Scrap your Boilerplate with XPath-like Combinators

Ralf Lämmel

Microsoft, Data Programmability Team, USA

Abstract

XML programming involves idioms for expressing 'structure shyness' such as the descendant axis of XPath or the default templates of XSLT. We initiate a discussion of the relationships between such XML idioms and generic functional programming, while focusing on the (Haskell-based) 'Scrap your boilerplate' style of generic programming (SYB). This work gives insight into mechanisms for traversal and selection. We compare SYB and XSLT. We approximate XPath in SYB. We make a case for SYB's programmability, when compared to XPath's fixed combinators. We allude to strengthened type checking for SYB traversals so as to reject certain, trivial behaviors.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.2.13 [*Software Engineering*]: Reusable Software

General Terms Design, Languages

Keywords XML programming, Generic functional programming

1. Introduction

Our point of departure is generic functional programming [8, 4] with its dedication to forms of polymorphism that admit functionality to be applicable to a large class of types. Different 'benchmark' problems appear in the generic programming literature, e.g., generic equality, read and show. With an eye on XML programming, we are interested in 'structure shyness': focus on the *relevant* data parts of richly structured data, find these parts and compose behavior over them. The 'Scrap your boilerplate' literature (SYB, [14]) works with the 'increase salary' benchmark (or the 'paradise' benchmark): given a datum with the organizational structure of a company, increase *everyone's* salary by a factor. Fig. 1 shows a company with nested departments; the salaries of all employees are subject to a raise. If we assume Haskell datatypes for the data in Fig. 1, then it is trivial to implement the paradise benchmark as a Haskell function in SYB style:

```
paradise :: Data a => Float -> a -> a    -- Type inferrable
paradise factor = everywhere (mkT (\(S s) -> S (s*factor)))
```

S is the constructor for salary terms. `everywhere` is a traversal scheme. `mkT` makes a transformation from the identity function updated in a type-specific case; here: a lambda function that rewrites salary terms. The `Data a =>` ... constraint on the polymorphic input type of `paradise` expresses that the function is applicable to values of any 'traversable' (say, generic-programming-enabled) type.

POPL'07 January 17–19, 2007, Nice, France.
Copyright © 2007 ACM 1-59593-575-4/07/0001...$5.00

```
<company>
  <dept> <!-- Top-level department -->
    <name>SYB technology</name>
    <manager>
      <person>Ralf</person>
      <salary>4288</salary>
    </manager>
    <subunits>
      <employee> <!-- Subordinate employee -->
        <person>Joost</person>
        <salary>4237</salary>
      </employee>
      <dept> <!-- Subordinate department -->
        <name>SYB goes XML</name>
        <!-- ... Rest of department ... -->
      </dept>
    </subunits>
  </dept>
  <!-- ... Further top-level departments ... -->
</company>
```

Figure 1. A sample company

```
<xsl:stylesheet>
  <xsl:param name="factor"/>
  <xsl:template match="salary">
    <xsl:copy>
      <xsl:value-of select=". * $factor"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Figure 2. An XSLT program for the paradise benchmark — The first XSLT template matches on salary nodes and multiplies the salary value by the given factor. The second template applies *otherwise*; it models *recursion into and reconstruction of nodes*.

```
static void Paradise(double factor, XmlDocument c) {
  foreach (XmlElement x in c.SelectNodes("//salary"))
    x.InnerText =
     (Double.Parse(x.InnerText) * factor).ToString();
}
```

Figure 3. 'Paradise' in C# + DOM/XPath — The XPath expression "`//salary`" is evaluated by `SelectNodes` resulting in a node set. The `foreach` loop iterates over the node set so that all salary nodes are updated by an assignment to `InnerText`.

While XML programming and generic functional programming have emerged independently of each other, it turns out that they involve similar problems and idioms. XSLT 1.0 [26, 9], a prominent XML programming language, handles the paradise benchmark easily; cf. Fig. 2; likewise for (ancient) DOM-/XPath-based APIs [24, 25] for XML programming; cf. Fig. 3, or for modern XML programming APIs such as LINQ to XML [18]; cf. Fig. 4. Static typing is important in generic functional programming, and it is also coming along for XML programming, e.g., in XPath 2.0 and thereby in XQuery 1.0 and XSLT 2.0 [28, 30, 27, 29].

