# Generic validation in an XPath-Haskell data binding

Rui Guerra  Johan Jeuring  S. Doaitse Swierstra

Institute of Information and Computing Sciences
Utrecht University
The Netherlands
{rui, johanj, doaitse}@cs.uu.nl

**Abstract**

An XPath data binding for a given host language provides a translation of XPath expressions to expressions in the host language. This paper discusses an XPath-Haskell data binding. XPath validation ensures that a path addresses a possibly nonempty set of nodes in XML documents described by an XML Schema. We present a generic function (defined by induction on the type structure) that validates XPath expressions with respect to an XML Schema. We use Generic Haskell, an extension of Haskell that supports the construction of generic programs. Furthermore we present generic programs that use a valid XPath expression to query and update documents in a typed way.

For some XPath expressions, the information present in a given schema is used to avoid visiting parts of a document whose type ensures that no matches or updates will occur.

## 1 Introduction

An XML document is *valid* if it conforms to a DTD or an XML Schema.

A type system can statically guarantee the validity of XML documents and transformations, and there has been an increased interest in using typed programming languages for XML processing [1, 2, 19, 6].

Instead of defining a special-purpose XML-programming language, we are interested in embedding XML values in an existing programming language, more specifically Haskell [10]. Such an embedding is called a *data binding*. There are a number of advantages of an XML data binding to an existing programming language, in particular to Haskell, over special-purpose XML-programming languages. Using Haskell as the target for an XML data binding offers all the advantages of using a higher order programming language with a powerful type system. Furthermore, we can reuse existing libraries and code in our programs for XML. Finally, we do not have to learn a new programming language, together with a new type system and tools, for writing programs that manipulate XML documents. The obvious disadvantage of using an XML data binding is that several XML-specific aspects, such as regular expression matching, are lost, or more difficult to address.

Several XML-Haskell data bindings have been constructed [1, 7, 16, 20]. In order to use the Haskell type checker to statically guarantee XML validity it is crucial to use a type-preserving data binding such as UUXML [1] or HaXml [20].

When processing an XML document it is often necessary to refer to a collection of subdocuments. XPath [4] is a language, recommended by W3C, that provides a means for addressing a collection of subdocuments. Existing XML-Haskell data bindings that include XPath functionality, such as Xtract [20], are untyped. In this paper we investigate how to embed XPath in Haskell in a typed way.

An XPath expression is *valid* if it addresses a possibly nonempty set of nodes in an XML document described by an XML Schema. If, by analysing the XML Schema, we find that an XPath expression always addresses an empty set of nodes, we call it *invalid*. Since an XPath expression is valid with respect to an XML Schema, we say that it *conforms to* an XML Schema. A type checker tests whether or not a value has a certain type. However, checking whether an XPath expression has the type of XPath expression only ensures that it is well-formed.

There is no straightforward way to use the Haskell type checker to verify the validity of an XPath expression. The good news however is that we can write a *generic function*, indexed by the type corresponding to the XML Schema, that can test whether an XPath expression is valid. This generic function can be expressed in Generic Haskell [5], a superset of Haskell. Here we give its type:

$valid\langle \mathsf{t} \rangle :: \mathsf{XPath} \to \mathsf{Bool}$

For example, given a Haskell data type Book for representing books, we can test validity of an XPath expression with respect to Book by means of the instance $valid\langle \mathsf{Book} \rangle$ of function *valid*. This function tests whether an XPath expression is valid with respect to Book, without accessing any specific value of this data type. By indexing *valid* with a specific type t, the Generic Haskell compiler generates a Haskell function that tests whether a given XPath expression is valid with respect to such type t.

XPath expressions are used in queries and in transformation programs. A typed approach to a querying or transformation system ensures:

**Query validity:** a query/transformation returns a possibly nonempty set of results.

**Result validity:** the query/transformation result is of the expected type.

We present generic functions to query and transform documents of any data type generated by a Haskell-XML data binding. Generic functions are typed, and so are their instances. This implies that the instances of the generic functions for querying and transforming documents are type correct by definition.

Furthermore, for some XPath expressions, the type information is used to prevent visiting parts of the document whose type ensures that no matches or updates will occur.

Thus we use the type systems of Haskell and Generic Haskell to obtain a typed data binding for XPath, and typical functions in which XPath expressions are used.

For example, the authors of a book can be queried by writing:

$compileQuery\langle \mathsf{Book} \rangle$ `"//Author"` *book*

where Book is the type of the document *book*. To double the price of a book, we write:

$compileUpd\langle \mathsf{Book} \rangle$ `"/Book/Price"` $(*2)$ *book*

This paper is organised as follows. Section 2 gives a Haskell-XML data binding and briefly introduces Generic Haskell. Section 3 gives a data binding for a subset of XPath and defines a generic validity test for XPath expressions. Section 4 describes a generic matching function for XPath. Section 5 describes generic functions for querying an XML document and Section 6 for updating it. Section 7 discusses related work, and Section 8 concludes.

