XPTO
An XPath Preprocessor with Type-aware Optimization

Flávio Ferreira and Hugo Pacheco

Departamento de Informática
Universidade do Minho, Braga, Portugal,
`{flavioxavier|hpacheco}@di.uminho.pt`

Abstract. Various languages allow specific query languages for selection and transformation of portions of documents. Such structure-shy queries are defined generically for different data types, and only specify specific behaviours for a few relevant subtypes. This is, for example a well-known feature of XML query languages, that allow selection of element nodes without specifying exactly the path to these elements.
We have implemented a system for performing optimizations on XPath expressions by compilation into schema-specialized programs. The core of the system consists of a combinator library, based on algebraic laws for transformation of structure-shy and XPath specific features 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 queries. The front-end for this system transforms an XML Schema file and an XPath query into an internal representation where optimizations are preformed and generates an Schema specific query program written in the functional language Haskell. The front-end itself is implemented in Haskell.

Keywords Haskell, XPath, schema-specialization, optimization, preprocessor

1 Introduction

Developed for sharing data across different information systems, the XML markup language structures the data in a tree-based representation that can be stored it in regular text files. XPath is a simple query language for using XML that is 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, and understandable, by focusing on the essence of the algorithm rather than oozing with boilerplate code [13]. However, structure shyness reduces the efficiency of queries, due to the need to traverse the whole document when looking for particular elements and the need to perform dynamic checks to determine whether to apply specific or generic behavior for each element node.

Various efforts have been made in the last few years to improve XPath efficiency [15,8]: one of them is schema-based optimization [12]. Schema-awareness is usualy 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 transforming XPath structure-shy queries into structure-sensitive point-free functional programs [3]. This system relies upon a set of algebraic laws, formulated for transformation of point-free programs [2]. 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.

Much of the theoretical work, including the Haskell implementation of the rewrite system itself, has been explored in a previous paper [3]. We will focus on parsing and outputting the optimized queries and the infrastructure needed harnesses this work into a fully functional XPath preprocessor.

This system is particular useful when the same query needs to be run against multiple 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(Figure 1) 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.

In Section 2, we present some concrete examples to motivate our approach. In Section 3, we explain in detail our implementation and show how the front-end can be used for tackling different scenarios. In Section 4, we briefly present the formalization of XPath axis and some of the algebraic laws for specialization into point-free structure-sensitive programs. Later, in Section 5, we perform some tests on our tool and compare it with other XPath processors. We end with a discussion of related work (Section 6) and concluding remarks (Section 7).

2 Motivating Examples

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 2 representing documents that hold information about movies and actors.

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 *descendant-or-self* XPath axis is structure-shy, in the sense that it does not specifies the exact path to reach *title* elements

The following grammar describes a very resumed XPath syntax:

xpath	:= *expr* (`','` *expr*) ?
expr	:= *unionexpr* | *numliteral*
unionexpr	:= *expr* `'union'` *expr*
location	:= `'/'` ? (*step* (`'/'` *step*)∗)
step	:= *axis* `'::'` *test pred* ∗
axis	:= `'child'` | `'descendant'` | `'self'` | `'descendant-or-self'`
test	:= *name* | `'*'` | `'text()'` | `'node()'`
pred	:= `'['` *xpath* `']'`
name	:= *any document tag*

The full syntax is available in the XPath language reference [19]. 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*.

Fig. 1. Summary of XPath.

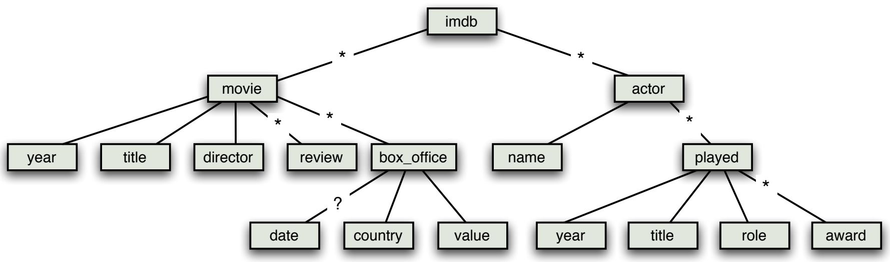

Fig. 2. A movie database schema, inspired by IMDb (http://www.imdb.com/).

from the document's *imdb* root element. From the user's perspective, structure-shyness helps specifying queries that are become more understandable, concise, and adaptative to other schemas. However, we would like to optimize the query by taking into account information about the schema. Knowing that *title* elements can occur under *movie* and *played* elements, we want to derive an optimized query similar to:

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

Retrieve all movie actors from the document

```
//movie/actor
```

This query asks to retrieve every *actor* elements that are direct children of *movie* elements appearing at any depth in the document's tree. This is an example of a query that, for this particular schema, can be simplified to an empty query because there is no *actor* element under *movie*.