```
static void Paradise(double factor, XElement c) {
  foreach (var e in c.Descendants("salary"))
    e.ReplaceContent((double)e * factor);
}
```

Figure 4. 'Paradise' in C# + LINQ to XML — The invocation of the `Descendants` method (lazily) returns all descendants with the given element name, `salary`. Again, the `foreach` loop iterates over the node set so that all salary nodes are updated by `ReplaceContent`.

Contributions of the paper We start a discussion of the relationships between problems and idioms in XML and generic functional programming. This is the first attempt of this kind. Specifically:

- We discover that the XSLT and SYB styles of programming are idiomatically similar, in some respects; cf. §3.
- We manage to approximate key elements of 'downward XPath' as SYB-style generic function combinators; cf. §4.
- Based on examples, we suggest that the XPath-like set of axes should be enriched by SYB-inspired variations; cf. §5.
- Inspired by XPath, we enrich the SYB setup in a way that traversed terms carry parental and sibling context; cf. §6.
- We show how to use the Haskell type system to perform extra sanity checks on SYB- and XPath-like traversals; cf. §7.

Road-map of the paper

- §2 takes an inventory of DOM, SYB, XPath and XSLT idioms.
- §3 — §7 develops the aforementioned contributions.
- §8 describes related work.
- §9 concludes.

Source code is available at `http://www.cwi.nl/~ralf/popl07`.

2. Idiomatic inventory

XPath-like node selection XML programming setups may facilitate XPath [25] (or XPath-like) query axes for node selection. Typical axes are these: child, parent, sibling and descendant. The code in Fig. 3 (DOM/XPath) indeed devises a string-encoded XPath query, to select salary nodes along the descendant axis. The code in Fig. 4 (LINQ to XML) is idiomatically equivalent, but the API at hand favors XPath-like query combinators (say, methods) over string encoding. It should be noted that the XSLT variation on the paradise benchmark (cf. Fig. 2) does *not* carry out XPath-like node selection. Instead, XSLT's template mechanism is put to work for achieving genericity, as we will discuss shortly.

Transformations vs. queries In the SYB context, there exists the dichotomy transformations vs. queries. A transformation 'rewrites' the input data. A query 'selects' and 'collects' or 'aggregates' parts of the input data. Transformations and queries are often composed. For example, a source-code transformation may perform designated analyses ('queries') in due course.

The paradise benchmark is a clear-cut example of a transformation: the global transformation of a company boils down to many local changes of salary nodes. Other transformations may be less 'shape-preserving'; they may rewrite the input data more arbitrarily using more profound strategies and larger scopes.

Fig. 5 shows a clear-cut example of a query: the sample totals all salaries in a company. The LINQ to XML code uses (again) the descendant axis to find all salary nodes; this list is then fed to an aggregation function `Sum`. The SYB code uses a traversal scheme, `everything`, to identify all salary nodes and aggregate them by the binary operation "+" as the traversal walks the term (i.e., *without* going through an explicit list of salary nodes).

Haskell + SYB

```
total :: Data a => a -> Float     -- Type inferrable
total = everything (+) (0 `mkQ` (\(S s) -> s))
```

C# + LINQ to XML

```
static double Total(XElement c) {
  return c.Descendants("salary")     // Select nodes
          .Select(s => (double)s)    // Coerce to doubles
          .Sum();                    // Sum up doubles
}
```

Figure 5. Total over all salaries in a company

SYB's transformations and queries The terms transformation and query are particularly meaningful in pure, typed functional programming; generic transformations and queries are functions that are polymorphic in their argument type; transformations *preserve* the type in the result; queries compute a result of a *fixed* type. Accordingly, SYB provides the following two type schemes:

```
type GenericT   = forall a. Data a => a -> a  -- Transformations
type GenericQ r = forall a. Data a => a -> r  -- Queries
```

Genericity is expressed by these types through universal quantification with a `Data` bound for generic-programming-enabled datatypes.

Mutable node sets When XPath is embedded into an *imperative* OO language, then transformations can be expressed by iteration over query results (i.e., sets or lists of XML nodes) such that each single node may be transformed separately. This was illustrated for both, DOM/XPath and LINQ to XML. Hence, in such an impure setup, *generic programming is mainly concerned with queries* as opposed to transformations. For instance, LINQ to XML queries are of the following type:

```
IEnumerable<XElement>
```

Here `XElement` is the OO class for mutable XML trees of LINQ to XML; `IEnumerable` denotes the .NET generics type for (potentially lazy) lists. A typical transformation was shown in Fig. 4.

SYB-like traversal control XPath style can be summarized as follows: 'queries are composed from predefined axes'. By contrast, SYB style essentially involves 'programmable traversal'. One defines general-purpose traversal schemes and problem-specific traversals as polymorphic, recursive functions using a fundamental primitive for one-layer traversal, `gfoldl`, which is akin to the folklore left-associative list fold, except that it folds over immediate subterms ('children') of a term, and has access to the constructor.

The SYB code for the paradise benchmark uses the transformation scheme `everywhere`; The SYB code for the query that totals all salaries uses the query scheme `everything`. Let us reconstruct both schemes in the sequel.