## 2 Haskell and XML

### 2.1 Data binding

A common approach for processing XML documents is to use an existing programming language. Since we want to write strongly-typed, lazy higher-order functions, we use the functional language Haskell.

An XML document can be translated to Haskell using two different approaches. One is to use a universal data type that by means of which any XML document can be represented, independent of its DTD or XML Schema. We call this the untyped approach to data binding. Existing XML-Haskell untyped data bindings are HXML [7] and Haskell XML Toolbox [16]. The other approach is to map a DTD or an XML Schema to Haskell types and XML documents to Haskell values of the corresponding type, and we call this the typed approach. HaXml [20] implements both approaches. In addition to an untyped data binding, it includes a tool for translating any valid XML DTD into Haskell types. UUXML [1] is a typed data binding that maps XML Schema to Haskell.

```
<element name="Book">
  <complexType>
    <sequence>
    <choice>
      <element name="English"/>
      <element name="German"/>
      <element name="Dutch"/>
      <element name="Other"/>
    </choice>
    <element name="Title"   type="string"/>
    <element name="Author"  type="string"
    maxOccurs="unbounded"/>
    <element name="Date"    type="integer"/>
    <element name="Price"   type="integer"/>
    </sequence>
  </complexType>
</element>
```

```
data Book = Book Ch1 Title [Author] Date Price
data Ch1 = English | German | Dutch | Other
newtype Title  = Title  String
newtype Author = Author String
newtype Date   = Date   Int
newtype Price  = Price  Int
```

Figure 1: XML schema fragment and respective Haskell data type definition

Here we are interested in a typed data binding since we want to have as many static guarantees as possible. To not expose the idiosyncrasies of a specific data bind-

ing, we introduce a simplified version of a typed data binding by means of an example.

We give a partial XML Schema in Figure 1, with a possible corresponding Haskell data type. The data type Book describes a book as a language (one of *English*, *German*, *Dutch* and *Other*), a title and several authors of type String, and a date and a price, both of type Int. The following XML value is valid with respect to the XML Schema from Figure 1:

```
<Book>
   <English/>
   <Title>   Data on the Web  </Title>
   <Author>  Abiteboul        </Author>
   <Author>  Buneman          </Author>
   <Author>  Suciu            </Author>
   <Date>    1999             </Date>
   <Price>   20               </Price>
</Book>
```

and is translated to the value *book*:

```
book :: Book
book = Book English
            (Title   "Data on the Web")
            [Author "Abiteboul"
            ,Author "Buneman"
            ,Author "Suciu"]
            (Date 1999)
            (Price 20)
```

Since Haskell is a strongly typed language, an XML document that is mapped to a Haskell value has a specific type. In the example the value *book* is of type Book. The type annotation (*book* :: Book) is optional, since it may be inferred by the type system.

### 2.2 Generic Haskell and XML

Genericity is an attractive advantage of using a universal data type to represent XML documents [20]. Since XML documents are translated to a universal data type, a program defined for such a universal data type works for any XML document. On the other hand, a typed data binding has the advantage of statically guaranteeing XML validity, as mentioned in the previous section. By defining our programs in Generic Haskell, we get the best of both approaches: our programs are typed *and* generic.

Generic Haskell is an extension of Haskell that supports the definition of generic programs. In this paper, we use the most recent version of Generic Haskell, known as *Dependency-style Generic Haskell* [13]. So-called dependencies both simplify and increase expressiveness of generic programming. Here we give a very brief introduction to Generic Haskell; we refer the interested reader to more extensive background material [13, 9]. A generic program can be applied to any value of a large class of types. In order to do so, we map each data type that appears in a source program to its structural representation. This representation is expressed in terms of a limited set of data types, called structure types. A generic program is defined by induction on these structure types. Whenever a generic program is applied to a user-defined data type, the Generic Haskell compiler takes care of the mapping between the user-defined data type and its corresponding structural representation.

The idea of the translation of data types is thus: replace choice between constructors by a Sum (nested right-associatively if there are more than two constructors), and replace the sequence of arguments of a constructor by a Prod (nested right-associatively if there are more than two arguments). The structure types Sum and Prod are defined as:

$$\textbf{data}\ \mathsf{Sum}\ \mathsf{a}\ \mathsf{b} = \mathit{Inl}\ \mathsf{a} \mid \mathit{Inr}\ \mathsf{b}$$
$$\textbf{data}\ \mathsf{Prod}\ \mathsf{a}\ \mathsf{b} = \mathsf{a} \times \mathsf{b}$$

A nullary constructor is replaced by the structure type Unit:

$$\textbf{data}\ \mathsf{Unit} = \mathit{Unit}$$

The arguments of the constructors are not translated. For example, the data type of binary trees:

$$\textbf{data}\ \mathsf{Tree}\ (\mathsf{a} :: *) = \mathit{Leaf} \mid \mathit{Node}\ (\mathsf{Tree}\ \mathsf{a})\ \mathsf{a}\ (\mathsf{Tree}\ \mathsf{a})$$

is represented by:

$$\textbf{type}\ \mathit{Str}\ (\mathsf{Tree})\ (\mathsf{a} :: *) =$$
$$\quad \mathsf{Sum\ Unit\ (Prod\ (Tree\ a)\ (Prod\ a\ (Tree\ a)))}$$