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. Merging the results means applying the set union over the result sets. XPath's result sets have order, what means that all *title* tags will appear before *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.

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 3.

3 Front-end

Figure 3 defines the architecture of our tool, that can be divided into six functional blocks: parser, transformation, generation, compilation, execution and evaluation.

The *parsing* block consists on parsers for XPath and XML Schemas specifiers. They involve the conversion of parsed abstract syntax trees into our Haskell type-safe representation.

The *transformation* block is the kernel of our tool and specializes XPath queries into optimized point-free expressions. As already mentioned, this kernel was developed in previous work.

The optimized query can then be outputted as Haskell source code (*generation*), along with the constraining type definition, and furtherly *compiled and linked* with a XML parser into a program and *executed* with one or more XML argument files. It is also possible to ignore the last three blocks, by *evaluating* the resulting point-free function directly from our tool.

Parsers 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. In the conversion to our type-safe representation, XML and XSD parsers were inherited from the 2LT project [1]: for XML we use the HaXml parser, implemented with monadic parser combinators[9], and for XSD we define XML Schema instances for HaXml (XML Schemas are themselves XML files). The XPath parser and pretty-printer were hand-crafted and implemented with the fast combinator parser library Parsec [14], based on predictive LL(1) grammars. They are XPath 2.0 compliant and support the full specification [19].

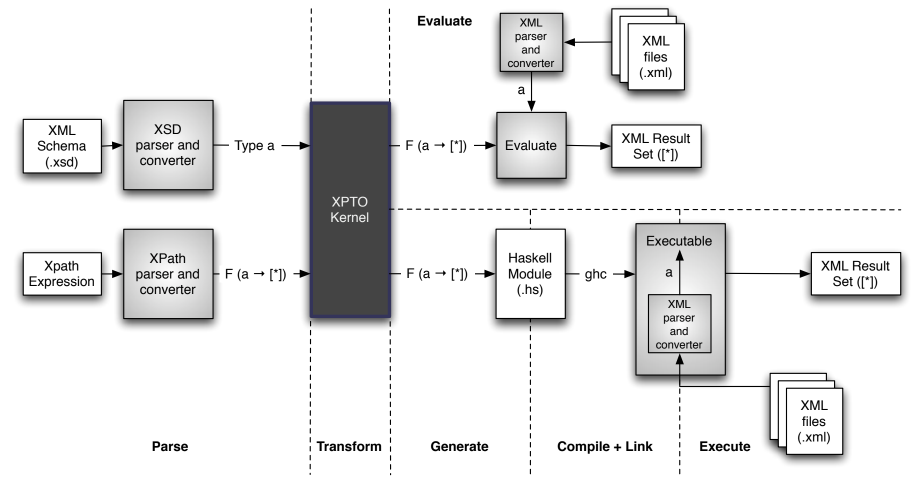

Fig. 3. Architecture of our tool

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 allows the lazy parser to stop when the further elements are not included in the result set. 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 dramatically improve parsing times, such as the third presented example, where the selected element must belong to one of the initial occurrences on the left branch of the XML tree. Parsing is also avoided when the query is optimized to an empty query.

4 Rewrite System

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. We will 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 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 powerful generalization of Haskell data types [17]. For all parameterized data $\mathit{Type}\ a$, their inhabitants must be representations of type a.

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.

The developed rewrite system is defined as a composition of strategies, that are themselves smaller rewrite systems. Strategies for this rewrite system are type-preserving, and can be encoded in Haskell as monadic functions. Constructors can be defined for different approaches like point-free, strategic or Xpath representations.

Our strategy is defined as specialization of the XPath strategic combinators into point-free functional programs, and a powerful set of laws needs to be applied in order to guarantee that the resulting functions are on the canonical form, this means, do not contain any redundancy.

For a detailed explanation on the representation of types,functions and XPath expressions and to the strategy used in the rewriting of queries, please refer to [3] or to the extended version of this paper in `http://haskell.di.uminho.pt/2ltdocs/xpto/xpto.pdf`.

We will demonstrate how to apply the specialization to each of the examples initially presented. The first step is to parse the input schema into our type representation:

$$\boxed{\left.\begin{array}{l} \mathit{fst} \circ f \triangle g = f \\ \mathit{snd} \circ f \triangle g = f \end{array}\right\} \times\text{-}\textsc{Cancel}}$$

Fig. 4. Example of a point-free law.