```
everywhere :: GenericT -> GenericT
everywhere f = f . gmapT (everywhere f)
```

The transformation `everywhere` f fully recurses into a given term to apply f to all of its subterms in bottom-up order. The scheme `everywhere` uses the auxiliary `gmapT` combinator to transform the immediate subterms of a term.

```
everything :: (r -> r -> r) -> GenericQ r -> GenericQ r
everything k f x = foldl k (f x) (gmapQ (everything k f) x)
```

The query `everything` k f fully recurses into a given term to apply f to all of its subterms and aggregate the intermediate results using the binary operation k. (The result type is a list or set type, or some other 'monoidal' type that comes with an associative operation k.) The scheme `everything` uses the auxiliary `gmapQ` combinator to map over the immediate subterms of a term.

For brevity, we skip `gfoldl` and the definitions of `gmapT` and `gmapQ` in terms of this primitive. One can think of `gmapT` and `gmapQ` as being defined as follows, when applied to a term of shape $C\ t_1\ \cdots\ t_n$ (with C as constructor, and $t_1, \ldots, t_n$ as children):

$$\texttt{gmapT}\ f\ (C\ t_1\ \cdots\ t_n) = C\ (f\ t_1)\ \cdots\ (f\ t_n)$$
$$\texttt{gmapQ}\ f\ (C\ t_1\ \cdots\ t_n) = [f\ t_1, \ldots, f\ t_n]$$

SYB-like type dispatch SYB configures generic traversals by type-specific cases. To this end, a generic *default* is specialized in one or more specific types. In our examples, we either change or select salaries; cf. `mkT` vs. `mkQ`:

```
paradise factor = everywhere (mkT (\(S s) -> S (s*factor)))
total = everything (+) (0 ‘mkQ‘ (\(S s) -> s))
```

`mkT` specializes the identity function in one type; `mkQ` specializes a constant function in one type. Type-specific cases can also be chained by the operators `extT` and `extQ`. The following definitions derive single-case specialization from the chainable operators:

```
mkT f     = id ‘extT‘ f
r ‘mkQ‘ f = const r ‘extQ‘ f
```

These chainable forms are eventually defined in terms of a fundamental primitive for type cast, `cast`, which reports success vs. failure of cast through a `Maybe` value. Hence, the chainable forms attempt to *cast the input term to the argument type of the type-specific case*; the latter is applied in case of success, and the generic default is applied otherwise. Thus, for $\mathsf{ext} \in \{\mathsf{extT}, \mathsf{extQ}\}$:

```
(g ‘ext‘ s) x = case cast x of
                  Just x’ ->  s x’
                  Nothing ->  g x
```

For completeness’ sake, these are the relevant types:

```
mkT   :: Typeable x => (x -> x) -> GenericT
mkQ   :: Typeable x => r -> (x -> r) -> GenericQ r
extT  :: Typeable x => GenericT -> (x -> x) -> GenericT
extQ  :: Typeable x => GenericQ r -> (x -> r) -> GenericQ r
cast  :: (Typeable b, Typeable a) => a -> Maybe b
```

`mkT` and `mkQ` *construct* a generic function, while `extT` and `extQ` *transform* a generic function (to incorporate another type-specific case). The `Typeable` bound enables `cast`; the typeclass `Typeable` is a superclass of the typeclass `Data`.

In XML programming, the notion of ‘specific type’ may (often) correspond to ‘elements with a specific name’ such as elements for companies, departments, managers and salaries. *XML transformations and queries routinely dispatch on element names.*

XSLT-like recursive templates XSLT is biased towards (but not restricted to) transformations as opposed to queries [10]. The XSLT style of transformation is fundamentally different from DOM/XPath. That is, the predominant form of genericity in XSLT programs is modeled by XSLT’s mechanism for (default) templates. (XPath may still be used in individual XSLT templates for selection.) For clarity, we repeat the default template that is used in the XSLT code for the paradise benchmark; it applies whenever the specific template for salaries does not match.