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

$$\textbf{data}\ \mathsf{EP}\ (\mathsf{a} :: *)\ (\mathsf{b} :: *) = \mathit{EP}\ (\mathsf{a} \to \mathsf{b})\ (\mathsf{b} \to \mathsf{a})$$

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 (Sum), products (Prod), on base types such as Unit, Int and String. To be able to inspect constructor names, we will also encode the constructors in the structure type of a data type, 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) (a:: *) =
  Sum (Con Unit)
      (Con (Prod (Tree a) (Prod a (Tree a))))
```

For example, we define a very simple generic function that extracts the strings and integers that appear in an XML document of a data type t. For the data type Book, the function *contentBook* is defined by:

```
contentBook (Book language
                  (Title t)
                  authors
                  (Date d)
                  (Price p)) =
  t : [s | Author s ← authors] ++ [show d, show p]
```

On *book*, function *contentBook* returns the list ["Data on the Web", "Abiteboul", "Buneman", "Suciu", "1999", "20"].

The generic function *content* abstracts from the Book data type and it returns the document's content for any data type.

```
content⟨t :: *⟩ :: (content⟨t⟩) ⇒ t → [String]
content⟨Unit⟩       Unit     = []
content⟨Int⟩        int      = [show int]
content⟨String⟩     str      = [str]
content⟨Sum α β⟩    (Inl a)  = content⟨α⟩ a
content⟨Sum α β⟩    (Inr b)  = content⟨β⟩ b
content⟨Con c α⟩    (Con a)  = content⟨α⟩ a
content⟨Prod α β⟩   (a × b)  = content⟨α⟩ a
                             ++ content⟨β⟩ b
```

Instantiating the generic function *content* for the data type *book*, *content*⟨Book⟩, gives a Haskell function that is equivalent to the function *contentBook* defined above. The expression *content*⟨Book⟩ *book* would give the same list as above.

The function *content*⟨t⟩ is called a type-indexed value, since it takes an explicit type argument (t) that further specifies the function value.

## 3 XPath data binding and validation

XPath [4] is a language for addressing parts of an XML document and was designed to be used in other languages such as XQuery and XSLT. An XPath data binding for a given host language provides a translation of XPath expressions to expressions in the host language.

We give a Haskell data binding for a subset of XPath defined in XPath 2.0. This subset consists of: node tests, the child axis (/), the descendant-or-self axis (//) and wildcards (*). This subset of XPath expressions will be extended with predicates in Appendix A. The following Haskell types encode our subset of XPath expressions:

```
type XPath     = [Axis]
data Axis      = Axis AxisName NameTest
data AxisName  = Child | Dos deriving Eq
data NameTest  = QName String
               | Wildcard
```

The translation from XPath expressions to Haskell values of type XPath is straightforward. A well-formed XPath expression is parsed to a well-typed value of XPath using a function *toXPath*.

We not only want to verify that an XPath expression is well formed, but also that it is valid with respect to an XML Schema: that it addresses a possibly non-empty set of nodes in a document that is valid with respect to the given XML Schema. For example, the XPath expression "/Book/German" is valid with respect to Book, whereas "/Book/Spanish" is not, since "Spanish" is not one of the languages in which a Book can be written according to the XML schema given in Figure 1.

Well-formedness is guaranteed by the Haskell type checker. However, there is no straightforward way to use the type checker to validate an XPath expression with respect to a (translated) XML Schema. Nevertheless, by using a type-indexed function we can test whether a path is valid without being given a specific value.

For example, an XPath expression can be tested for validity with respect to the data type Book by writing:

```
valid⟨Book⟩ (toXPath "/Book/German")
```

where *valid* is the type-indexed function that tests whether the given XPath expression is valid with respect to the argument type t.

An XPath expression is tested for validity by sequentially matching its Axis with the tag names occurring in the XML Schema, which in our XML-Haskell data binding, have been mapped to constructor names.

The above XPath expression `"/Book/German"` is validated by comparing its first axis `"/Book"` with the top level constructor name of the type Book. If this succeeds, the next axis `"/German"` is compared with the constructor names of the arguments of *Book* which are *English*, *German*, *Dutch*, *Other*, *Title*, *Author Date* and *Price*. The validation succeeds when the entire XPath expression has been consumed.

We now present our first version of the generic function *valid*:

$$
\begin{array}{l}
valid\langle \mathsf{t} :: * \rangle :: (valid\langle \mathsf{t} \rangle) \Rightarrow \mathsf{XPath} \rightarrow \mathsf{Bool} \\
valid\langle \mathsf{Unit} \rangle \quad xpath = False \\
valid\langle \mathsf{Int} \rangle \quad xpath = False \\
valid\langle \mathsf{String} \rangle \quad xpath = False \\
valid\langle \mathsf{Sum}\ \alpha\ \beta \rangle \quad xpath = valid\langle \alpha \rangle\ xpath \\
\qquad\qquad\qquad\qquad \vee\ valid\langle \beta \rangle\ xpath \\
valid\langle \mathsf{Prod}\ \alpha\ \beta \rangle\ xpath = valid\langle \alpha \rangle\ xpath \\
\qquad\qquad\qquad\qquad \vee\ valid\langle \beta \rangle\ xpath \\
valid\langle \mathsf{Con}\ c\ \alpha \rangle \quad xpath = \\
\quad null\ xpath \\
\quad \vee\ (head\ xpath)\ `eq`\ (conName\ c) \\
\quad \wedge\ (null\ (tail\ xpath) \vee valid\langle \alpha \rangle\ (tail\ xpath)) \\
\quad \vee\ (isDos\ (head\ xpath) \wedge valid\langle \alpha \rangle\ xpath)
\end{array}
$$

Note how *valid* is defined over the structure of the argument type t.

For the base types such as Unit, Int, String no validation is performed, since in our XPath definition, the element's content cannot be addressed.

For a type defined in terms of a choice ($\mathsf{Sum}\ \alpha\ \beta$) or a product ($\mathsf{Prod}\ \alpha\ \beta$) of types, function *valid* tests whether or not the XPath expression is valid with respect to the type $\alpha$ or $\beta$.

The interesting case is where an XPath expression is matched against a constructor ($\mathsf{Con}\ c\ \alpha$). The first Axis (*head xpath*) is compared for equality with the constructor's name (*conName c*) by means of the function *eq*. In case of a *Wildcard*, function *eq* succeeds.

$$
\begin{array}{l}
eq :: \mathsf{Axis} \rightarrow \mathsf{String} \rightarrow \mathsf{Bool} \\
eq\ (Axis\ \_\ (QName\ n))\ str = n \equiv str \\
eq\ (Axis\ \_\ Wildcard) \quad \_ \quad = True
\end{array}
$$

Given an Axis that does not match the constructor's name, validation fails with respect to that constructor without validating the rest of the path (*tail xpath*) with respect to the type $\alpha$.

In case an Axis is a descendant-or-self, tested by function *isDos*:

$$
\begin{array}{l}
isDos :: \mathsf{Axis} \rightarrow \mathsf{Bool} \\
isDos\ (Axis\ axisname\ \_) = axisname \equiv Dos
\end{array}
$$

it is tested for validity with respect to the type $\alpha$. and to all its descendant types.

Function *valid* is only guaranteed to terminate for non-recursive data types. If a data type is recursive, the call $valid\langle\alpha\rangle\ xpath$ in the Con case might loop. The actual implementation of *valid* detects recursive data types by maintaining a list of visited constructor names. Since we want to illustrate the generic aspects of the XPath validation problem, we have given the simple version of the code here. The full version is available at `http://www.cs.uu.nl/~rui/`

## 4 Pattern matching XPath expressions

Validating an XPath expression against a schema is particularly useful when an XPath expression is used in another language. In the case of a query language, such as XQuery, or a transformation language, such as XSLT, XPath validation can be used to identify queries or matches that will never succeed. Furthermore, information from the schema can also be used to prevent matching in those subtrees for which it is guaranteed that no matches will occur.

Let us start by defining a generic matching function for an XPath expression. Such a function tests whether an XPath expression addresses a part of a given document.

As an example, we define three possible usages of *compileMatch*:

$$
\begin{array}{l}
compileMatch\langle \mathsf{Book} \rangle\ \texttt{"/Book/English"}\ book \\
compileMatch\langle \mathsf{Book} \rangle\ \texttt{"/Book/German"}\ \ book \\
compileMatch\langle \mathsf{Book} \rangle\ \texttt{"/Book/Spanish"}\ book
\end{array}
$$

Only the first XPath expression matches the document *book*. The second expression, although it is valid with respect to Book, does not match the document *book*. The XPath expression `"/Book/Spanish"` is not valid with respect to Book, thus it is useless to try to match it against any document of type Book.

The function $match\langle \mathsf{t} \rangle$ only inspects a document, if the XPath expression is valid with respect to t. For the type of the generic function *match* we take:

$$match\langle \mathsf{t} :: * \rangle :: \mathsf{XPath} \rightarrow \mathsf{Maybe}\ (\mathsf{t} \rightarrow \mathsf{Bool})$$

A match can be "compiled" by means of *compileMatch*, which returns an error if the *match* fails to validate a path, and otherwise it returns a function that matches the valid path against a document. The generic function *compileMatch* is not defined by means of a typecase construct, but in terms of another generic function. Such a function is called a *generic abstraction* [13].

```
compileMatch⟨t :: ∗⟩ :: (match⟨t⟩)
  ⇒ XPath → t → Bool
compileMatch⟨t⟩ path =
  case match⟨t⟩ path of
    Just matchDoc → matchDoc
    Nothing       → error "Invalid XPath!"
```

Given a valid XPath expression with respect to t, *match* returns a function that matches the given XPath expression with any document of type t. The *match* function is defined in terms of a generic traversal *crushS*. In the next section, *crushS* will also be used to query a document.