$t = \mathit{xsd2type}$ `"../examples/imdbNoTVDir.xsd"`

After that we have to, for each query, parse the XPath query into our functional XPath representation:

$$\begin{array}{l}
\mathit{q1} = \mathit{imdb} \,/\, ((\mathit{child} \,/\, \mathit{movie} \,/\, \mathit{child} \,/\, \mathit{title}) \bigcup (\mathit{child} \,/\, \mathit{actor} \,/\, \mathit{child} \,/\, \mathit{played} \,/\, \mathit{child} \,/\, \mathit{title})) \\
\mathit{q2} = \mathit{descself} \,/\, \mathit{child} \,/\, \mathit{movie} \,/\, \mathit{child} \,/\, \mathit{actor} \\
\mathit{q3} = \mathit{descself} \,/\, \mathit{child} \,/\, \mathit{movie} \,/\, ((\mathit{child} \,/\, \mathit{title}) \bigcup (\mathit{child} \,/\, \mathit{review})) \blacktriangleright \mathit{index}\ 3
\end{array}$$

Finally, we convert the XPath structure-shy queries into point-free structure-sensitive functions.

For the first example, the query has been divided into two expressions, one for each possible occurrence of the tag *title* in the document. In point-free notation (Figure 4), *fst* and *snd* take the first and second elements of a pair, and *split* generates a pair by application of two distinct functions to a value. As long as our representation of XML elements is opaque, we need to define a generic decapsulator unX, where X is any element, opens the content of a type. Since XPath result sets are untyped, *mkDyn* allows us to represent different types as the same type. Note that the outer *listcat* is concatenating the result of these two expressions.

$$pf2 = listcat \circ ((listmap\ (mkDyn \circ fst \circ unEmovie) \circ fst \circ unEimdb) \triangle (concat \circ (listmap\ (listmap\ (mkDyn \circ fst \circ unEplayed) \circ snd \circ unEactor) \circ snd \circ unEimdb)))$$

For the second example, as expected, the query was reduced to a void path, and always returns empty.

$$pf1 = listnil$$

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

$$pf3 = index\ 3 \circ concat \circ listmap\ (listcat \circ ((wrap \circ mkDyn \circ fst \circ unEmovie) \triangle (listmap\ mkDyn \circ fst \circ snd \circ snd \circ unEmovie))) \circ fst \circ unEimdb$$

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.

$$t' = serializeTypeSmart\ t$$

For last, we may generate an Haskell module with the point-free query and the schema's type definition. The boolean argument sets if we allow laziness in parsing or not.

$$prettyPrint\ (type2HsModule\ True\ t'\ pf)$$

5 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[10]. Next we present some benchmarking tests on the example XPath queries. [1]

[1] 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.

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

Here, we represent XML element tags from the schema's type definition from Figure 2 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.

Fig. 5. Haskell datatypes for the schema of Figure 2.