```
<xsl:template match="@*|node()">
  <xsl:copy>
    <xsl:apply-templates select="@*|node()"/>
  </xsl:copy>
</xsl:template>
```

XSLT style is purely functional [9]: a template copies the matched tree or recurses into (selected parts of) it. Hence, tree selection (‘querying’) and processing (‘transformation’) are amalgamated. A common form of recursive function application is to ‘apply all templates’, but templates can also be applied more selectively.

```
paradise factor = stylesheet
 where
   stylesheet        :: GenericT
   stylesheet        =  applyTemplates ‘extT‘ transformSalary
   transformSalary   :: Salary -> Salary
   transformSalary   =  \(S s) -> S (s*factor)
   applyTemplates    :: GenericT
   applyTemplates    =  gmapT stylesheet
```

Figure 6. ‘Paradise’ in Haskell + SYB in simulated XSLT style

3. XSLT style vs. SYB style

XSLT style is similar to SYB style. In particular, XSLT’s templates with problem-specific match conditions are similar to the type-specific cases in SYB. A template with a match condition for an element name, e.g., `match="salary"`, corresponds to an SYB-like, type-specific case that is universally applicable to the specified type, say the type of salaries.

In general, XSLT’s match conditions may be organized as general logical formulas over various observable axes of nodes. We contend that this style exposes a flavor of operating on a universal representation. In principle, SYB has access to similar, reflective power, but native SYB style uses type dispatch to organize matches. Each type-specific case may narrow down the match, in which case result types need to be wrapped in `Maybe` (or another `MonadPlus`) and designated traversal schemes are to be chosen.

XSLT’s default templates are reminiscent of SYB’s recursion with the `gmap` family. The `copy` idiom preserves the enclosing element — just like `gmapT` preserves the outermost constructor. The `apply-templates` idiom maps the templates over all children — just like `gmapQ` maps over the list of immediate subterms. The separation of the two XSLT idioms would be notably hard to separate in the Haskell type system without escaping to a universal type.

Fig. 6 shows XSLT-style SYB code for the paradise benchmark. This experiment triggers additional observations. First, both the native SYB code and the XSLT-style SYB code are statically type-checked; by contrast, XSLT 1.0 does not provide static type checking; XSLT 2.0 will comprise schema-aware processing based on and enhancing the static typing notion for XPath 2.0. At this point, though, template application, as in default templates, in particular, seems to rely, at least in part, on runtime type checking. Second, the XSLT-style SYB code *explicitly* composes all templates through the chaining operator `extT`, whereas XSLT provides the convenience of juxta-positioning all templates in a stylesheet, while XSLT language rules prioritize templates. Third, the XSLT-style SYB code clarifies that an XSLT stylesheet ties up recursion in itself, whereas the native SYB code leaves it to a reusable traversal scheme to tie up recursion. In other words, XSLT style dictates ad-hoc recursive traversal, whereas SYB style recommends a separation of concerns: recursive *traversal schemes* vs. *node processors*. This is a heritage of term-rewriting strategies a la Stratego [23].

4. Elements of downward XPath in SYB

Let us approximate key elements of downward XPath in SYB so that we can compare XPath-like node selection with SYB style. (By no means, we claim to reconstruct downward XPath. We anyhow neglect ‘XML-isms’ in the following discussion: attributes, text nodes, XML comments, processing instructions, whitespace.)

We associate XPath axes (or query patterns) with Haskell combinators, which map a given term to *a list of terms*. Using the list monad or other combinators for list processing, one can chain up such filters. It is important to note that XPath [25, 28] comprises rules that define when *node lists vs. node sets* are used. We glance over this (important) detail, and admit that SYB style may be incapable of aligning with XPath’s identity factorization unless we were modeling term *identities* in our functional setup.

XPath's self axis, i.e., '.', selects the input. In Haskell:

```
self :: x -> [x]
self x = [x]
```

XPath's unconstrained child axis, i.e., '*' or 'child::*', selects all child elements. This axis shows up in XSLT default rules, but not much elsewhere. It is intrinsically untyped, and hence it does not occur in native SYB programming, but here is a reconstruction:

```
anyChild :: GenericQ [Any]
anyChild = gmapQ Any
```

We use a universe of all castable terms:

```
data Any = forall x. Typeable x => Any x
```

Now consider XPath's child axis, when constrained by an element name, such as in 'child::salary'. In Haskell, `anyChild` is filtered by type (in place of an element name).