```
match⟨t :: ∗⟩ :: (crushS⟨t | Bool⟩)
               ⇒ XPath → Maybe (t → Bool)
match⟨t⟩       = crushS⟨t⟩ (∨) True False
```

We call *crushS* a selective traversal since it navigates a document depending on whether or not the XPath expression is valid with respect to the node's type.

The definition of *crushS* is given in Figure 2. The type of *crushS* is parameterized over two type variables, separated by a vertical bar. Type parameters that appear to the left of the vertical bar are the generic type variables that we have seen before, for example the type parameter t in function *valid*. Type parameters that appear to the right of the vertical bar are called *non-generic* type variables. Such non-generic variables appear in type-indexed functions that are *parametrically polymorphic* with respect to some type variables. We can further specify this type when we use *crushS*, as can be seen in the type signature of the function *match*.

The generic function *crushS*⟨t⟩ takes four arguments: a binary operator, two default values and a path. Given a valid XPath expression, depending on whether or not a node is addressed by a path, *crushS* returns either the first or the second default value. Since a path can address several nodes, a binary operator is used to compose several results.

The behaviour of the function *crushS*⟨t⟩ depends on its type argument t. For a type defined in terms of a choice (Sum $\alpha$ $\beta$) of types, *crushS* tests whether a path is valid with respect to the types $\alpha$ and $\beta$. If a path is valid for the left definition but a given document is defined in terms of the right definition, or vice versa, *crushS* returns the second default value, which in the case of *match* is *False*.

In case of a type defined as a product (Prod $\alpha$ $\beta$) of types, *crushS* tests whether a path is valid with respect to each type argument. *crushS* only computes a pattern matching function for those types that validate the XPath expression. The results of the matching functions are combined with a binary operator, which in the case of *match* is the logic operator or.

Given a type defined as a constructor (Con $c$ $\alpha$), the first Axis of the XPath expression is matched against the constructor name. In case of failure, it is ensured that no matches will occur with respect to the type $\alpha$ thus the subdocument defined by $\alpha$ will not be visited.

A descendant-or-self axis is matched against a constructor and all its descendant. This prevent us to use the type information to optimize the document traversal. However, this situation can be avoid by translating a descendant-or-self axis into a minimal number of child axes [**?**].

The validation succeeds when the entire path is consumed, and the first default value is returned, which in the case of *match* is the value *True*.

## 5 Generically querying a document

XQuery [19] is a typed, functional language for querying XML documents. It uses XPath expressions to address parts of a document. Here is an example of an XQuery expression, which asks for the authors of a book:

```
document("book.xml")//Author
```

where `"book.xml"` is an XML document file. In Generic Haskell we write:

```
compileQuery⟨Book⟩ "//Author" book
```

where *book* is the Haskell value that corresponds to the content of the XML document file. The result of this query is the following list of Author's: [*Author* `"Abiteboul"`, *Author* `"Buneman"`, *Author* `"Suciu"`].

A typed approach to a querying system validates a query and its result. A valid query with a validated result type returns a possibly nonempty set of type-correct results. We use generic programs for validating a query and its result.

An XPath expression may address a set of nodes of any type, so to guarantee result validity a query system has to verify whether or not the addressed nodes are of some expected type.

A query can be defined by means of the previously defined *crushS* function. Result validity can be enforced by specifying the non-generic quantified variable of *crushS* to be the expected type of the query result. In Generic Haskell, it is possible to extend an already existing type-indexed function by adding new cases or by overriding existing ones. Such a function makes use of so-called *default cases* [13]. This default case can be seen as a cast operation, where only values of the expected type will be included in the result.

```
crushS⟨t :: * | a :: *⟩ :: (crushS⟨t | a⟩)
   ⇒ (a → a → a) → a → a
   → XPath → Maybe (t → a)
crushS⟨Unit⟩   _ _ _ _ = Nothing
crushS⟨Int⟩    _ _ _ _ = Nothing
crushS⟨String⟩ _ _ _ _ = Nothing
crushS⟨Sum α β⟩ op e1 e2 xpath =
  case (crushS⟨α⟩ op e1 e2 xpath
       , crushS⟨β⟩ op e1 e2 xpath) of
    (Nothing, Nothing) → Nothing
    (mfa, mfb) → Just (λd → case d of
                         (Inl a) → mapply e2 mfa a
                         (Inr b) → mapply e2 mfb b)
crushS⟨Prod α β⟩ op e1 e2 xpath =
  case (crushS⟨α⟩ op e1 e2 xpath
       , crushS⟨β⟩ op e1 e2 xpath) of
    (Nothing, Nothing) → Nothing
    (mfa, mfb) → Just (λ(a × b) →
                       op (mapply e2 mfa a)
                          (mapply e2 mfb b))
crushS⟨Con c α⟩ op e1 e2 xpath
  | ¬ (eq (head xpath) (conName c))
    ∨ null xpath =
      let md = ifDos xpath (crushS⟨α⟩ op e1 e2)
      in Just (λ(Con a) → mapply e1 md a)
  | null (tail xpath) =
      let md = ifDos xpath (crushS⟨α⟩ op e1 e2)
      in Just (λ(Con a)
                  → op e1 (mapply e1 md a))
  | otherwise =
      let crush = crushS⟨α⟩ op e1 e2
      in case (ifDos xpath crush
              , crush (tail xpath)) of
        (Nothing, Nothing) → Nothing
        (md, mfa) → Just (λ(Con a) →
                          op (mapply e1 md a)
                             (mapply e1 mfa a))
ifDos xpath vA =
  if     isDos (head xpath)
  then case vA xpath of
          Nothing → Nothing
          Just ca  → Just (λ(Con a) → ca a)
  else  Nothing
mapply :: b → Maybe (a → b) → a → b
mapply b mf a = case mf of
                    Nothing → b
                    (Just f) → f a
```

Figure 2: *crushS* function

The function *queryAuthor* extends and specializes the generic traversal *crushS* to return values of type [Author].

