrel{:\!*\!:}$.
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 5, we discuss some possible future work which may provide static error detection.

3 Improving a Haskell–XML Schema Data Binding

A program that processes XML documents can be implemented using an *XML data binding.* An XML data binding [23] translates an XML document to a value of some programming language. Such bindings have been defined for a number of programming languages including Java [21, 24], Python [26], Prolog [7] and Haskell [35, 37, 1]. The default translation scheme of a data binding may produce unwieldy, convoluted and redundant types and values. Our own Haskell–XML Schema binding, called UUXML [1], 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 *reduce*$\{|\mathsf{t}|\}$ and *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.

3.1 The Problem with UUXML

We do not have the space here to describe UUXML in detail, but 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-iliad">
  <author>Homer</author>
  <title>The Iliad</title>
</doc>
```

Our binding tool translates each of the types `doc` and `docType` into a pair of types (explained in the next section),

```
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-iliad"))))(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 Iliad")))))(Rep
(ZS Nothing (Rep ZZ))))))))))
```

Even without knowing the details of the encoding or definitions of the unfamiliar datatypes, one can see the problem here; 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-iliad"`. For larger, more complex documents or document types, the problem can be even worse.

3.2 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 [37] 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[3] than DTDs.

Third, the UUXML translation closely follows the Model Schema Language (MSL) formal semantics [4], 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`.[4] 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 [1], 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 [35], 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

[3] "Mixed content" refers to character data interspersed with elements. For example, in XHTML a `p` element can contain both character data and other elements like `em`.

[4] To help illustrate this in the example schema translation, we posited that `docType` had a hypothetical subtype `publicationType`. It appears as the body of the second constructor of LE_T_docType in section 3.1.

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, yet more. 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.

3.3 Exploiting Isomorphisms

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

```
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 denote the parts which require customization—a maintenance hazard since the schema structure may change.

