XPTO
An Xpath Preprocessor with Type-aware Optimization

Flávio Ferreira[*] Hugo Pacheco[†]

February 27, 2007

Engenharia de Sistemas e Informática, Universidade do Minho, Portugal

Abstract

Various languages allow specific query languages for selection and transformation of portions of documents. Such queries are defined generically for different data types, and only specify specific behaviours for a few relevant subtypes. This is a well-known feature of XML query languages, that allow selection of element nodes without exhaustively specifying intermediate nodes.

We have implemented a system for performing optimizations on XPath expressions through schema-specialization of their structure-shy properties. The core of the system consists of a combinator library, based on algebraic laws for transformation of structure-shy programs, conversion into structure-sensitive programs, and vice-versa. We show how the core library can be extended with laws for specific XPath features and adapted to construct an effective rewrite system for specialization and optimization of XPath structure-shy programs. The front-end for this system carries the conversion of XML Schema and XPath files into internal representations and the generation of Haskell programs containing optimized queries as Haskell functions. The front-end itself was implemented over the functional language Haskell.

Keywords Haskell, XPath, schema-specialization, optimization

1 Introduction

Query languages are designed to query collections of data over documents in a host language. This concept encompasses all the techniques to retrieve information from large sets of structured data. The probably most popular query language is the SQL language, modelled for handling data stored in relational databases. Developed for the sharing of data across different information systems, the XML markup language structures the data under a tree-based representation and stores it in regular text files.

[*]flavioxavier@gmail.com
[†]hpacheco@gmail.com

XPath is a simple SQL homologous for the XML technology, but is also an essential ingredient of XQuery and XSLT. It follows a structure-shy programming technique to navigate through the hierarchy of XML nodes. Structure-shy programs can be significantly more concise, by focusing on the essence of the algorithm rather than oozing with boilerplate code, what makes them more concise and understandable [24]. However, such advantages potentially reduce the efficiency of these queries, due to the need to look up for possible elements in the whole document and the resource to dynamic checks to determine wether to apply specific or generic behavior for each data node.

As a simple XML query language, various efforts have been made in the last few years to improve XPath efficiency [26] [18]: one of them is schema-based optimization [22]. In field's literature, schema-awareness is synonym of using schema-knowledge to perform some optimizations on XPath expressions by trying to eliminate impossible path expressions or to remove redundant conditions. We go further by performing type-specialization and optimization over the query, this means, the query semantics are simplified taking in account the document's structure, as described in the schema's type definition, and refining the specialized query into a Haskell functional program. A type-safe, type-directed rewrite system can be created for transformation of XPath structure-shy queries into structure-sensitive point-free functional programs [7]. This system relies upon a set of algebraic laws, formulated for transformation of point-free programs [5]. Type-specialized queries are point-free program specific for the schema in use, and may be run against any XML document that conforms to such schema.

In this report, we will focus on developing a front-end for optimization of Xpath queries, based on the presented rewrite system. Much of the theoretical work have been explored in the previous paper [7]. Both the front-end and the rewrite system are implemented in Haskell. Schema type definitions and optimized point-free queries can be outputted as Haskell datatypes and functions that, composed with a proper XML parser, generate a complete Haskell program.

A variety of contexts can be found for generated programs, whenever the same query needs to be run against documents conforming to the same schema multiple times. For example, web services commonly involve extraction of information from XML databases. Such extractions can be expressed according to previously well-defined selection functions in the XPath language and are likely to be performed several times (imagine a regular PHP website based on a XML database). Software maintenance is also strictly related to generation of tests and summary reports on data stored in XML databases. Tests and reports tend to be executed on a regular basis. Any other contexts in which query-specialization may play a role involve application integration relying on data retrieval from XML databases.

In Section 2, we present some concrete examples to motivate our approach. In Section 3, we briefly recapitulate previous work on two-level type-safe representation of data and selectors. In Section 4, we present the formalization of XPath axis and some of the algebraic laws for specialization into point-free

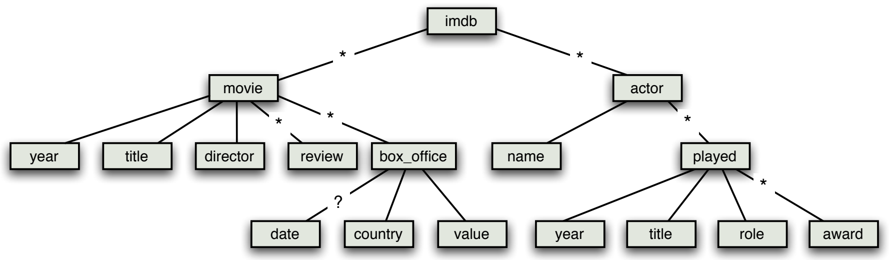

Figure 1: A movie database schema, inspired by IMDb (http://www.imdb.com/).

The following grammar describes a very resumed XPath syntax:

$$\begin{array}{ll}
\textit{xpath} & := \textit{expr}\ (\texttt{','}\ \textit{expr})\,? \\
\textit{expr} & := \textit{unionexpr} \mid \textit{numliteral} \\
\textit{unionexpr} & := \textit{expr}\ \texttt{'union'}\ \textit{expr} \\
\textit{location} & := \texttt{'/'}\ ?\ (\textit{step}\ (\texttt{'/'}\ \textit{step}) *) \\
\textit{step} & := \textit{axis}\ \texttt{'::'}\ \textit{test}\ \textit{pred} * \\
\textit{axis} & := \texttt{'child'} \mid \texttt{'descendant'} \mid \texttt{'self'} \mid \texttt{'descendant-or-self'} \\
\textit{test} & := \textit{name} \mid \texttt{'*'} \mid \texttt{'text()'} \mid \texttt{'node()'} \\
\textit{pred} & := \texttt{'['}\ \textit{xpath}\ \texttt{']'} \\
\textit{name} & := \textit{any document tag}
\end{array}$$

The full syntax is available in the XPath language reference [30]. Abbreviated syntax is available and heavily used where for instance `//` expands to `/descendant-or-self::node()/` and an element name without preceding axis modifier expands to `/child::`*name*.

Figure 2: Summary of XPath.

structure-sensitive programs. In Section 5 and Section 6, we explain in detail our implementation and show how the front-end can be used for tackling different scenarios. Later, in Section 7, we perform some tests on our tool and comparisons with other XPath processors and conclude about the efficiency and applicability of the technologies and approach used. We end with a discussion of related work (Section 8) and concluding remarks (Section 9).

2 Motivating Example

In this section, we will study some query examples for different degrees of structure-shyness. They will be specialized against the XML Schema in Figure 1 representing documents that hold information about movies and actors.

Retrieve all movie actors from the document

```
//movie/actor
```

This query asks to retrieve *actor* every elements that are direct children of *movie* elements appearing at any depth in the document's tree. However, we won't study this query in deeper detail as it is resumed to void, because there is no *actor* element under *movie*, once the *actor* is only defined at the same depth (Figure 1).

Retrieve all titles from the document

```
//title
```

This query selects all *title* elements at arbitrary depth, and works in a very similar way to the previous one. The $desc - or - self$ XPath axis is structure-shy, in the sense that it does not specifies the full tree path to reach *title* elements from the document's *imdb* root element. From the user's perspective, structure-shyness helps simplify queries, specially when the user has notion of the schema structure. The queries tend to become more understandable, concise, and adaptative to other schemas. However, we would like to optimize the query by improving it's structure-sensitivity to the schema. Knowing that *title* elements can occur under *movie* and *played* elements, we can derive:

```
imdb/(movie/title union actor/played/title)
```

Retrieve the first movie title from the document

```
(//movie/title)[1]
```

This query extends is a special case of the previous one. In order to choose only *title* elements child of movies, we have to specify that requirement. There is also an index for selecting the first element of all *title* elements inside *movie* elements.

In the optimized form, since we know that all *movie* elements must have a *title* child, the index can be factored out into the *movie* selection:

```
(imdb/movie)[1]/title
```

Note that this example query is completely different from having:

```
//movie/title[1]
```

In this second form, the query indexes the first element of all *title* elements (it doesn't enforce titles inside movies for the index). Since only one title appears at once, the index can be removed and the query can be specialized further to:

```
imdb/movie/title
```

Retrieve the third element from merging movie titles and reviews

```
(//movie/(title union review))[3]
```

This query can be constructed from the previous example, by changing the selection to both *title* and *review* elements, inside *movie* elements. The index was changed from the first element to the third. Merging the results means applying the set union over the result sets ($\mathit{title} \bigcup \mathit{review}$). XPath's result sets have order, what means that all title tags will appear after review tags. By indexing the final result, we are asking for a title if there are more than 2 title *elements* defined, or for a *review* otherwise.