```
queryAuthor⟨t :: *⟩ :: (crushS⟨t | [Author]⟩)
   ⇒ ([Author] → [Author] → [Author])
   → [Author] → [Author] → XPath
   → Maybe (t → [Author])
queryAuthor extends crushS
queryAuthor⟨Author⟩ op v1 v2 xpath@(axis : path)
   | null path ∧ eq axis "Author" = Just (:[])
   | otherwise = crushS⟨Author⟩ op v1 v2 xpath
```

The query result is the list of values (XML elements) that the last axis of the matched XPath expression addresses. Given an XPath expression that does not match a value of type Author, *crushS* is used.

In addition to result validity, a query must be valid itself. In our approach, query validity and XPath validity coincide. Thus given an valid XPath expression, query validity is guaranteed.

Just as for a *match* we can "compile" a query to guarantee that only valid queries are performed.

```
compileQuery⟨t :: *⟩ :: queryAuthor⟨t⟩
   ⇒ String → t → [Author]
compileQuery⟨t⟩ path =
  let xpath = toXPath path
  in  case queryAuthor⟨t⟩ (++) [] [] xpath of
        Just res → res
        Nothing → error "Invalid query!"
```

Notice that *queryAuthor* is used with two empty list default values. The first default value is used in case a path does not match a document. The second default value is used when a path matches a value that is not of the expected result type.

## 6 Updating a document

In the previous sections we have defined generic functions that use XPath expressions to match or query a document. XPath expressions are also used to address a set of nodes that have to be transformed or updated, for example in XSLT and UpdateX. UpdateX is an XQuery-Based language for processing updates in XML, which uses XPath expressions [17]. In UpdateX, to double the price of a book, you might write:

```
replace value of
  document("book.xml")/Book/Price
with document("book.xml")/Book/Price * 2;
```

In Generic Haskell this would be:

```
updPrice⟨t :: *⟩ :: (mapS⟨t | Price⟩)
  ⇒ XPath
  → Maybe ((Price → Price) → t → Maybe t)
updPrice extends mapS
updPrice⟨Price⟩ (axis : path)
  | null path ∧ eq p "Price" =
      Just (λu p → Just (u p))
  | otherwise = mapS⟨Price⟩ (axis : path)

compileUpd⟨t :: *⟩ :: updPrice⟨t⟩
  ⇒ String → (Price → Price) → t → Maybe t
compileUpd⟨t⟩ path =
  let xpath = toXPath path
  in case updPrice⟨t⟩ xpath of
       Just upd → upd
       Nothing  → error "Invalid Update!"
```

Figure 3: Updating the price of a book

```
compileUpd⟨Book⟩ "/Book/Price" (*2) book
```

Our approach to updating a document is very similar to querying a document. Instead of using a "type-unifying" traversal *crushS*, the update function uses a "type-preserving" traversal that performs a computation during the traversal of a data structure while preserving its type [12].

The function *mapS* is a generic type-preserving traversal that only visits the nodes addressed by a valid path. Since its implementation is similar to *crushS*, we only give its type signature:

```
mapS⟨t :: * | a :: *⟩ :: (mapS⟨t | a⟩)
  ⇒ XPath → Maybe ((a → a) → t → Maybe t)
```

As for a query, an update has to guarantee path and result validity.

Function *updPrice*, defined in Figure 3, extends and specializes the generic traversal *mapS* for values of type Price. This guarantees that only values of type Price will be updated.

Path validity is guaranteed by "compiling" the update by means of *compileUpd*, see Figure 3.

## 7 Related Work

Although there exist several XML-Haskell data bindings, we are not aware of typed XPath-Haskell data bindings.

The *Haskell XML Toolbox* [16] and *HXML* [7] translate XML documents to values of a predefined tree type. Both include an XPath filter that selects part of a document, and a query result is always a value of a universal type. The advantage of this approach is that it is simple, but it is untyped and precludes the possibility of partially checking the correctness of parts of the program before actually starting to process data.

HaXml [20] translates DTDs to algebraic data types in a structural fashion. Furthermore, HaXml includes a component Xtract for processing queries. Xtract is a grep-like tool for XML documents, loosely based on the XPath and XQL query languages. However, Xtract does not perform any validation against a document's DTD, nor does it attach a DTD to its output. We perform validation by checking a path against the Haskell counterpart of the schema/DTD.

WASH [18] is a family of embedded domain specific languages for programming Web applications in the functional language Haskell. The Haskell type class system is used to check for correctness of constructed values by translating the information from the DTD to Haskell classes. However a query system has not been implemented.

Lämmel and Peyton Jones [11] describe a design pattern for writing programs that traverse data structures built from rich mutually-recursive data types. A document is queried by means of pattern matching rather than by using an XPath expression. Their approach is supported by the GHC implementation of Haskell. However, generic traversal functions (obviously) do not avoid subterms that are known to be irrelevant by virtue of type information. We expect that our generic validation, matching and update functions can be expressed in the 'scrap your boilerplate' approach as well.

The literature on special-purpose XML programming languages such as XQuery [19], $\mu$XQ [6] and CDuce [2] is substantial. Such languages offer specific functionalities, such as powerful pattern matching, regular expressions and precise type inference for patterns and error localization.

A different approach to address parts of a document is to use regular expression patterns. There has been some work in extending Haskell with regular expression patterns. Harp [3] is a lightweight implementation of regular expression patterns that fits seamlessly with Haskell pattern matching. However since this approach has only been developed for lists, it cannot be used to query or match data types generated by a typed Haskell-XML data binding. XHaskell [14] is an extension of Haskell with regular expression types, regular expression pattern-matching and semantic-subtyping in style of XDuce. The key idea of this approach is that explicit type annotations are translated into type class constraints.

## 8 Conclusions and future work

Using Haskell as the target language for an XML data binding offers all the advantages of using a higher order programming language with a powerful type system. Haskell's type checker can be used to statically guarantee XML validity.

In order to exploit these advantages it is crucial to use a type-preserving data binding. The obvious disadvantage of a type-preserving data binding compared with a non-typed data binding is its lack of generality. This disadvantage is circumvented by using generic programs that work on arbitrary data types generated by the data binding.

We give a simple and straightforward Haskell data binding for a subset of XPath. A translated XPath expression is a value of type XPath and it is guaranteed to be well-formed by the Haskell compiler. However, we cannot use the Haskell compiler to guarantee that an XPath expression conforms to an XML Schema, neither can it be used to infer the type of the set of nodes an XPath expression addresses.

Since Generic Haskell programs are defined over the structure of an argument type, we can check validity of an XPath expression with respect to a given type. Although our approach is not static, we can check validity very early when running the program, and before actually using the path for matching in some specific data structure.

We have given generic functions for querying and updating XML documents, while ensuring that the given XPath expression addresses a nonempty set of nodes (Query validity) and the nodes addressed are of the expected type (Result validity). Moreover, for some XPath expressions, these functions use information present in a schema to prevent visiting subtrees that are guaranteed to never contain matches.

Preliminary investigations into generic functions for implementing XPath's filter predicates are promising. Future work may involve extending the XPath translation to cover the complete set of Axes.

A different approach to mapping XPath to Haskell would be to define an XPath type parameterized with the document type of the argument to which the XPath is applied. We are currently investigating the possibility to implement such an XPath type by means of a type-indexed data type [8]. The obvious advantage of such an implementation would be that the Haskell type system gives validity for free.