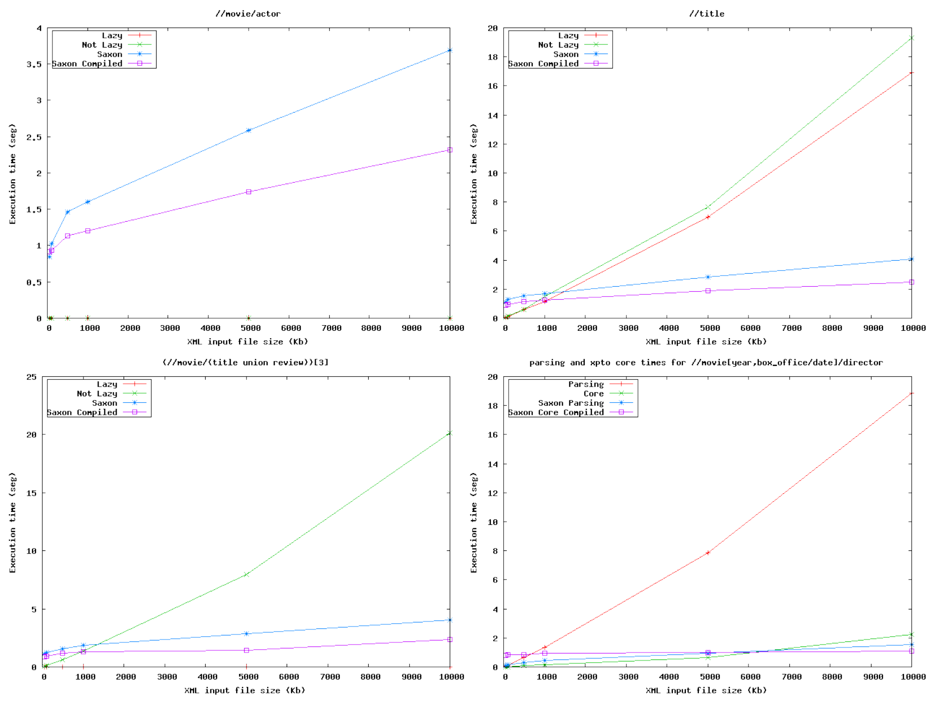

Fig. 6. Comparison tests between our tool and Saxon SA and profiling times.
2

By analysis of the results in Figure 5, we can see that Saxon has a reasonably high starting time of $\sim 0.80s$ for all the queries. However, it proves to be very efficient in general. Parsing time grows linear with the XML document size, and execution times are almost constant for queries with different features.

Our tool performs very differently. 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 processing is instantaneous. 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 do not require evaluating the whole document, lazy parsing proves to make a significant difference, depending on the relative position of the queried elements in the document.

Concluding, despite the lack of optimizations and specializations, Saxon SA proves to be faster than our implementation, which can still be greatly improved by refining transformation strategies and internal representations. Parsing times have a great influence on the results ($> 90\%$ of the total time), and Java language is a much faster language than Haskell in handling large structures.

However, the most relevant feature for determining this theory's success is the precise cost of evaluating an XPath query in relation to the query's complexity.

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 [7].

6 Related work

Type-directed partial evaluation Partial evaluation is a technique for specializing programs with knowledge 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 [4], and more recently Ens-Lyon [18], present type-directed partial evaluators based on typed lambda-calculus. 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, 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.

Saxon compiled queries Kay is the creator of Saxon [10], 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 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 [11], meaning that the optimizations are mostly related to java code and algorithmia. Schema-awareness implies validating the input and output documents against a schema 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* [6] propose a method for normalizing XML queries into a minimal "core" language, as specified in the XPath/XQuery formal semantics [5]. 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 [16]. Normalized XPath queries in the "core" language belong to a state-less 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, such as, avoid evaluating backward axis by converting them to the most similar forward axis representation.

7 Concluding Remarks

In this paper, we discuss the practical application of our Haskell-based Xpath optimization system and the achieved performance. More information on the transformation theory used in this tool can be found at [3] or in the extended version available at `http://haskell.di.uminho.pt/2ltdocs/xpto/xpto.pdf`.

In particular, we have embedded the general transformation kernel into an XPath transformation framework, by creating specific rules for handling XPath combinators. The developed framework also embeds a complete XPath 2.0 parser and XML parsing support, wrapped inside a front-end for evaluating XPath queries based on schema-validation and automated generation of structure-sensitive Haskell programs.

At last, we illustrate by example how the framework can be used to optimize different queries.

7.1 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.

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 existent *Type* representation should be extended in order to support type constraints.

XML Parsing Parsing times represent the most of 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.

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 [3], the developed rewrite system could be adapted and linked with the two-level transformation framework, in order to allow optimization of the 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 target model.

Acknowledgments

Thanks to Alcino Cunha and Joost Visser for the dedication and inspiring discussions about representation and transformation of XPath.

References

1. 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.

2. A. Cunha and J. Sousa Pinto. Point-free program transformation. *Fundam. Inform.*, 66(4):315–352, 2005.
3. 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*, 2007.
4. 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.
5. D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. *W3C Working Draft, February*, 2005.
6. Pierre Genevès and Kristoffer Rose. Compiling xpath into a state-less forward-only subset. Technical report, IBM T. J. Watson Research Center, 2004.
7. 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.
8. 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.
9. G. Hutton and E. Meijer. Monadic parser combinators. *Journal of Functional Programming*, 8(4):437–444, 1996.
10. Michael Kay. Saxon: Anatomy of an xslt processor. In *IBM developerWorks*, 2001.
11. Michael Kay. Xslt and xpath optimization. In *XML Europe*, 2004.
12. A. Kwong and M. Gertz. Schema-based optimization of XPath expressions. *Submitted for publication, available from the authors*, 2002.
13. R. Lämmel, E. Visser, and J. Visser. Strategic programming meets adaptive programming. *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 168–177, 2003.
14. 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.
15. Philippe Michiels. Xquery optimization. Technical report, University of Antwerp, Belgium, 2003.
16. Dan Olteanu, Holger Meuss, Tim Furche, and François 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.
17. 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.
18. K.H. Rose. Type-directed partial evaluation in Haskell. *"Preliminary proceedings of the 1998 APPSEM workshop on normalization by evaluation. BRICS Notes, nos. NS 981"*, 1998.
19. W3C. XML path language (XPath) 2.0, W3C candidate recommendation, 2006.