Retrieve all directors for movies with year and box office date

```
//movie[year,box_office/date]/director
```

This last example uses a filtering predicate. The notation is similar to indexes, but since it doesn't receive a numeric expression, it is evaluated as the non-empty condition: if the query returns at least one element, then it succeeds, otherwise it fails.

The exact meaning of the query is to select every *director* elements child of *movie* elements, for the movies that have non-empty *year* and *box_office*'s *date* childs. As we can check in the schema for Figure 1, there must always be an *year* element for a *title* and a list of *box_office*. As monoids, lists allow empty lists, so we can't simplify it in the query. For each *box_office* element, the *date* element is optional, reason for which it can't be simplified as well.

The optimized query for this example is as follows:

```
imdb/movie[box_office/date]/director
```

In the following sections we will demonstrate how these transformations can be achieved through algebraic rewriting of XPath axis representations. This examples will be revisited in Section 5.

3 Type-safe Representation of Types and Queries

The various algebraic laws for query transformation can be harnesses into a type-safe, type-directed rewriting system for generalization, specialization and optimization of structure-shy programs. In this section, we present a mean of guaranteeing type-safeness in the representation of types, values and functions over them.

To ensure type-safety in our rewire system, a universal type representation of types does not suffice. Some rewrite laws make explicit reference to types, and therefore enforce their own type definition. To achieve this, we will need type-representations at the value level, which can be provided by using *generalized algebraic data types* (GADTs), a powerfull generalization of Haskell data types [28]. For all parameterized data *Type a*, their inhabitants must be representations of type *a* [19].

```
data Type a where
  Int    :: Type Int
  Bool   :: Type Bool
  String :: Type String
  List   :: Type a → Type [a]
  Prod   :: Type a → Type b → Type (a, b)
  Either :: Type a → Type b → Type (Either a b)
  Func   :: Type a → Type b → Type (a → b)
  Data   :: String → EP a b → Type b → Type a
  ⋆ :: Type a → Type ⋆
  ...
data ⋆ where ⋆ :: Type a → a → ⋆
```

Notice that, in this declaration, the type *a* that parameterizes *Type a* is restricted differently in each constructor. This makes the difference between a GADT and a common Haskell 98 parameterized datatype, where the parameters in the result type must always be unrestricted in all constructors. In the definition of a *Type a*, the GADT allows data constructors to return types of values other than the original type of the value they were given, and the parameter *a* of each constructor is restricted exactly to the type that the constructor represents.

Given a ground type *a* it is possible to use the Haskell type system to infer its representation. We can define a class with all representable types.

```
class Typeable a where typeof :: Type a
```

Most instances of this class are trivially defined. For example, for integers and functions we have

```
instance Typeable Int where typeof = Int
instance (Typeable a, Typeable b) ⇒ Typeable (a → b)
  where typeof = Func typeof typeof
```

New data types can be easily defined using the *Data* constructor. For example, the user-defined Haskell datatypes that represent our schema of Figure 1 can be represented in Haskell by the user-defined datatypes shown in Figure 3.

In the *Data* constructor, you can notice that it requires a value of type *Ep a b*, where *Type b* is the encapsulated type and *Type a* the resulting type for the newly defined datatype. *Data* works as a wrapper for a type (similar to a XML node tag), where *Ep* is an embedding-projection pair that converts values from the user-defined type into values of the isomorphic type. The type *a* is expected to be the sum-of-products representation of the user-defined type *a*.

Here, *Typeable* instances are assumed for *Movie* and *Actor*. The instance for *Imdb* is defined on top of instances for it's subtypes.

```
instance Typeable Imdb where
  typeof = Data "imdb" (EP unImdb Imdb) typeof
```

newtype $\mathit{Imdb} = \mathit{Imdb}\{\mathit{unImdb} :: ([\mathit{Movie}], [\mathit{Actor}])\}$
newtype $\mathit{Movie} = \mathit{Movie}\{\mathit{unMovie} :: (\mathit{Title}, (\mathit{Year}, ([\mathit{Review}],$
  $(\mathit{Director}, [\mathit{BoxOffice}]))))\}$
newtype $\mathit{Actor} = \mathit{Actor}\{\mathit{unActor} :: (\mathit{Name}, [\mathit{Played}])\}$
...

Here, we represent XML element tags from the schema's type definition from Figure 1 with Haskell data types. Each **newtype** defines a XML node, with it's own markup tag. For each node, a reverse method is provided for untagging values of it's type.

Figure 3: Haskell datatypes for the schema of Figure 1.

In XPath, combinators enjoy a very relaxed typing. Selection functions are implemented as aggregation functions for sets of types, but there is no distinction in grouping properties for values of different types. For this reason, sets of values don't have a type distinction, what can not be achieved in our type-safe representation. In order to overcome this issue, we defined a $\star$ constructor which hides a type inside it's definition. This way, it allows us to define *Type a* instances without being constrained to a specific type, but that type is not lost, since it is encapsulated inside the $\star$.

Analogously to types, we need to represent functions in the same type-safe manner. For this purpose, we resort again to a GADT, allowing function's type-checking for free: impossible or incorrect compositions of functions are checked against haskell's native type system and rejected. Constructors can be defined for different approaches like point-free, strategic or Xpath combinators. Such constructors are explained in [7].

Remembering some of the most important point-free combinators:

```
data F f where
  Id     :: F (a→a)
  Comp   :: Type b → F (b→c) → F (a→b) → F (a→c)
  Fst    :: F ((a, b)→a)
  Snd    :: F ((a, b)→b)
  (△)    :: F (a→b) → F (a→c) → F (a→(b, c))
  Plus   :: Monoid a → F ((a, a)→a)
  unData :: F (a→b)
  MkAny  :: F (a→⋆)
  Fun    :: String → (a→b) → F (a→b)
  Wrap   :: F (a→[a])
  ...
```

In this report we will focus on the XPath combinators required to implement XPath's most relevant features:

```
type Q r = ∀a . Type a → a → r
```

data $\mathsf{F}\ f$ **where**
  ...
  $\mathit{Index} :: \mathit{Int} \to \mathsf{F}\ ([a] \to [a])$
  ...
  $\mathit{Self} :: \mathsf{F}\ (\mathsf{Q}\ [\star])$
  $\mathit{Child} :: \mathsf{F}\ (\mathsf{Q}\ [\star])$
  $\mathit{Descendant} :: \mathsf{F}\ (\mathsf{Q}\ [\star])$
  $(//) :: \mathsf{F}\ (\mathsf{Q}\ [\star])$
  $(@) :: \mathsf{F}\ (\mathsf{Q}\ [\star])$
  $\mathit{Name} :: \mathit{String} \to \mathsf{F}\ (\mathsf{Q}\ [\star])$
  $(/) :: \mathsf{F}\ (\mathsf{Q}\ [\star]) \to \mathsf{F}\ (\mathsf{Q}\ r) \to \mathsf{F}\ (\mathsf{Q}\ r)$
  $(?) :: \mathsf{F}\ (\mathsf{Q}\ [\star]) \to \mathsf{F}\ (\mathsf{Q}\ \mathit{Bool}) \to \mathsf{F}\ (\mathsf{Q}\ [\star])$
  $(\blacktriangleright) :: \mathit{Type}\ a \to \mathsf{F}\ (\mathsf{Q}\ a) \to \mathsf{F}\ (a \to b) \to \mathsf{F}\ (\mathsf{Q}\ b)$
  $(\blacktriangle) :: \mathsf{F}\ (\mathsf{Q}\ a) \to \mathsf{F}\ (\mathsf{Q}\ b) \to \mathsf{F}\ (\mathsf{Q}\ (a, b))$
  $\mathit{NonEmpty} :: \mathsf{F}\ (\mathsf{Q}\ \mathit{Bool})$

There are two classes of generic combinators: type-unifying (queries), are defined as overloaded functions that return a result for a specific type; and type-preserving (traversals), that preserve the type of the result. XPath combinators are implemented as strategic type-unifying combinators. They were initially defined in terms of strategic combinators by Lämmel [23].