```
childOfType :: Typeable r => GenericQ [r]
childOfType = typeFilter . anyChild
  where
    typeFilter :: Typeable r => [Any] -> [r]
    typeFilter = map fromJust . filter isJust . map unAny
      where
        unAny (Any x) = cast x
```

It is straightforward to post-compose `childOfType` with a normal (monomorphic) filter so as to achieve the affect of an XPath predicate *p* such as in 'child::salary[*p*]'.

Let us now look at XPath's descendant axis. This time, we use a recursive traversal scheme, `everything`, instead of the one-layer traversal with `gmapQ` in the case of the child axis. Again, we show both, the unconstrained and the type-constrained descendant axis, where the latter one does not facilitate the former one so as to avoid building a huge, transient list of untyped terms:

```
anyDescendant :: GenericQ [Any]
anyDescendant = everything (++) anyChild

descendantOfType :: Typeable r => GenericQ [r]
descendantOfType = everything (++) childOfType
```

The 'type of interest' is normally specified by a type annotation. For instance, given a company, *c*, all its descendant departments (cf. datatype `Dept`) are obtained as follows:

```
descendantOfType c :: [Dept]
```

Two asides are worth noting. First, the aggregation of lists of descendants in terms of '++' is notoriously inefficient; we could employ plain 'list consing' by switching to a higher-order accumulation scheme. Second, the above examples only cover queries. One can easily provide a similar suite for 'axes-controlled transformations' by a mechanical adaptation of the code for the query operators; both suites can even be overloaded.

5. SYB-inspired enrichment of XPath

XPath is a fixed domain-specific selector language for XML. The existing XPath operators may end up being inconvenient (in terms of expressivity) or computationally suboptimal. The programmability of traversal schemes in a generic programming framework allows for tailored schemes for queries and transformations.

Imagine a salary raise should be restricted to certain departments. Here are some sets of departments in the company *c*:

top-most All *top-most* departments.
bottom-most All *bottom-most* departments.
non-top-most All but the *top-most* departments.
non-bottom-most All but the *bottom-most* departments.
inner All non-top-most and non-bottom-most departments.

The descendant axis gives us *all* departments and we would need to engage in filtering out those of interest. This is inconvenient and computationally expensive. For instance, in terms of the API of LINQ to XML, one can query the top-most departments as follows:

```
c.Descendants("dept").Where(x => ! x.Descendants("dept").Any())
```

While XPath itself is a fixed language, one can argue that an API could simply be extended to support new axes. However, a computationally inexpensive definition requires access to a low level of traversal. The SYB-based query scheme for top-most descendants uses `gmapQ` and *ties up recursion in a designated manner*:

```
topMostOfType :: Typeable r => GenericQ [r]
topMostOfType x = case cast x of
                    Just x' ->  self x'
                    Nothing ->  recurse
  where
    recurse = concat (gmapQ topMostOfType x)
```

While SYB is more flexible than XPath, the choice of a fixed DSL for selection offers advantages, too. That is, an XPath processor can engage in domain-specific optimizations, while free-wheeling SYB style is only accessible to optimizations of an Haskell implementation. SYB style relies on higher-order style, non-trivial polymorphism and overloading — thereby ruling out some optimizations.

6. XPath-inspired enrichment of SYB

SYB is conceptually aimed at traversal, say 'going downward' in terms. Ancestor terms may be passed around actively by the programmer, e.g., on the grounds of the reader/environment monad. By contrast, XPath readily provides parent and sibling axes because these are important elements of XML programming.

A DOM-like representation of XML trees provides access to parent and sibling axes by maintaining links from each node to its parent. A pure functional language like Haskell needs to go a different route. In the following, SYB is enriched in a way that traversed terms carry around context for the parent and sibling axes. Traversal into terms remains to be the primary idiom. (Hence, the proposed technique does not aim at arbitrarily navigable or editable trees; cf. the discussion of 'the zipper' and friends in the related work section.) The context of a term comprises its parent term (including the context of the parent) and the position of the term in the ordered child list of the parent:

```
data Context = Root
             | Node { parent :: InContext Any, position :: Int }
```

The following datatype models 'terms in their context':

```
data InContext a = InContext Context a
contextOf (InContext ctx _) = ctx   -- project to context
noContext (InContext _ x) = x       -- project to plain term
```

SYB needs to be generalized in a systematic manner to use 'terms in context' instead of plain terms. Most notably, the types of (generic) transformations and queries evolve as follows:

```
type GenericT   = forall a. Data a => InContext a -> a
type GenericQ r = forall a. Data a => InContext a -> r
```

The `gmap` family of one-layer traversal combinators also needs to be lifted to terms in context. These new `gmap` combinators place each child in its context before transforming or querying it.

Now, the definition of the ancestor axis is straightforward:

```
anyAncestor :: GenericQ [Any]
anyAncestor x = anyAncestor' (contextOf x)
  where anyAncestor' Root = []
        anyAncestor' c = noContext (parent c)
                       : anyAncestor' (contextOf (parent c))

ancestorOfType :: Typeable r => GenericQ [r]
ancestorOfType = typeFilter . anyAncestor
```

Looking up siblings is equally simple.

```
class IsReachableFrom x y
instance (ReachableFrom () xs (y,()), IsElem x xs) => IsReachableFrom x y

-- Determine all types ys reachable from the types xs; ps serves as an accumulator position.
class ReachableFrom ps ys xs | ps xs -> ys
instance ReachableFrom ps ps ()
instance (ChildTypes (x,xs) ns, Diff ns ps ns', Append ps ns' ps', ReachableFrom ps' ys ns') => ReachableFrom ps ys (x,xs)

class ChildTypes x y | x -> y
instance ChildTypes Prelude.Char ()     -- primitive type
instance ChildTypes Prelude.Float ()    -- primitive type
instance ChildTypes [x] (x,([x],()))    -- lists
instance ChildTypes () ()               -- empty product
instance (ChildTypes x ys, ChildTypes xs ys', Append ys ys' ys'', Nub ys'' ys''') => ChildTypes (x,xs) ys''' -- non-empty products
```

Figure 7. Reachable types (We use folklore functions, `Append`, `Diff`, `IsElem` and `Nub`, but lifted to the type level.)

7. Sanity checks for SYB traversals

Normal Haskell type checking alone does not establish various interesting properties of SYB traversals. It is perhaps unsurprising that we are not prevented from defining non-terminating traversals. For the application of predefined traversal schemes, however, we may want to statically rule out certain trivial behaviors such as (i) a query is bound to return the empty node set; (ii) a transformation is bound to fail; (iii) a transformation is bound to behave like the identity function. In the following, we illustrate a technique to perform corresponding checks within the Haskell type system.

For instance, the following query always returns an empty list:

```
descendantOfType c :: [Company]
```

That is, terms of type `Company` cannot possibly occur directly or transitively under terms of type `Company` (such as c). This fact follows from the datatype declarations for company terms; the type `Company` only occurs at the root of terms.

Given an application of `descendantOfType` with input type x and result type $[y]$, the idea is now to check that y is reachable from x. This idea is very similar to the type-checking rules for descendants and friends in the XPath 2.0 formal definition [29]. Instead of engaging in an offline program analysis, we can use type-level programming [3, 17, 20, 19, 12]. We keep the definition of `descendantOfType`, while we simply add a constraint to its type:

```
descendantOfType :: (Data x, Typeable r, IsReachableFrom r x)
                 => x -> [r]
```

The reachability constraint stands for the transitive closure of the parent-child-type relation *at the type level*. Each axis or query combinator requires a similar, designated constraint. For instance, the 'descendant-*or-self*' axis requires the *reflexive* and transitive closure of the parent-child-type relation.

Fig. 7 shows an encoding of `IsReachableFrom`. (Mutually recursive, regularly polymorphic datatypes are covered.) The definition relies on *type-level* access to the child types of types. To this end, we assume that user-defined types are reified through a type-level relation, `ChildTypes`, between parent type and the heterogeneous list [12] of child types (using nested, right-associative, explicitly terminated, binary products). For instance:

```
-- User-defined algebraic datatypes
data Company = C [Dept]
data Dept    = D Name Manager [SubUnit]