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

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

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

For example, the 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 two datatype 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|}
```

3.4 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 of 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 $\mathsf{a} \rightarrow \mathsf{a} :\!+\!: \mathsf{a}$ 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.

3.5 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 succumb to 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.

4 Generic Isomorphisms

In this section, we describe how to automatically generate isomorphisms between pairs of datatypes. Our implementation platform is Generic Haskell, and in particular we use dependency-style GH [17]. This section assumes a basic familiarity with Generic Haskell, but the definitions are all remarkably simple.

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

In each case, we build the requisite morphism by reducing a value $v :: \mathsf{t}$ to a value of a universal data type $u = \mathit{reduce}\{\!|\mathsf{t}|\!\}\ v :: \mathsf{Univ}$. 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.

4.1 Handling Products

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

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

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.

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

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

$$\begin{array}{lcl}
\textbf{type}\ \mathsf{ReduceBase}\{[\star]\}\ \mathsf{t} & = & \mathsf{t} \to \mathsf{UBase} \\
\mathit{reducebase}\{\!|\mathsf{t} :: \kappa|\!\} & :: & \mathsf{ReduceBase}\{[\kappa]\}\ \mathsf{t} \\
\mathit{reducebase}\{\!|\mathsf{Int}|\!\}\ i & = & \mathit{UInt}\ i \\
\textbf{type}\ \mathsf{ExpandBase}\{[\star]\}\ \mathsf{t} & = & \mathsf{UBase} \to \mathsf{t} \\
\mathit{expandbase}\{\!|\mathsf{t} :: \kappa|\!\} & :: & \mathsf{ExpandBase}\{[\kappa]\}\ \mathsf{t} \\
\mathit{expandbase}\{\!|\mathsf{Int}|\!\}\ (\mathit{UInt}\ i) & = & i
\end{array}$$

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:

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

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.

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

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:

$$
\begin{array}{lcl}
\textbf{type } \mathsf{Len}\{[\star]\}\ \mathsf{t} & = & \mathsf{Int} \\
\mathit{len}\{|\mathsf{t} :: \kappa|\} & :: & \mathsf{Len}\{[\kappa]\}\ \mathsf{t} \\
\mathit{len}\{|\mathsf{Int}|\} & = & 1 \\
\mathit{len}\{|\mathsf{Unit}|\} & = & 0 \\
\mathit{len}\{|\mathsf{a} :\!*\!: \mathsf{b}|\} & = & \mathit{len}\{|\mathsf{a}|\} + \mathit{len}\{|\mathsf{b}|\}\ .
\end{array}
$$

Observe that Unit is assigned length zero.

Now we can write $\mathit{expand}\{|\mathsf{t}|\}$; like $\mathit{reduce}\{|\mathsf{t}|\}$, it is defined in terms of a helper function $\mathit{exp}\{|\mathsf{t}|\}$, this time in a dual fashion with the ‘unparsed’ remainder appearing as output.

$$
\begin{array}{lcl}
\textbf{type } \mathsf{Exp}\{[\star]\}\ \mathsf{t} & = & \mathsf{Univ} \to (\mathsf{t}, \mathsf{Univ}) \\
\mathit{exp}\{|\mathsf{t} :: \kappa|\} & :: & \mathsf{Exp}\{[\kappa]\}\ \mathsf{t} \\
\mathit{exp}\{|\mathsf{Int}|\}\ (u : \mathit{us}) & = & (\mathit{expandbase}\{|\mathsf{Int}|\}\ u, \mathit{us}) \\
\mathit{exp}\{|\mathsf{Int}|\}\ [\,] & = & \mathit{error}\ \texttt{"exp"} \\
\mathit{exp}\{|\mathsf{Unit}|\}\ \mathit{us} & = & (\mathsf{Unit}, \mathit{us}) \\
\mathit{exp}\{|\mathsf{a} :\!*\!: \mathsf{b}|\}\ \mathit{us} & = & \textbf{let } (u, \mathit{us}') = \mathit{exp}\{|\mathsf{a}|\}\ \mathit{us} \\