The *Index* combinator is used to filter certain elements over a result set, and is encoded as point-free combinator, since it is more generic than XPath context and may be applied in a normal point-free composition. In XPath, it receives a boolean predicate as an argument, but has been simplified in our approach to receive a single integer for the filtered element. This makes it easier to write simplification rules over this combinators, but is likely to change in a future version.

The $\blacktriangleright$ combinator defines composition of Xpath combinators with point-free combinators. This is a very important feature for guaranteeing the system's extensibility, since it allows extending the supported XPath features by composition with new point-free combinators. A practical example for this constructor is the query:

```
imdb[2]
```

Since the *Index* constructor is defined as a point-free function, we might compose it with the XPath axes, giving:

  $\mathit{Child}/\mathit{imdb}\ \blacktriangleright\ (\mathit{Index}\ 2)$

The $\blacktriangle$ combinator lifts the semantics of a point-free $\triangle$ into XPath combinators. It allows a simple way of defining tuples of XPath combinators and is very usefull essentially for binary operators. Imagine an Xpath constructor that allows the definition of constant values:

data F **where**
  ...
  $\mathit{Constant} :: a \to \mathsf{F}\ (\mathsf{Q}\ a)$
  ...

We could now create the constant XPath expression:

```
1+2
```

and convert it to our combinators:

$(\mathit{Constant}\ 1 \blacktriangle \mathit{Constant}\ 2) \blacktriangleright (\mathit{Plus}\ \mathit{mint}) \blacktriangleright (\mathit{Comp}\ \mathit{Wrap}\ \mathit{MkAny})$

Here, the sum of two numbers is performed by the monoid for integers. The result integer needs to be wrapped into a $[\star]$ for the type of the XPath result set to be valid.

If we would want to compose the last two examples

```
imdb[1+1]
```

we could preprocess the sum into a regular integer, since it returns a fixed value, or we could lift the *Index* combinator into a XPath combinator itself

data F **where**
  ...
  $\mathit{Index} :: \mathsf{F}\ (\mathsf{Q}\ \mathit{Int}) \to \mathsf{F}\ (\mathsf{Q}\ [a]) \to \mathsf{F}\ (\mathsf{Q}\ [a])$
  ...

$(\mathit{Constant}\ 1 \blacktriangle \mathit{Constant}\ 1) \blacktriangleright (\mathit{Plus}\ \mathit{mint})\ \mathit{Index}\ (\mathit{Child}/\mathit{imdb})$

At last, the @ combinator maps the XPath *attribute* axis, which returns all attributes of the actual element. It works on a similar way to the Xpath *child* axis (*Child*). An example query for this feature would be

```
imdb/attribute::version
```

encoded as

Child/*imdb*/*Child*/@/*version*

4 Point-free query calculus and rewriting

The developed rewrite system is defined as a composition of strategies, that are themselves smaller rewrite systems, in a similar approach to the F combinators presented in Section 3. Strategies for this rewrite system are type-preserving, and can be encoded in Haskell as monadic functions of the following type:

type $\mathit{Rule} = \forall f\ .\ \mathit{Type}\ f \to \mathsf{F}\ f \to \mathit{RewriteM}\ (\mathsf{F}\ f)$

Note that a *Rule* type receives an explicit argument type f, allowing rules to make type t-based rewriting decisions.

The *RewriteM* monad extends rules with the capability to store a proof trace during rewriting, including intermediate results and names of applied rules, and induces partiality on the transformation, since it has an instance on *MonadPlus*.

Now that the rewrite system's reasoning is explained, we can define any type-preserving strategies for transformation of programs. For this report, we

$index\ m \circ index\ n = index\ m$	
if $n \equiv 1$	*index*-COMP
$index\ m \circ index\ n = zero$, **otherwise**	
$index\ n \circ map\ f = map\ f \circ index\ n$	*index*-MAP
$index\ n \circ zero = zero$	*index*-ZERO
$index\ 1 \circ wrap = wrap$	
$index\ n \circ wrap = zero$, **if** $n > 0$	*index*-DEF
$index\ n \circ f = zero$, **if** $n < 1$	
$cond\ p\ f\ g = f$, **if** $f \equiv g$	*cond*-DEF
$cond\ b\ f\ (cond\ p\ g\ h) = cond\ p\ (cond\ b\ f\ g)\ (cond\ b\ f\ h)$	*cond*-FUSION1
$cond\ p\ f\ g \circ h = cond\ (p \circ h)\ (f \circ h)\ (g \circ h)$	*cond*-FUSION2
$(f \nabla g) \circ i1 = f$	
$(f \nabla g) \circ i2 = g$	*sum*-CANCEL
$h \circ (f \nabla g) \circ i1 = h \circ f$	
$h \circ (f \nabla g) \circ i2 = h \circ g$	
$i1 \nabla i2 = id$	*sum*-REFLEX
$(i1 + i2) = id$	
$f \circ (g \nabla h) = (f \circ g) \nabla (f \circ h)$	*sum*-FUSION
$(k1 \circ i1) \nabla (k2 \circ i2) = k1$, **if** $k1 \equiv k2$	*sum*-ETA
$f + g = (i1 \circ f) \nabla (i2 \circ g)$	*sum*-DEF

Figure 4: Our added laws for point-free program calculation.

will focus on optimizing XPath structure-shy queries into point-free structure-sensitive functions.

Our strategy is defined as specialization of the XPath strategic combinators into point-free functional programs.

In order to guarantee that the resulting function is on the canonical form, this means, does not contain any redundancy, a powerfull set of laws has to be applied. The first step is to convert all the XPath combinators into strategic combinators. Then, all the possible specializations are applied over the strategic program. At last, the strategic program is converted into a point-free expression, that can be simplified and refined into the result optimized function.

All this rewriting work is performed by the the *optimizexp* function, where *reduce* evaluates the *RewriteM* monad and *xpath* applies exhaustively all the presented rules:

$$optimizexp :: Typeable\ b \Rightarrow Type\ a \rightarrow \mathsf{F}\ (\mathsf{Q}\ b) \rightarrow \mathsf{F}\ (a \rightarrow b)$$
$$optimizexp\ a\ q = reduce\ xpath\ (Func\ a\ typeof)\ (ApplyQ\ a\ q)$$

Despite not being the purpose of this work, optimized point-free queries can still be converted into a XPath structure-sensitive equivalent representation (*pf2xp*).

$apQ_A\ (q \blacktriangleright f) = f \circ (apQ_A\ q)$	$\blacktriangleright$-APPLY
$a \blacktriangle b = mkQ_\star\ (apQ_\star\ a \triangle apQ_\star\ b)$	$\blacktriangle$-DEF
$@/name\ n = child/(name\ \texttt{'@'} : n)$	@-NAME

Figure 5: Our added laws for XPath strategic program calculation.

5 One-level transformation

Now that we have explained how our rewrite system works, we will demonstrate how to apply the specialization to each of the examples initially presented.

The first step is to parse the schema into the *Type a* representation:

$> DynT\ t \leftarrow xsd2type$ `"../examples/imdbNoTVDir.xsd"`

After that we have to, for each query, parse the XPath query into the $\mathsf{F}\ (\mathsf{Q}\ \star)$ representation.

$> q1 \leftarrow readXPath$ `"//movie/actor"` $\gg\!= return \circ xPath2PF$
$(descself\ /\ ((child\ /\ movie)\ /\ (child\ /\ actor)))$

$> q2 \leftarrow readXPath$ `"//title"` $\gg\!= return \circ xPath2PF$
$(descself\ /\ (child\ /\ title))$

$> q3 \leftarrow readXPath$ `"(//movie/title)[1]"` $\gg\!= return \circ xPath2PF$
$((descself\ /\ ((child\ /\ movie)\ /\ (child\ /\ title)))\ |> index\ 1)$

$> q4 \leftarrow readXPath$ `"(//movie/(title union review))[3]"` $\gg\!=$
$return \circ xPath2PF\ ((descself\ /\ ((child\ /\ movie)\ /$
$((union\ (child\ /\ title))\ (child\ /\ review))))\ |> index\ 3)$

$> q5 \leftarrow readXPath$ `"//movie[year,box_office/date]/director"` $\gg\!=$
$return \circ xPath2PF\ (descself\ /\ (((child\ /\ movie)\ ?\ (((union$
$(child\ /\ year))((child\ /\ box\_office)\ /\ (child\ /\ date)))$
$/\ nonempty))\ /\ (child\ /\ director)))$

Finally, we call the *optimizexp′* strategy (similar to *optimizexp* but prints reductions), responsible for converting XPath structure-shy queries into point-free structure-sensitive functions ($\mathsf{F}\ (a \rightarrow [\star])$).