-- Type-level representation
instance ChildTypes Company ([Dept],())
instance ChildTypes Dept (Name,(Manager,([SubUnit],())))
```

The type-level representation can be mechanically derived from the original types without ado, e.g., by means of program generation with Template Haskell [22].

8. Related work

HaXML's content filters The seminal paper on HaXML [31] delivered generic combinators for content filtering with coverage of the downward-axes of XPath including variations on the descendant axis. The HaXML paper suggested that one must make a decision to either go for generic, *untyped* combinators or for a type-based translation so that XML programming is eventually carried out as 'normal' (i.e., non-generic) programming on datatypes. The present paper clarifies that no such draconian choice is necessary. We have delivered strongly typed XPath-like query operators.

XSLT-like recursion It is a natural question to ask whether a 'proper' functional language can do a better job in expressing XSLT transformation scenarios than XSLT itself. Several such efforts were carried out for Scheme [13, 11, 21]. Typically, these efforts focus on recursion, pattern matching and replacement; they are indeed idiomatically close to XSLT as opposed to HaXML-like content filtering or XPath-like querying. A sophisticated project is SXML/SXSLT [13, 11] — not just because of its coverage of full XML, but also because it comes with a pre-post-order scheme for recursion that can even accommodate local stylesheets, which are added in the process of descending into XML trees.

The Zipper et al. Our technique for accessing the parent and sibling axes is reminiscent of the 'zipper' technique [7], which is based on a data structure that enables navigation and editing on terms through 'pointer reversal'. There also exists a variation on the zipper, the web [5]. Furthermore, advanced generic functional programming with *type-indexed data types* has been put to work to derive the zipper data structure for a given data type [6].

The simplicity of our technique rests on a deliberate restriction. That is, parent and position information provides read-only context along traversal. Without this restriction, we were mixing purely functional SYB style with mutable updates. Our approach immediately works for *systems of mutually recursive datatypes*; the aforementioned approaches have not been described in such generality.

HXML, a non-validating XML parser written in Haskell [2], provides a notion of navigable trees, which however deals with the generic tree representation only, whereas our development integrates 'trees in context' with strongly typed generic programming.

Adaptive programming There is an established generic programming notion that is closely related to our efforts: adaptive programming (AP, [16]). In [15], we had already compared AP with strategic programming (SP), which is similar to SYB, but there are some XML-specific issues that are worth adding. The specifications in AP are similar to XPath in so far that object types are listed that act as 'milestones' to be encountered (or to be avoided) on paths along an object traversal. These milestones can be distant from each other in a sense similar to XPath's descendant axis.

The SYB or SP style of programming is less declarative, more operational in that it is described how to descend into structures, how to search for relevant data. SYB provides flexibility, but the AP setup is specifically meant to enable powerful, meta-data-driven optimizations for shortcutting traversals over object graphs.

Sanity checks of traversals We do not claim any originality for the overall idea of checking on trivial behaviors of traversals. Instead, we have shown that such checks can be integrated into Haskell type checking. Our ad-hoc approach to sanity checking calls for a more compositional derivation of *checked* traversals from *checked* combinators.

Again, adaptive programming comprises similar static checks. That is, an adaptive programming system may perform so-called compatibility checks of class graph vs. traversal specification such that adaptive programs are rejected when the milestones cannot possibly be encountered in the specified order. The XQuery 1.0 and XPath 2.0 formal semantics [27] also describes related forms of type errors. We also refer to [1] for a type system for path correctness in XML query languages.

9. Conclusion

Typical XML programmers do not care too much about generic functional programming, but they have access to a rich variety of query axes and to simple-to-use recursion patterns. This observation suggests that a comparison of generic functional programming and 'structure-shy' XML programming may be informing and enriching for both sides.