**Acknowledgements.** Frank Atanassow, Arthur Baars, Andres Löh, Karina Olmos, Alexey Rodriguez, and three anonymous referees commented on previous versions of this paper. The first author is supported by Fundação para a Ciência e a Tecnologia (SFRH/BD/11142/2002).

## References

[1] F. Atanassow, D. Clarke, and J. Jeuring. UUXML: A type-preserving XML schema-haskell data binding. In B. Jayaraman, editor, *PADL'04*, volume 18-19 of *Lecture Notes in Computer Science*, pages 71–85. Springer-Verlag, 2004.

[2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming 2003*, volume 38, pages 51–63. ACM Press, August 25-29 2003.

[3] N. Broberg, A. Farre, and J. Svenningsson. Regular expression patterns. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, volume 39, pages 67–78. ACM Press, 2004.

[4] J. Clark and S. DeRose. XML path language. W3C Recommendation. Available from `http://www.w3.org/TR/xpath`, November 1999.

[5] Dave Clarke, Johan Jeuring, and Andres Löh. The Generic Haskell User's Guide – Beryl release. Technical Report UU-CS-2002-047, Utrecht University, 2002. Also available from `http://www.generic-haskell.org/`.

[6] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types for path correctness of XML queries. In *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 126–137. ACM Press, 2004.

[7] J. English. HXML. Available from `http://www.flightlab.com/~joe/hxml`, 2001.

[8] R. Hinze, J. Jeuring, and A. Löh. Type-indexed datatypes. In E.A. Boiten and B. Möller, editors, *Proceedings of the 6th Mathematics of Program Construction Conference*, volume 2386 of LNCS, pages 148–174, 2002.

[9] 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.

[10] S. Peyton Jones et al. Haskell 98 – A non-strict, purely functional language. Available from `http://www.haskell.org/onlinereport`, February 1999.

[11] R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the International Conference on Functional Programming (ICFP 2004)*, volume 39, pages 244–255. ACM Press, September 2004.

[12] R. Lämmel, J. Visser, and J. Kort. Dealing with large bananas. In J. Jeuring, editor, *Proceedings of Workshop on Generic Programming'2000, Technical Report, Universiteit Utrecht*, pages 46–59, 2000.

[13] A. Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.

[14] K. Zhuo Ming Lu and M. Sulzmann. An implementation of subtyping among regular expression types. In *Proceedings of The Second ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *LNCS*, pages 57–73. Springer-Verlag, 2004.

[15] K. Zhuo Ming Lu and M. Sulzmann. XHaskell: Regular expression types for haskell Haskell. Technical Report TRC9/04, National University of Singapore, 2004. Also available from `http://www.comp.nus.edu.sg/~sulzmann/`.

[16] M. Schmidt. Design and implementation of a validating XML parser in Haskell. Available from `http://www.fh-wedel.de/~si/HXmlToolbox`, 2002.

[17] G. M. Sur, J. Hammer, and J. Siméon. UpdateX - an XQuery-based language for processing updates in XML. In *International Workshop on Programming Language Technologies for XML (PLAN-X 2004)*, pages 244–255, January 2004.

[18] P. Thiemann. A typed representation for HTML and XML documents in haskell. *Journal of functional programming*, 12(5):435–468, 2002.

[19] P. Walder. XQuery: a typed functional language for querying XML. In *Proceedings of the 4th International School on Advanced Functional Programming, AFP 2002*, pages 188–212, August 19-24 2002.

[20] M. Wallace and C. Runciman. Haskell and XML: generic combinators or type-based translation? In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 148–159. ACM Press, 1999.

## A Extending XPath with predicates

The node-set addressed by an XPath can be filtered by means of a predicate. For example, a filter can be added to the query, defined in Section 5, to select only the second author of a Book by writing:

$$compileQuery\langle \mathsf{Book} \rangle \texttt{ "//Author[2]" } book$$

In the XPath Haskell data binding described in this paper we have not shown how to implement XPath filters. Adding filters to the data binding is not difficult. We will show how to implement the *proximity position* predicate defined in the XPath standard [4]. The *proximity position* of a member of a node-set with respect to an axis is defined to be the position of the node in the node-set.

The XPath subset definition given in Section 3 can be extended with a position predicate:

$$\begin{aligned}&\textbf{data } \mathsf{Axis} = Axis\ \mathsf{AxisName\ NameTest\ Predicate}\\&\textbf{data } \mathsf{Predicate} = Position\ \mathsf{Int}\\&\qquad\qquad\quad \mid None\end{aligned}$$

To filter a result by means of a position predicate, it suffices to extend the function *crushS* defined in Figure 2 with an extra case for values of type $[\alpha]$.

$$\begin{aligned}&crushS\langle[\alpha]\rangle\ \_\ \_\ \_\ \_\ [\,] = Nothing\\&crushS\langle[\alpha]\rangle\ op\ e_1\ e_2\\&\quad xpath@((Axis\ l\ n\ (Position\ i)) : \_) =\\&\qquad \textbf{case } crushS\langle\alpha\rangle\ op\ e_1\ e_2\ xpath\ \textbf{of}\\&\qquad\quad Nothing\ \rightarrow Nothing\\&\qquad\quad (Just\ fa) \rightarrow Just\ (\lambda doc \rightarrow\\&\qquad\qquad\qquad \textbf{if}\quad i < (length\ doc)\\&\qquad\qquad\qquad \textbf{then } fa\ (d\ !!\ (i-1))\\&\qquad\qquad\qquad \textbf{else}\ \ e_2)\\&crushS\langle[\alpha]\rangle\ op\ e_1\ e_2\ xpath =\\&\quad \textbf{case } crushS\langle\alpha\rangle\ op\ e_1\ e_2\ xpath\ \textbf{of}\\&\qquad Nothing\ \rightarrow Nothing\\&\qquad (Just\ fa) \rightarrow Just\ (foldr\ (op.fa)\ e_2)\end{aligned}$$

The extended definition is easy to understand: given an Axis with a position predicate filter and a valid XPath expression, *crushS* only traverses the values of a list after filtering with the predicate; if the given Axis has no position predicate *crushS* traverses every value of a list.

Other predicates and their abbreviations such as //Author [*last* ()], which selects the last Author of the node-set, or /Book [Author = `"Abiteboul"`], which selects Books of the node-set that have one or more Author with name `"Abiteboul"`, can be implemented in a similar fashion.