For the first example, as expected, the query was reduced to a void path, and always returns the zero monoid for lists.

$>$ **let** $pf1 = optimizexp'\ t\ q1$
1250 *reductions*
listnil

For the second example, the query has been divided into two expressions, one for each possible occurrence of the tag *title* in the document. Note that the outer *listcat* is concatenating the result of these two expressions.

$>$ **let** $pf2 = optimizexp'\ t\ q2$
1090 *reductions*
$(listcat \circ (((listmap\ (mkDyn \circ (fst \circ unEmovie))) \circ (fst \circ unEimdb)) \triangle (concat \circ ((listmap$

$((\mathit{listmap}\ (\mathit{mkDyn} \circ (\mathit{fst} \circ \mathit{unEplayed}))) \circ (\mathit{snd} \circ \mathit{unEactor}))) \circ (\mathit{snd} \circ \mathit{unEimdb})))))$

For the third example, the index gets applied to the movie result set. The first *title* is directly extracted from the first *movie* in the document.

$> \mathbf{let}\ \mathit{pf3} = \mathit{optimizexp}'\ t\ \mathit{q3}$
1250 *reductions*
$((\mathit{listmap}\ (\mathit{mkDyn} \circ (\mathit{fst} \circ \mathit{unEmovie}))) \circ (\mathit{index}\ 1 \circ (\mathit{fst} \circ \mathit{unEimdb})))$

For the fourth query, not much can be optimized, in terms of XPath structure. The point-free function has exactly the same semantics as the XPath expression.

$> \mathbf{let}\ \mathit{pf4} = \mathit{optimizexp}'\ t\ \mathit{q4}$
1331 *reductions*
$(\mathit{index}\ 3 \circ (\mathit{concat} \circ ((\mathit{listmap}\ (\mathit{listcat} \circ ((\mathit{wrap} \circ (\mathit{mkDyn} \circ (\mathit{fst} \circ \mathit{unEmovie}))) \triangle$
$((\mathit{listmap}\ \mathit{mkDyn}) \circ (\mathit{fst} \circ (\mathit{snd} \circ (\mathit{snd} \circ \mathit{unEmovie}))))))) \circ (\mathit{fst} \circ \mathit{unEimdb}))))$

For the latter example, the most relevant difference is in the encoding of the *NonEmpty* combinator. It is expressed as a conditional test $(\mathit{cond} :: (a \to \mathit{Bool}) \to (a \to b) \to (a \to b) \to a \to b)$. The *cond* operator is applied to a list of elements. The guard condition is implemented as a $\mathit{or} :: [\mathit{Bool}] \to \mathit{Bool}$, and for each element of the list is returned a *True* constant value. Once the list has more that one value, the Haskell lazy evaluator returns success, meaning that there is at least one child obeying to the given description. For each *movie*, a *cond* is applied to the list of *box office* childs. Repeatedly, for each existant *box office* is applied a *cond* to test wether the optional *date* exists or not.

Remember that the *year* element always exists under *movie* and was simplified

$> \mathbf{let}\ \mathit{pf5} = \mathit{optimizexp}'\ t\ \mathit{q5}$
1588 *reductions*
$(\mathit{concat} \circ ((\mathit{listmap}\ (((\mathit{cond}\ (\mathit{or} \circ (\mathit{listcat} \circ ((\mathit{wrap} \circ \mathit{true}) \triangle (\mathit{concat} \circ ((\mathit{listmap}\ (((\mathit{wrap} \circ \mathit{true})$
$\nabla \mathit{listnil}) \circ (\mathit{fst} \circ \mathit{unEbox\_office}))) \circ (\mathit{snd} \circ (\mathit{snd} \circ (\mathit{snd} \circ (\mathit{snd} \circ \mathit{unEmovie})))))))))))\ (\mathit{wrap} \circ$
$(\mathit{mkDyn} \circ (\mathit{fst} \circ (\mathit{snd} \circ (\mathit{snd} \circ (\mathit{snd} \circ \mathit{unEmovie}))))))\ \mathit{listnil})) \circ (\mathit{fst} \circ \mathit{unEimdb})))$

If we want to generate an Haskell module with the query and corresponding data types for the schema, we need to serialize the *Type a* structure, because there can be no two data types with the same name, and the schema might contain elements with the same tag but different types.

$> \mathbf{let}\ t' = \mathit{serializeTypeSmart}\ t$

For last, we output the generated code for the desired query (*pf1*). The boolean argument sets if we allow laziness in parsing or not.

$> \mathit{prettyPrint}\ (\mathit{type2HsModule}\ \mathit{True}\ t'\ \mathit{pf1})$

For a more detailed explanation of the rules applied during optimization, please refer to [7].

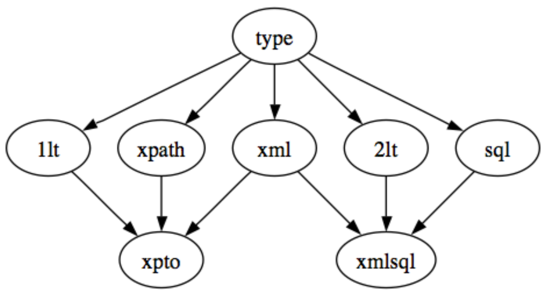

Figure 6: Structure of the repository (http://haskell.di.uminho.pt/repos/darcsweb.cgi?r=Two%20Level%20Transformation;a=summary).

6 Front-end

This project inherits some of the theory and implementation from the previous 2LT project[2] and extends it with a new front-end, one-level rewrite rules, and an XPath parser. For this reason, the original distribution has been refactored into transformation, parsing and front-end modules, with all of them sharing a root module containing the GADT representation of types and queries, along with their Spine generic mappings.

The relations between the modules define a partially ordered set, shown in Figure 6

Architecture The final solution is to be used in three phases: generation of the optimized query as an Haskell program; followed by compilation of the generated program; and execution with one or more XML argument files. There is also the option to run all the three phases at once, by evaluating the generated PF expression automatically.

The generation of the optimized queries represents the front-end's kernel, containing functions for parsing, convertion into our type-safe representation and application of transformation rules. The result of applying the optimization strategy is a point-free Haskell program, that can be outputted into a file for later compilation (Figure 8).

Since we process several XML languages, the front-end functions are combined with parsers and pretty-printers for XML, XSD and XPath abstract syntax trees. The conversion to our type-safe representation, XML and XSD parsers in use were inherited from the 2LT project [2]: for XML we use the HaXml parser and printer [1] and for XSD we use the XML Schema instances from the XsdMetz tool [29] with some modifications, which in turn uses HaXml (XML Schemas are themselves XML files). In order to support lazy parsing, we have upgraded HaXml to development version 1.17, enforcing the conversion of XML Schema instances into the parser combinators technique. The Xpath parser and pretty-printer were hand-crafted and implemented with the fast combina-

```
Usage: xpto [OPTION...]

  -h          --help            Show usage info
  -i XML      --input=XML       Input xml files
  -x XSD      --xsd=XSD         Input schema file
  -s Haskell  --source=Haskell  Input Haskell source file
  -o FILE     --output=FILE     Output file (default: stdout)
  -q XPATH    --xpath=XPATH     Input xpath query.
  -v          --validate        Schema validation on
```

Figure 7: Usage options for the xpto demo program

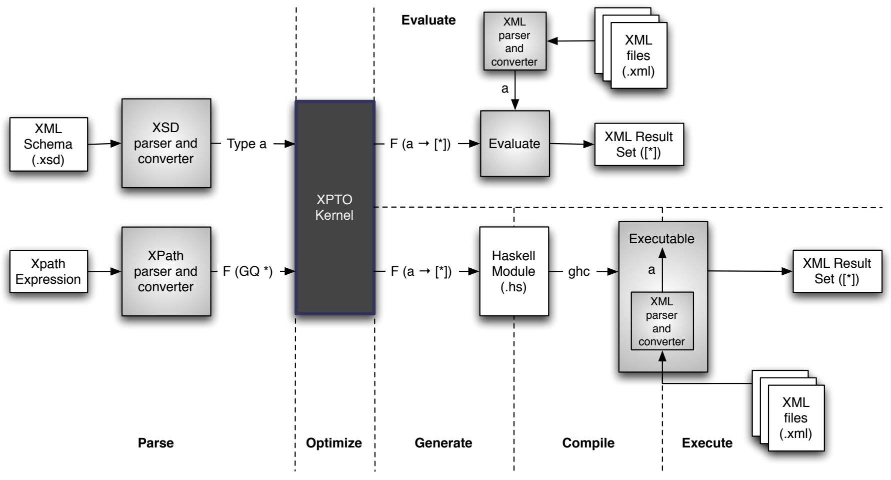

Figure 8: Architecture for our solution

tor parser library Parsec [25]. They are XPath 2.0 compliant and support the full specification [30]. There was also written a extended to abbreviate syntax converter.