As a first attempt at such a comparison, we carried out some experiments and analyses. First, SYB style and XSLT style turn out to be notably related. Second, SYB can approximate key fragments of XPath. Third, SYB provides more control on traversal, when compared to XPath and XSLT. Fourth, SYB-like term traversal can be vitally enhanced to provide parental and sibling context. Fifth, some known forms of trivial (unappreciated) behavior of XML queries and transformations also apply to the SYB style, and Haskell's type system can accommodate corresponding checks.

The most interesting directions for future work are these: (i) a more complete reconstruction of XPath, XSLT and XQuery (within the framework of (generic) functional programming); (ii) the systematic provision of type-driven traversal specialization for optimized (say, shortcutting) traversals; (iii) the more profound analysis of traversal properties to be checked statically.

References

[1] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types for path correctness of XML queries. In *ICFP'04, Proceedings*, pages 126–137. ACM Press, 2004.

[2] J. English. HXML, 2002. Web site: `http://www.flightlab.com/~joe/hxml/`.

[3] T. Hallgren. Fun with functional dependencies. In *Joint Winter Meeting of the Dep. of Science and Computer Eng., Chalmers University of Technology and Goteborg University, Varberg, Sweden*, Jan. 2001. `http://www.cs.chalmers.se/~hallgren/Papers/wm01.html`.

[4] R. Hinze. A new approach to generic functional programming. In *POPL'00, Proceedings*, pages 119–132. ACM Press, 2000.

[5] R. Hinze and J. Jeuring. Weaving a web. *Journal of Functional Programming*, 11(6):681–689, 2001.

[6] R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. *Science of Computer Programming*, 51(1–2):117–151, 2004.

[7] G. P. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[8] P. Jansson and J. Jeuring. PolyP – a polytypic programming language extension. In *POPL'97, Proceedings*, pages 470–482. ACM Press, 1997.

[9] M. Kay. What kind of language is XSLT? *IBM developerWorks*, 2001–2005. `http://www-128.ibm.com/developerworks/xml/library/x-xslt/?dwzone=x`.

[10] M. Kay. Comparing XSLT and XQuery. In *Proceedings of XTech 2005: XML, the Web and beyond*, 2005. `http://www.idealliance.org/proceedings/xtech05/papers/02-03-01/`.

[11] O. Kiselyov and S. Krishnamurthi. SXSLT: Manipulation Language for XML. In V. Dahl and P. Wadler, editors, *PADL'03, Proceedings*, volume 2562 of *LNCS*, pages 256–272. Springer-Verlag, 2003.

[12] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell Workshop 2004, Proceedings*. ACM Press, Sept. 2004.

[13] O. Kiselyov and K. Lisovsky. XML, XPath, XSLT implementations as SXML, SXPath, and SXSLT, Sept. 2002. Presented at the International Lisp Conference (2002); `http://okmij.org/ftp/`.

[14] R. Lämmel and S. Peyton Jones. Scrap your boilerplate, website, `http://www.cs.vu.nl/boilerplate/`, 2003–2006.

[15] R. Lämmel, E. Visser, and J. Visser. Strategic programming meets adaptive programming. In *AOSD'03, Proceedings*, pages 168–177. ACM Press, 2003.

[16] K. J. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and Efficient Implementation. *ACM TOPLAS*, 26(2):370–412, 2004.

[17] C. McBride. Faking It (Simulating Dependent Types in Haskell). *Journal of Functional Programming*, 12(4–5):375–392, July 2002.

[18] Microsoft Corp. XLinq overview, 2005. `http://msdn.microsoft.com/netframework/future/linq/`, Note that XLinq is now called 'LINQ to XML'.

[19] M. Neubauer, P. Thiemann, M. Gasbichler, and M. Sperber. A Functional Notation for Functional Dependencies. In *Haskell Workshop 2001, Proceedings*, ENTCS, pages 101–120. Elsevier, 2001.

[20] M. Neubauer, P. Thiemann, M. Gasbichler, and M. Sperber. Functional logic overloading. In *POPL'02, Proceedings*, pages 233–244. ACM Press, 2002.

[21] K. Normark. XML Transformations in Scheme with LAML – a Minimalistic Approach, Oct. 2003. Presented at the International Lisp Conference (2003); `http://www.cs.aau.dk/~normark/laml/`.

[22] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Haskell Workshop 2002, Proceedings*, pages 1–16. ACM Press, 2002.

[23] E. Visser, Z. el Abidine Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *ICFP'98, Proceedings*, pages 13–26. ACM Press, 1998.

[24] W3C. Document Object Model (DOM), 1997–2003. `http://www.w3.org/DOM/`.

[25] W3C. XML Path Language (XPath), Version 1.0, W3C Recommendation, 16 Nov. 1999. `http://www.w3.org/TR/xpath`.

[26] W3C. XSL Transformations (XSLT), Version 1.0, W3C Recommendation, 16 Nov. 1999. `http://www.w3.org/TR/xslt`.

[27] W3C. XQuery 1.0: An XML Query Language, W3C Candidate Recommendation, 3 Nov. 2005. `http://www.w3.org/TR/xquery/`.

[28] W3C. XML Path Language (XPath), Version 2.0, W3C Candidate Recommendation, 8 June 2006. `http://www.w3.org/TR/xpath20/`.

[29] W3C. XQuery 1.0 and XPath 2.0 Formal Semantics, W3C Candidate Recommendation, 8 June 2006. `http://www.w3.org/TR/xquery-semantics/`.

[30] W3C. XSL Transformations (XSLT) Version 2.0, W3C Candidate Recommendation, 8 June 2006. `http://www.w3.org/TR/xslt20/`.

[31] M. Wallace and C. Runciman. Haskell and XML: generic combinators or type-based translation? In *ICFP'99, Proceedings*, pages 148–159. ACM Press, 1999.