 & & \qquad (v, \mathit{us}'') = \mathit{exp}\{|\mathsf{b}|\}\ \mathit{us}' \\
 & & \textbf{in } (u :\!*\!: v, \mathit{us}'') \\
\textbf{type } \mathsf{Expand}\{[\star]\}\ \mathsf{t} & = & \mathsf{Univ} \to \mathsf{t} \\
\mathit{expand}\{|\mathsf{t} :: \kappa|\} & :: & \mathsf{Expand}\{[\kappa]\}\ \mathsf{t} \\
\mathit{expand}\{|\mathsf{t}|\}\ u & = & \textbf{case } \mathit{exp}\{|\mathsf{t}|\}\ u \textbf{ of} \\
 & & \quad (v, [\,]) \to v \\
 & & \quad (v, \_) \to \mathit{error}\ \texttt{"expand"}
\end{array}
$$

In the last case, 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 $\mathit{expand}\{\!|\mathsf{t}|\!\}$ 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:

$$\mathbf{data}\ \mathsf{Zero}\ .$$

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

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

Note that this datatype Univ is isomorphic to:

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

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

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

If $u = \mathit{UIn}\ r\ a\ b$ then we shall 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.

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

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

i	$::\ \mathsf{I}$	$\mapsto \mathit{UIn}\ 0\ 1\ u$
$\mathit{Inl}\ i$	$::\ \mathsf{I}\ {:}{+}{:}\ \mathsf{Zero}$	$\mapsto \mathit{UIn}\ 0\ 1\ u$
$\mathit{Inr}\ i$	$::\ \mathsf{Zero}\ {:}{+}{:}\ \mathsf{I}$	$\mapsto \mathit{UIn}\ 0\ 1\ u$
$\mathit{Inl}\ i$	$::\ \mathsf{I}\ {:}{+}{:}\ \mathsf{I}$	$\mapsto \mathit{UIn}\ 0\ 2\ u$
$\mathit{Inr}\ i$	$::\ \mathsf{I}\ {:}{+}{:}\ \mathsf{I}$	$\mapsto \mathit{UIn}\ 1\ 2\ u$
$\mathit{Inl}\ (\mathit{Inl}\ i)$	$::\ (\mathsf{I}\ {:}{+}{:}\ \mathsf{I})\ {:}{+}{:}\ \mathsf{I}$	$\mapsto \mathit{UIn}\ 0\ 3\ u$
$\mathit{Inl}\ (\mathit{Inr}\ i)$	$::\ (\mathsf{I}\ {:}{+}{:}\ \mathsf{I})\ {:}{+}{:}\ \mathsf{I}$	$\mapsto \mathit{UIn}\ 1\ 3\ u$
$\mathit{Inr}\ i$	$::\ (\mathsf{I}\ {:}{+}{:}\ \mathsf{I})\ {:}{+}{:}\ \mathsf{I}$	$\mapsto \mathit{UIn}\ 2\ 3\ u$

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

$$
\begin{array}{lcl}
\textbf{type}\ \mathsf{Arity}\{[\star]\}\ \mathsf{t} & = & \mathsf{Int} \\
\mathit{arity}\{|\mathsf{t} :: \kappa|\} & :: & \mathsf{Arity}\{[\kappa]\}\ \mathsf{t} \\
\mathit{arity}\{|\mathsf{Int}|\} & = & 1 \\
\mathit{arity}\{|\mathsf{Zero}|\} & = & 0 \\
\mathit{arity}\{|\mathsf{a}\ {:}{+}{:}\ \mathsf{b}|\} & = & \mathit{arity}\{|\mathsf{a}|\} + \mathit{arity}\{|\mathsf{b}|\}
\end{array}
$$

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

$$
\begin{array}{lcl}
\textbf{type}\ \mathsf{Reduce}\{[\star]\}\ \mathsf{t} & = & \mathsf{t} \to \mathsf{Univ} \\
\mathit{reduce}\{|\mathsf{t} :: \kappa|\} & :: & \mathsf{Reduce}\{[\kappa]\}\ \mathsf{t} \\
\mathit{reduce}\{|\mathsf{Int}|\}\ i & = & \mathit{UIn}\ 0\ 1\ (\mathit{reducebase}\{|\mathsf{Int}|\}\ i) \\
\mathit{reduce}\{|\mathsf{Zero}|\}\ \_ & = & \bot \\
\mathit{reduce}\{|\mathsf{a}\ {:}{+}{:}\ \mathsf{b}|\}\ (\mathit{Inl}\ x) & = & \mathit{UIn}\ r\ (a + \mathit{arity}\{|\mathsf{b}|\})\ u \\
\quad \textbf{where}\ \mathit{UIn}\ r\ a\ u & = & \mathit{reduce}\{|\mathsf{a}|\}\ x \\
\mathit{reduce}\{|\mathsf{a}\ {:}{+}{:}\ \mathsf{b}|\}\ (\mathit{Inr}\ x) & = & \mathit{UIn}\ (r + \mathit{arity}\{|\mathsf{a}|\})\ (\mathit{arity}\{|\mathsf{a}|\} + a)\ u \\
\quad \textbf{where}\ \mathit{UIn}\ r\ a\ u & = & \mathit{reduce}\{|\mathsf{b}|\}\ x\ .
\end{array}
$$

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:

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

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.

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

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

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:

```
a :*: b ⩽ a                          a ⩽ a :+: b
```

***Ad Hoc* Coercions.** Schema (and most other conventional languages) also defines a subtyping relation between primitive types. For example, `int` is a subtype of `integer` which is a subtype of `decimal`. We can easily model this by (adding some more base types and) modifying the functions which convert base types.

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

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 Conclusions

In this paper, we have described a simple, powerful and general mechanism for automatically inferring a well-behaved class of isomorphisms, and demonstrated how it addresses some usability problems stemming from the complexity of our Haskell-XML Schema data binding, UUXML. Our mechanism leverages the power of an existing tool, Generic Haskell, and the established and growing theory of type isomorphisms.

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.

5.1 Related Work

Besides UUXML, we have already mentioned the HaXML [37] and WASH [35] XML data bindings for Haskell. The Model Schema Language semantics [4] is now superseded by newer work [32]; we are investigating how to adapt our encoding to the more recent treatment. Special-purpose languages, such as XSLT [36], XDuce [12], Yatl [6], XMλ [22, 31], SXSLT [13] and Xtatic [9], take a different approach to XML problems.

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 [29]. Since then the area has ballooned; good places to start on the theory of type isomorphisms is Di Cosmo's book [8] and the paper by Bruce et al. [5]. More recent work has focused on linear type isomorphisms [2, 33, 30, 20].

In category theory, Mac Lane initiated the study of coherence in a seminal paper [18]; his book [19] treats the case for monoidal categories. Beylin and Dybjer's use [3] 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 [15, 14] and Hermida [10] 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.

5.2 Future Work

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 [28] for a survey of approaches.

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

Implicit coercions. Thatte introduced a declaration construct for introducing user-defined, *implicit* conversions between types [34], 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.

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.
- 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 $\mathit{coerce}\{|\forall \mathsf{a}\ \mathsf{b}\,.\,(\mathsf{a},\mathsf{b}) \rightarrow (\mathsf{b},\mathsf{a})|\}$ we could infer all canonical isos, without restriction, and perhaps handle examples like Date_NL and Date_US from section 1.

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* [11]. 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:

$$\begin{array}{lll} \mathit{reduce}\{|\mathsf{t}|\} & :: & \mathsf{t} \to \mathsf{NF}\{[\mathsf{t}]\} \\ \mathit{expand}\{|\mathsf{t}|\} & :: & \mathsf{NF}\{[\mathsf{t}]\} \to \mathsf{t}\ . \end{array}$$

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

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

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

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

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

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

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

Acknowledgements. The authors thank Dave Clarke, Bastiaan Heeren 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. In *Proc. 7th Brazilian Symposium on Programming Languages*, 2003. See also Utrecht University technical report UU-CS-2003.

2. Vincent Balat and Roberto Di Cosmo. A linear logical view of linear type isomorphisms. In *CSL*, pages 250–265, 1999.
3. 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.
4. Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler. MSL: A model for W3C XML Schema. In *Proc. WWW10*, May 2001.
5. Kim B. Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.
6. Sophie Cluet and Jérôme Siméon. YATL: a functional and declarative language for XML, 2000.
7. Jorge Coelho and Mário Florido. Type-based XML processing in logic programming. In *PADL 2003*, pages 273–285, 2003.
8. Roberto Di Cosmo. *Isomorphisms of Types: From lambda-calculus to Information Retrieval and Language Design*. Birkhäuser, 1995.
9. Vladimir Gapeyev and Benjamin C. Pierce. Regular object types. In *European Conference on Object-oriented Programming (ECOOP 2003)*, 2003.
10. C. Hermida. Representable multicategories. *Advances in Mathematics*, 151:164–225, 2000.
11. 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.
12. Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Third International Workshop on the Web and Databases (WebDB), volume 1997 of Lecture Notes in Computer Science*, pages 226–244, 2000.
13. Oleg Kiselyov and Shriram Krishnamurti. SXSLT: manipulation language for XML. In *PADL 2003*, pages 226–272, 2003.
14. Thomas S.H. Leinster. *Operads in Higher-Dimensional Category Theory*. PhD thesis, Trinity College and St John's College, Cambridge, 2000.
15. Tom Leinster. *Higher Operads, Higher Categories*. Cambridge University Press, 2003.
16. 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/`.
17. Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In *Proceedings of the International Conference on Functional Programming (ICFP '03)*, August 2003.
18. Saunders Mac Lane. Natural associativity and commutativity. *Rice University Studies*, 49:28–46, 1963.
19. 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).
20. Bruce McAdam. How to repair type errors automatically. In *Trends in Functional Programming (Proc. Scottish Functional Programming Workshop)*, volume 3, 2001.
21. Brett McLaughlin. *Java & XML data binding*. O'Reilly, 2003.
22. Erik Meijer and Mark Shields. XMLambda: A functional language for constructing and manipulating XML documents. Available from `http://www.cse.ogi.edu/~mbs/`, 1999.
23. Eldon Metz and Allen Brookes. XML data binding. *Dr. Dobb's Journal*, pages 26–36, March 2003.
24. Sun Microsystems. Java Architecture for XML Binding (JAXB). `http://java.sun.com/xml/jaxb/`, 2003.

25. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, May 1997.
26. Uche Ogbuji. Xml data bindings in python, parts 1 & 2. *xml.com*, 2003. `http://www.xml.com/pub/a/2003/06/11/py-xml.html`.
27. Simon Peyton Jones, John Hughes, et al. Haskell 98 — A non-strict, purely functional language. Available from `http://haskell.org`, February 1999.
28. 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.
29. Mikael Rittri. Retrieving library identifiers via equational matching of types. In *Conference on Automated Deduction*, pages 603–617, 1990.
30. Mikael Rittri. Retrieving library functions by unifying types modulo linear isomorphism. *Informatique Theorique et Applications*, 27(6):523–540, 1993.
31. 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/`.
32. Jérôme Siméon and Philip Wadler. The essence of XML. In *Proc. POPL 2003*, 2003.
33. 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.
34. 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.
35. Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4&5):435–468, July 2002.
36. W3C. XSL Transformations 1.0. `http://www.w3.org/TR/xslt`, 1999.
37. Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *International Conference on Functional Programming*, pages 148–159, 1999.