Laziness XML parsing is the most expensive operation in XPath query processing, due to the great amounts of data frequently stored in such file databases. For this reason, we should avoid parsing unnecessary element nodes. Haskell lazy evaluation addresses this problem, by delaying term evaluation until it's result is known to be needed, leading to potential improvements on execution time. By matching the Xpath query against to the XSD schema definition, properties for order, repetition and existence of elements can be inferred. This knowledge, essential to our strategy, helps identifying and selecting only the desired nodes for the query and extends the lazy parser with the ability to stop at the last selected node's position in the input file. Note that, however, allowing laziness in parsing compromises validation of the full document, but also makes it less sensitive to possible errors at greater depth.

There are plenty of examples where laziness would improve dramatically the parsing times. The most basic case is when the query doesn't conform to the schema. In such a case the return set will be empty, without the need to even parse the XML input document.

Any other example is when the query only selects elements from the beginning of the XML document, which is the case of the motivation example

```
//movie/title[1]
```

In this case, this query returns the title of the first movie in the XML database which, according to the schema, must be the first child element of the root element imdb.

FrontEndXpto class The front-end should be responsible for converting parser structures into type-safe context-dependant data types. The following class captures the pattern behind this scenario and generalises it for any parsed query matching a two-level structure. The *Maybe* monad indicates the partiality of the conversions.

$\textbf{type}\ \mathit{Query}\ a = \mathsf{F}\ (a \to [\star])$

$\textbf{class}\ \mathit{FrontEndXpto}\ t\ v\ q \mid t \to v, v \to t, t \to q, q \to t\ \textbf{where}$
$\quad \mathit{parsetype} :: t \to \mathit{Maybe}\ \mathit{DynType}$
$\quad \mathit{printtype} :: \mathit{Type}\ a \to \mathit{Maybe}\ t$
$\quad \mathit{parsevalue} :: \mathit{Bool} \to \mathit{Type}\ a \to v \to \mathit{Maybe}\ a$
$\quad \mathit{printvalue} :: \mathit{Type}\ a \to a \to \mathit{Maybe}\ v$
$\quad \mathit{parsequery} :: \mathit{Type}\ a \to q \to \mathit{Maybe}\ (\mathit{Query}\ a)$
$\quad \mathit{printquery} :: \mathit{Query}\ a \to \mathit{Maybe}\ q$

$\textbf{instance}\ (\mathit{FrontEndXpto}\ \mathit{Schema}\ (\mathit{Document}\ \mathit{Posn})\ \mathit{XPath}\ \mathit{HsModule})\ \textbf{where}\ \ldots$

The initial class methods, over type and values, refer to the original FrontEnd class as defined in [2]. The boolean argument present in the *parsevalue* function consists on a switch for controlling document validation. In other words, parsing and further conversion laziness cannot be activated if document validation is a requirement.

In this tool, *parsequery* composes the conversion of the query with the optimization strategy and returns the optimized query in point-free fashion.

Despite having a single instance defined for the the context of XPath query optimization, different instances could be defined for other query languages. For instance, it would be perfectly reasonable to create an instance for the SQL language, where queries would correspond to SELECT statements over a database, and code would be PL/SQL code encapsulating a *SELECT* query and possibly appending some extra context behaviours to it.

We can reuse the interface of the *FrontEnd* class to program overloaded functions that represent the logic of our tool, and moreover, evaluate or generate source code for queries, on top of external abstract syntaxes. Although they could be generically implemented, both functions perform very particular operations, bound to the context of this work. The *Bool* argument indicates wether the operations assume laziness or require full document validation.

The *eval* function presented above wraps all the specialization and optimization strategy contained in *evalOptPF* into for a *FrontEnd* instance. The evaluation is implemented for the F function representation itself.

```
eval :: (FrontEndXpto t v q) ⇒ Bool → t → v → q → Maybe [⋆]
eval b t v q = do
  DynT pT ← parsetype t
  pV ← parsevalue b pT v
  pQ ← parsequery pT q
  return (evalOptPF pT pQ pV)
```

The *gen* function generates and outputs a valid Haskell program representing the *Type a* and *Query a* instances, created from their *FrontEnd* parameterized syntax trees.

Some of the generated code comes “for free“ from the *Spine* mapping for *Type a* constructors. Spines allow performing generic traversals over types, including show functions. We use this *Spine* instances to generate valid Haskell representations of types. Queries, optimized to point-free functions, also have an associate *Type* declaration and can be pretty-printed under the same technique.

In the generated module, types are defined using Haskell's **newtype**. All of this would work using a **data** declaration instead, but the **data** declaration incurs extra overhead in the representation of values for the declared type. The use of **newtype** avoids the extra level of indirection (caused by laziness) that the **data** declaration would introduce.

For each type is defined a *Typeable* instance and a mapping function over values for those type. *Typeable* instances enable the parser to derive *Type a* declarations natively for Haskell defined types.

All this code is wrapped into an instance of Haskell's language syntax module and can be pretty-printed into a output file.

```
gen :: (FrontEndXpto t v q) ⇒ Bool → t → q → Maybe HsModule
gen b t q = do
  DynT pT ← parsetype t
  pQ ← parsequery pT q
  return (type2HsModule b pT pQ)
```

7 Tests and Benchmarking

In this section we discuss the results of comparing the developed front-end against Saxon Schema-Aware, one of the most popular and fastest XPath processors in the market.

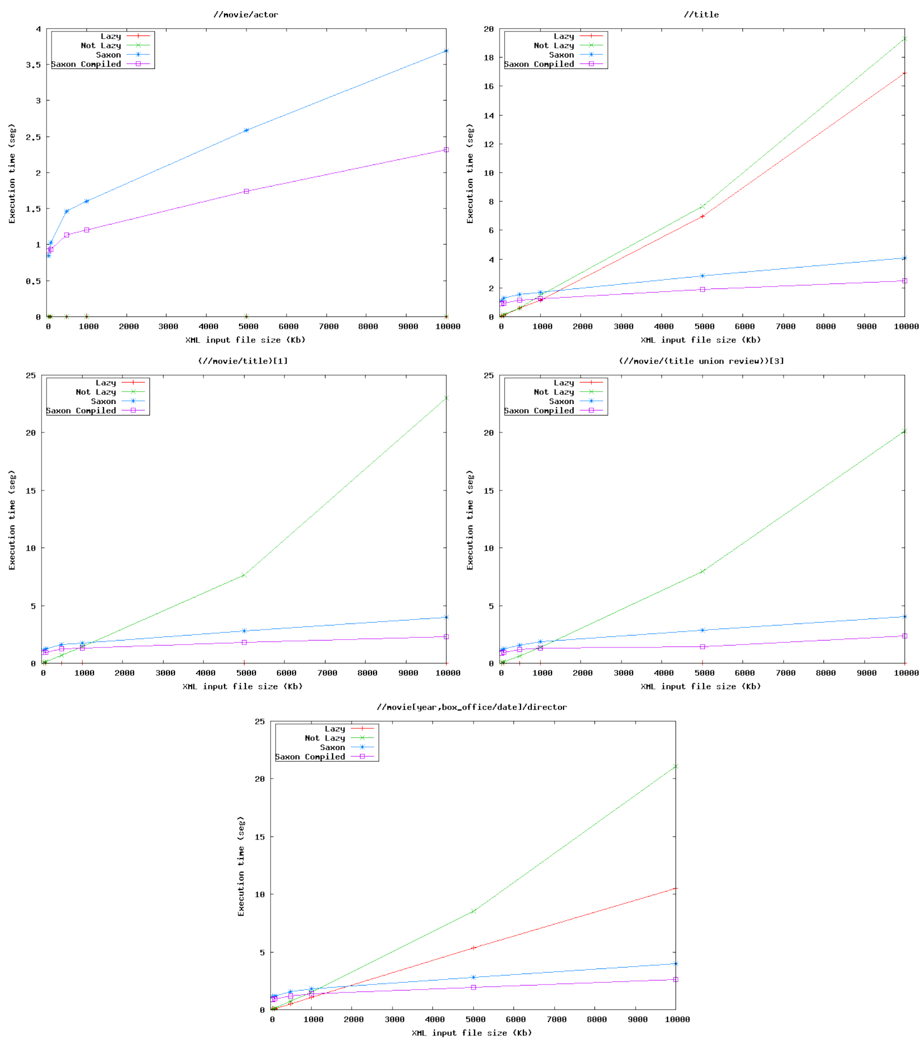

Figure 9: Comparison tests between our tool and Saxon SA

.

Benchmark and profiling tests were run for all the examples. For testing, GHC version 6.6 with optimization flag -O2 has been used. The Haskell XML parser used was HaXml development version 1.17. Here, we show the results of

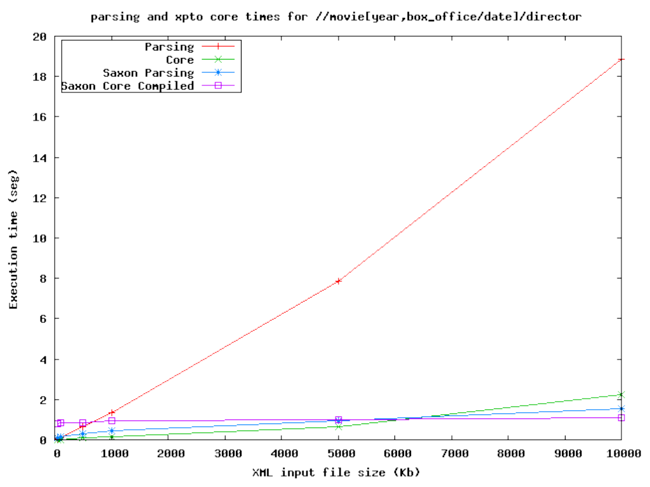

Figure 10: Profiling times for our tool and Saxon SA compiled queries

.

benchmarking all queries and the profiling for one of them[1].

By analysis of the results, we can see that Saxon has a reasonably high starting time of around 0.80s for all the queries. However, it proves to be very efficient in general. Parsing time has a very good linear relational with the XML database size, and execution times are almost constant for queries with different features.

The results for our tool are much more complex. For inconsistent queries that are optimized to a void path, such as the first example (Section 2), Haskell's lazy evaluator doesn't require the input XML document to be parsed and the total time is always 0s. On the other side, Saxon always parses the input XML file independently on the query, since it has optimistic algorithms for evaluating XPath queries, but doesn't refine the queries before evaluation.

For valid XPath queries that doesn't require evaluating the whole document, lazy parsing proves to make a significant difference, dependant on the relative position of the queried elements in the document. The proof for this property are examples 3 (Section 2) and 4 (Section 2), where indexing selects elements in starting positions of the input document and queries take both 0s for any database size.

For examples 2 (Section 2) and 5 (Section 2), lazy parsing still performs significant improvements, but Saxon is much faster for greater database sizes in both cases.

Concluding, despite the lack of optimizations and specializations, Saxon SA proves to be faster than our implementation and has better scalability. Our

[1]The complete profiling data, can be obtain at our darcs repository

compiled programs, although very slow in XML parsing, are instantaneous for smaller files or for queries that allow high laziness. Since parsing times are responsible for $> 90\%$ of the total time, we can still conclude that our tool is still useful, showing interesting results, and should be further develop.

Despite the great dependence on parsing times, our experimental system in the Haskell functional language can compete with Saxon SA, an already very optimized library for Java, a fast and very optimized language. The most relevant feature for determining this theory success is the precise cost of evaluating an XPath query in relation to the query's complexity. The rewrite system and the front-end can still be greatly improved, by refining transformation rules and internal representations.

We have studied the efficiency of our implementation, specially in relation to the XML database size. More theoretical tests should be done in the near future, inspired on Gottlob *et al* continuous study on the precise complexity of Xpath query processing [17] [16] [15].

8 Related work

Type-directed partial evaluation Partial evaluation is a technique for specializing programs with knownledge of some of it's input data. It can be seen as a special case of program transformation, but emphasizes full automation and generation of program generators as well as transformation of single programs. Further, it is adopted by compilers and interpreters and gives insight into the properties of programming languages themselves.

Danvy [8] presents a type-directed partial evaluator backed on typed lambda-calculus. More recently, Ens-Lyon [11] has published an implementation inspired on Danvy's concept. Type-directed partial evaluation uses no symbolic evaluation for specialization, and naturally processes static computational effects. Therefore, source programs must be closed and monomorphically typeable.

Our rewrite system is somehow similar to type-directed partial evaluation in the sense that we perform optimizations on queries, based on their composite type definition and preprocess their structure-shyness by partially evaluating generic traversals. By generating specialized and optimized programs, it resembles the effort of partial evaluation in program optimization and compilation.

XML algebras Through years, many algebras for XML queries have been formulated, either as independent processors, either for optimization of standard algebras (XPath and XQuery). The most relevant for our paper are XML query optimization algebras, from which we chose the PAT [3] and XAL [12] algebras. For both, algebraic equivalences can be conveniently expressed and grouped into a transformation system for query optimization. A large set of equivalences and corresponding rules are presented.

PAT-algebra expressions return node sets of a single static type. Optimization rules exploit schema information, increase the structure-shyness of queries and introduce structure-indices to short-cut navigation.

XAL algebra resides on the notion of collection. Operators are classified in three clusters, being the two most relevant ones extraction operators (select the desired information from XML documents) and construction operators (build new XML documents from extracted data). Optimizations are implemented for extraction operators and include monad laws and generalizations of relational laws to an object algebra context.

Our model of XPath, using strategy combinators and dynamic types, is more faithful, since it doesn't offer a limited set of axes but allows arbitrary functions and, more importantly, is completely extensible to non-XPath queries. It is applicable to any hierarchical data structure and eases conversions between structure-shy and structure-sensitive programs. One example of this is that we have the PAT-algebra combinators and rules encoded in our system.

Saxon compiled queries *Kay* is the creator of Saxon [20], a very popular XSLT and XQuery Processor that claims an important role on XML processing over Java and .NET. Since it's last version, Saxon Schema-Aware features direct compilation of XQuery queries (and ,consequently, XPath) into Java source code, reducing execution times.

The main difference to our strategy is in the notion of schema-aware optimizations. Saxon improves execution times mainly by removing the overhead of parsing the XPath query and performing some optimizations over it without schema-awareness [21], what tells that these optimizations are mostly related to java code and algorithmic optimizations. Schema-awareness implies validating the input and output documents against schema's type definition, what represents a cost in efficiency, compared to a non-schema-validating scenario.

In our approach, schema-awareness not only allows XML validation, but most of all consists on the specialization of queries according to the schema definition. The final result is a straightforward selection function with a built-in type representation, against which the input XML document is parsed.

Being the most similar to ours, with the same goals, this approach represents an important comparison reference, relevant in the testing of our solution and final conclusions about efficiency and usability.

Xpath core language Genevès *et al* [14] propose a method for normalizing XML queries into a minimal "core" language, as specified in the XPath/XQuery formal semantics [9]. This translation is achieved through a three-staged approach. The first step is to normalize the expression into a minimal but fully expressive "XPath core" expression, before replacing all the context position references for equivalents computed from the context node. At last, steps involving reverse axis are converted to steps using the corresponding forward axis [27]. Normalized XPath queries in the "core" language belong to a stateless forward-only subset, and therefore, are more straightforward and optimized queries.

Although not formally defined, this approach addresses the possibility to transform XPath queries into simpler and faster programs, preserving their semantics. Such normalizations may inspire new rules for our model, for example, avoid evaluating backward axis by converting them to the most similar forward axis representation.

Logic-based Xpath Genevés *et al* [13] describe a logic-based optimization system for XPath queries. Optimizations are performed by static analysis on the containment relations inherent to the XML tree model, at syntactic level. A containment relation $p1 \leqslant p2$ holds true when, for a node t, the set of nodes selected by $p1(t)$ is included in the set by $p2(t)$. Rules are provided for redundancy elimination, that handle union and eliminate qualifier conditions induced naturally in the path or by composition, and for void path elimination, by finding implicit contradictions in the path expressions.

Our rewriting system follows a completely different approach. Specialization is performed at the semantic level, taking in account the structural and semantic connections described in the schema. Although their system proposes to be universal, possibly embedded in any XPath engine, we tackle optimization through specialization into Haskell source code. The generated code can be seen as a processor for a specific query in a specific schema, what allows native schema-validation of input XML documents.

Such behaviour, schema-awareness and semantic integration, can be useful for exploiting schema and ontology hierarchies in XPath queries.

In our case, void path detection comes for free by matching the query against the schema definition.

Coupled transformations Previously, we have implemented a front-end for performing format evolution and data mappings [2], based on a type-safe, type-changing strategic rewrite system for two-level transformations [4]. In [6], this system has been extended with transformation of corresponding data processing programs. Format evolutions require associated migration functions for coupled values. Programs over the original type are then composed with these migration functions in order to create programs over the evolution. These resulting point-free programs have an explicit reference to the intermediate type, and can be optimized through generalization to structure-shy data processors by [7]. Coupled transformations now also encompasses migration and mapping of structure-shy data consumers and producers.

9 Concluding Remarks

9.1 Contributions

In this report, we discuss the practical application of our Haskell-based query optimization system. In particular, we make these contributions to the 2LT project:

1. We have created a complete XPath 2.0 parser, upgraded the old XML Schema parser to HaXml 1.17 and adapted the old XML to *Type* conversion modules for support of lazy parsing.
2. We have extended the program transformation kernel with rules for handling XPath new combinators.
3. We embed the general transformation kernel into a XPath query transformation framework, including a frontend for evaluating XPath queries based on schema-validation and for automated generating of structure-sensitive point-free programs.
4. We illustrate by example how the framework can be used to optimize different queries.
5. We have improved the maintainability of the 2LT suite, by refactoring it's structure, upgrading all the code to GHC version 6.6 and creating a darcs online repository that we maintain.

9.2 Future Work

Though already useful in practise, our approach suffers from various limitations that we intend to overcome.

Further combinators and languages Although studies prove that people tend not to use many Xpath 2.0 functionalities, and although our current solution covers most of most used, it is still limited. New combinators should be added, specially XPath native functions and operators in the form of point-free combinators.

Actually, our rewrite system is directed to optimization of queries as selection functions. Adding support for XML transformations and evolutions under XQuery or XSLT would prove the potentially of this approach in XML processing.

Improve internal representations Actually, XSD schemas are represented as instances of a generalized algebraic data type with basic constructors. However, many XML Schema constraints get lost in the conversion, such as type restrictions. The existant *Type* representation should be extended in order to support type constraints.

The actual representation of XML attributes is as a predefined prefix. Attributes don't have a specific namespace and share element's childhood with other non-attribute nodes. This representation is no more than an assumption, and should be formalized. One example for addressing this problem is with the *Child* axis. It matches all the child nodes of a certain element, independently of being attributes or not. This could lead to some loss in efficiency on the combinator and, worst of all, if an user queries for a node that matches the representation of an attribute with the constant prefix, he may fetch an element that doesn't exist but is in fact an attribute.

XML Parsing Parsing times represents the most significant share n the execution time of optimized queries. This is the most crucial aspect to be improved in the near future, if we want this tool to have impact in current XML processing techniques. Better parsing performance may be achieved by changing to the Haskell XML Toolbox parser (`http://www.fh-wedel.de/~si/HXmlToolbox/`), or by manually improving the actual HaXml version. Moreover, much of these limitations are bound to the language itself, reason for which we are seriously considering in mapping these type-safe structures, rewrite system and algebraic laws into an object oriented-language, with much more efficient and elaborated parsing libraries. This approach is not intended to replace the current Haskell implementation.

Coupled transformations integration This project was born as an extension to provide our coupled transformations rewrite system with the ability to refine not only data structures, but also migration functions for values bound to those structures. In the same line of the previous work [7], the developed program rewrite system could be adapted and linked with the two-level transformation framework, in order to allow optimization of the migration functions. Conversions between structure-shy and structure-sensitive are also applicable to migration functions.

The addition of a strategy for optimization of SQL queries would provide the two-level transformation framework with the ability to optimize transformed queries specifically for the destination model. There are many literary resources on approaches for optimization of relational queries.

Acknowledgments

Thanks to Alcino Cunha and Joost Visser for supervision, dedication and inspiring discussion about representation and transformation of XPath.

References

[1] HaXml 1.17. Haxml: Haskell and XML, 2006.

[2] Pablo Berdaguer, Alcino Cunha, Hugo Pacheco, and Joost Visser. Coupled schema transformation and data: Conversion for xml and sql. In *PADL 2007*, pages 290–304. Springer-Verlag, LNCS 4085, February 2007.

[3] Dunren Che, Karl Aberer, and Tamer &#x00d6;zsu. Query optimization in xml structured-document databases. *The VLDB Journal*, 15(3):263–289, 2006.

[4] A. Cunha, J.N. Oliveira, and J. Visser. Type-safe two-level data transformation. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Proc. Formal Methods, 14th Int. Symp. Formal Methods Europe*, volume 4085 of *LNCS*, pages 284–299. Springer, 2006.

[5] A. Cunha and J. Sousa Pinto. Point-free program transformation. *Fundam. Inform.*, 66(4):315–352, 2005.

[6] A. Cunha and J. Visser. Strongly typed rewriting for coupled software transformation. In M. Fernandez and R. Lämmel, editors, *Proc. 7th Int. Workshop on Rule-Based Programming (RULE 2006)*, ENTCS. Elsevier, 2006. To appear.

[7] A. Cunha and J. Visser. Transformation of structure-shy programs: Applied to xpath queries and strategic functions. In *ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation*, 2006.

[8] Olivier Danvy. Type-directed partial evaluation. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 242–257, New York, NY, USA, 1996. ACM Press.

[9] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. Xquery 1.0 and xpath 2.0 formal semantics, 2002.

[10] R. Ennals and S. Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs, 2003.

[11] Kristoffer Rose Ens-Lyon. Type-directed partial evaluation in haskell.

[12] Flavius Frasincar, Geert-Jan Houben, and Cristian Pau. Xal: an algebra for xml query optimization. In *ADC '02: Proceedings of the 13th Australasian database conference*, pages 49–56, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[13] Pierre Genevés and Jean-Yves Vion-Dury. Logic-based xpath optimization. In *DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering*, pages 211–219, New York, NY, USA, 2004. ACM Press.

[14] Pierre GenevÃĺs and Kristoffer Rose. Compiling xpath into a state-less forward-only subset. Technical report, IBM T. J. Watson Research Center, 2004.

[15] G. Gottlob, C. Koch, and R. Pichler. Xpath query evaluation: Improving time and space efficiency, 2003.

[16] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing xpath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[17] Georg Gottlob, Christoph Koch, Reinhard Pichler, and Luc Segoufin. The complexity of xpath query evaluation and xml typing. *J. ACM*, 52(2):284–335, 2005.

[18] Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Optimized translation of xpath into algebraic expressions parameterized by programs containing navigational primitives. In *WISE '02: Proceedings of the 3rd International Conference on Web Information Systems Engineering*, pages 215–224, Washington, DC, USA, 2002. IEEE Computer Society.

[19] R. Hinze, A. Löh, and B.C.d.S. Oliveira. "Scrap your boilerplate" reloaded. In M. Hagiya and P. Wadler, editors, *Proc. Functional and Logic Programming, 8th Int. Symp.*, volume 3945 of *LNCS*, pages 13–29. Springer, 2006.

[20] Michael Kay. Saxon: Anatomy of an xslt processor. In *IBM developerWorks*, 2001.

[21] Michael Kay. Xslt and xpath optimization. In *XML Europe*, 2004.

[22] A. Kwong and M. Gertz. Schema-based optimization of xpath expressions, 2002.

[23] R. Lämmel. Scrap your boilerplate with XPath-like combinators, 15 July 2006. Draft, 6 pages, Accepted as short paper at POPL 2007.

[24] R. Lämmel, E. Visser, and J. Visser. Strategic programming meets adaptive programming, 2003.

[25] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[26] Philippe Michiels. Xquery optimization. Technical report, University of Antwerp, Belgium, 2003.

[27] Dan Olteanu, Holger Meuss, Tim Furche, and Fran&#231;ois Bry. Xpath: Looking forward. In *EDBT '02: Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, pages 109–127, London, UK, 2002. Springer-Verlag.

[28] S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, Univ. of Pennsylvania, July 2004.

[29] Joost Visser. Structure metrics for xml schema. Technical report, Proceedings of XATA 2006, 2006.

[30] W3C. XML path language (XPath) 2.0, W3C candidate recommendation, 2006.

A Generated (lazy) code for the (//*movie*/*title*) [1] query

```
module Main where
import Type (Type (..), Typeable (..), EP (..), Dynamic (..))
import Text.XML.HaXml.Types (Document)
import XPathResult
import Pointless.Combinators
import Text.XML.HaXml.Posn
import PreludeXptoLazy
import Transform.XML.XMLtoTypeLazy (xml2values)
newtype Eimdb = Eimdb{ unEimdb :: ([Emovie], [Eactor])}
instance Typeable Eimdb where
  typeof = Data "imdb" (EP unEimdb Eimdb) typeof
instance Show Eimdb where
  show (Eimdb a) = "Eimdb " ++ show a
mapEimdb ::
  (([Emovie], [Eactor]) → ([Emovie], [Eactor])) → Eimdb → Eimdb
mapEimdb f = Eimdb ∘ f ∘ unEimdb
newtype Emovie = Emovie{ unEmovie ::
  (Etitle, (Eyear, ([Ereview], (Edirector, [Ebox_office]))))}
instance Typeable Emovie where
  typeof = Data "movie" (EP unEmovie Emovie) typeof
instance Show Emovie where
  show (Emovie a) = "Emovie " ++ show a
mapEmovie ::
  ((Etitle, (Eyear, ([Ereview], (Edirector, [Ebox_office])))) →
    (Etitle, (Eyear, ([Ereview], (Edirector, [Ebox_office])))))
     → Emovie → Emovie
mapEmovie f = Emovie ∘ f ∘ unEmovie
newtype Etitle = Etitle{ unEtitle :: String}
instance Typeable Etitle where
  typeof = Data "title" (EP unEtitle Etitle) typeof
instance Show Etitle where
  show (Etitle a) = "Etitle " ++ show a
mapEtitle :: (String → String) → Etitle → Etitle
mapEtitle f = Etitle ∘ f ∘ unEtitle
newtype Eyear = Eyear{ unEyear :: Int}
instance Typeable Eyear where
  typeof = Data "year" (EP unEyear Eyear) typeof
instance Show Eyear where
```

```
  show (Eyear a) = "Eyear " ++ show a
mapEyear :: (Int → Int) → Eyear → Eyear
mapEyear f = Eyear ∘ f ∘ unEyear
newtype Ereview = Ereview{ unEreview :: String}
instance Typeable Ereview where
  typeof = Data "review" (EP unEreview Ereview) typeof
instance Show Ereview where
  show (Ereview a) = "Ereview " ++ show a
mapEreview :: (String → String) → Ereview → Ereview
mapEreview f = Ereview ∘ f ∘ unEreview
newtype Edirector = Edirector{ unEdirector :: String}
instance Typeable Edirector where
  typeof = Data "director" (EP unEdirector Edirector) typeof
instance Show Edirector where
  show (Edirector a) = "Edirector " ++ show a
mapEdirector :: (String → String) → Edirector → Edirector
mapEdirector f = Edirector ∘ f ∘ unEdirector
newtype Ebox_office = Ebox_office{ unEbox_office ::
  (Either (Edate) (()), (Ecountry, Evalue))}
instance Typeable Ebox_office where
  typeof = Data "box_office" (EP unEbox_office Ebox_office) typeof
instance Show Ebox_office where
  show (Ebox_office a) = "Ebox_office " ++ show a
mapEbox_office ::
  ((Either (Edate) (()), (Ecountry, Evalue)) →
    (Either (Edate) (()), (Ecountry, Evalue)))
     → Ebox_office → Ebox_office
mapEbox_office f = Ebox_office ∘ f ∘ unEbox_office
newtype Edate = Edate{ unEdate :: String}
instance Typeable Edate where
  typeof = Data "date" (EP unEdate Edate) typeof
instance Show Edate where
  show (Edate a) = "Edate " ++ show a
mapEdate :: (String → String) → Edate → Edate
mapEdate f = Edate ∘ f ∘ unEdate
newtype Ecountry = Ecountry{ unEcountry :: String}
instance Typeable Ecountry where
  typeof = Data "country" (EP unEcountry Ecountry) typeof
instance Show Ecountry where
  show (Ecountry a) = "Ecountry " ++ show a
mapEcountry :: (String → String) → Ecountry → Ecountry
mapEcountry f = Ecountry ∘ f ∘ unEcountry

newtype Evalue = Evalue{ unEvalue :: Int }

instance Typeable Evalue where
  typeof = Data "value" (EP unEvalue Evalue) typeof

instance Show Evalue where
  show (Evalue a) = "Evalue " ++ show a

mapEvalue :: (Int → Int) → Evalue → Evalue
mapEvalue f = Evalue ∘ f ∘ unEvalue

newtype Eactor = Eactor{ unEactor :: (Ename, [Eplayed]) }

instance Typeable Eactor where
  typeof = Data "actor" (EP unEactor Eactor) typeof

instance Show Eactor where
  show (Eactor a) = "Eactor " ++ show a

mapEactor ::
  ((Ename, [Eplayed]) → (Ename, [Eplayed])) → Eactor → Eactor
mapEactor f = Eactor ∘ f ∘ unEactor

newtype Ename = Ename{ unEname :: String }

instance Typeable Ename where
  typeof = Data "name" (EP unEname Ename) typeof

instance Show Ename where
  show (Ename a) = "Ename " ++ show a

mapEname :: (String → String) → Ename → Ename
mapEname f = Ename ∘ f ∘ unEname

newtype Eplayed = Eplayed{ unEplayed ::
  (Etitle, (Eyear, (Erole, [Eaward]))) }

instance Typeable Eplayed where
  typeof = Data "played" (EP unEplayed Eplayed) typeof

instance Show Eplayed where
  show (Eplayed a) = "Eplayed " ++ show a

mapEplayed ::
  ((Etitle, (Eyear, (Erole, [Eaward]))) →
    (Etitle, (Eyear, (Erole, [Eaward]))))
      → Eplayed → Eplayed
mapEplayed f = Eplayed ∘ f ∘ unEplayed

newtype Erole = Erole{ unErole :: String }

instance Typeable Erole where
  typeof = Data "role" (EP unErole Erole) typeof

instance Show Erole where
  show (Erole a) = "Erole " ++ show a

mapErole :: (String → String) → Erole → Erole
mapErole f = Erole ∘ f ∘ unErole
```

```
newtype Eaward = Eaward{unEaward :: (Eaward_name, Eresult)}
instance Typeable Eaward where
  typeof = Data "award" (EP unEaward Eaward) typeof
instance Show Eaward where
  show (Eaward a) = "Eaward " ++ show a
mapEaward ::
  ((Eaward_name, Eresult) → (Eaward_name, Eresult)) →
    Eaward → Eaward
mapEaward f = Eaward ∘ f ∘ unEaward
newtype Eaward_name = Eaward_name{unEaward_name :: String}
instance Typeable Eaward_name where
  typeof = Data "award_name" (EP unEaward_name Eaward_name) typeof
instance Show Eaward_name where
  show (Eaward_name a) = "Eaward_name " ++ show a
mapEaward_name :: (String → String) → Eaward_name → Eaward_name
mapEaward_name f = Eaward_name ∘ f ∘ unEaward_name
newtype Eresult = Eresult{unEresult :: String}
instance Typeable Eresult where
  typeof = Data "result" (EP unEresult Eresult) typeof
instance Show Eresult where
  show (Eresult a) = "Eresult " ++ show a
mapEresult :: (String → String) → Eresult → Eresult
mapEresult f = Eresult ∘ f ∘ unEresult
parseEimdb :: Document Posn → Eimdb
parseEimdb = xml2values typeof
fXPath :: Eimdb → [Dynamic]
fXPath
   = ((listmap (mkDyn ∘ (fst ∘ unEmovie))) ∘ (index 1 ∘ (fst ∘ unEimdb)))
main :: IO ()
main
   = getXMLFile ≫= xmlmparse ≫=
     sequence_ ∘ map (putStrLn ∘ showResultSpace ∘ fXPath ∘ parseEimdb)
```