RC24075 (W0610-062) October 13, 2006
Computer Science

IBM Research Report

AXIL: An XPath Intermediate Language

Christoph Reichenbach*, Michael Burke, Igor Peshansky,
Mukund Raghavachari, Rajesh Bordawekar
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

*University of Colorado at Boulder

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

AXIL: An XPath Intermediate Language

Christoph Reichenbach*
⟨reichenb@colorado.edu⟩
University of Colorado at Boulder†

Michael Burke
⟨mgburke@us.ibm.com⟩
IBM Research

Igor Peshansky
⟨igorp@us.ibm.com⟩
IBM Research

Mukund Raghavachari
⟨raghavac@us.ibm.com⟩
IBM Research

Rajesh Bordawekar
⟨bordaw@us.ibm.com⟩
IBM Research

October 3, 2006

Abstract

XPath is a central component of many XML-processing languages; therefore, it is important to process XPath queries efficiently. We describe AXIL, a functional intermediate language, which allows us to simplify the task of optimising and compiling XPath for efficient execution. We describe interesting parts of the syntax and semantics of AXIL and particular design issues which shaped this language, sketch the translation from XPath into AXIL, and discuss our experience with using AXIL for an optimising XPath compiler backend.

1 Introduction

XPath 1.0 [CD99] is a language for describing sets of nodes of XML documents, relative to a given node within the document, the so-called *context node*. XPath allows queries such as "`./ancestor::*`", which selects all ancestor nodes of the context node (i.e., the parent of the context node, the parent of that parent and so on, until the root of the document is reached), or

```
./book[./author="Franz Kafka"][last()]
```

which selects all nodes of name "book" underneath the context node, if they satisfy two conditions: First, the "book" node must have at least one

*This work was supported by NSF Career Grant CCR-0133457
†Work performed at the IBM TJ Watson Research Center

"author" node containing the text "Franz Kafka". Second, the "book" node must be the *last* node to satisfy this first condition— that is, we select (at most) one book node, namely the very last one which lists "Franz Kafka" as an "author" (A more formal account of XPath 1.0 semantics can be found in [PW99, Wad99]).

In practice, XPath is a central component of many languages with native support for XML processing [HRS+05, DFF+05, Cla99]. Thus, it is important to be able to process XPath queries efficiently.

While it is quite possible to compile XPath 1.0 [CD99] queries directly to native code in one form or another, XPath 1.0 syntax trees leave much information implicit that is of relevance to compilation, particularly where compiler optimisations are concerned. For example, XPath predicate expressions' types can be inferred statically, which allows us to manifest coercions, freeing the run-time system from the need to detect these dynamically.

As a more complex example, consider a second query, applied to the same context node as the one listed above:

```
./article
```

To evaluate both queries, the same set of nodes must be traversed, while filtering according to different criteria. If we know that the results of both queries are needed, we can optimise their evaluation by taking advantage of this shared information.

Core XQuery, for example, allows us to split this up into first computing `./*` ("get all child nodes") and then passing the resulting set (or sequence, in XQuery terms) to two distinct filters, which can be expressed as XQuery `for` expressions. While this allows us to re-use the common *result* of the query, it does not help us re-use the *iteration* over the nodes— we will iterate twice over all child nodes.

A direct translation into and from Core XQuery would also not yield as efficient code as possible in other instances (e.g., consider the `preceding-sibling` and `following-sibling` axes (Section 4.2.1) select all siblings, then filter them according to whether they precede or succeed the context node in document order). A similar criticism could be applied to Wadler's partial XPath semantics [Wad99], where for example the context position is computed as the size of the set of all preceding nodes, if we were to use it as an implementation recipe.

In the following, we describe an intermediate language, AXIL, which allows us to express the kind of sharing noted above, covers the entirety of the XPath 1.0 functionality while having fewer constructs than XPath or Core XQuery, and furthermore is "pure" in a functional sense, allowing us to easily compile it into lazily (on-demand) or strictly (eagerly) evaluating code.

We begin by giving the design rationale for AXIL in Section 2, followed by its syntax and excerpts of its denotational semantics in Section 3, before discussing our current AXIL-based code generator (in Section 4). Preliminary performance results are presented in Section 5 before discussing related work in Section 6 and future work in Section 7.

2 Language Design Rationale

Our goals in designing AXIL were to simplify compilation, and to allow (at least) the following optimisations to be expressible:

- Support for expressing sharing not only of values, but also of iterations (loop fusion)
- Support for deforestation [MW92], which allows us to avoid the construction of intermediate node sets
- Support for standard hoisting and partial evaluation optimisations, such as constant folding
- A choice between strict and lazy evaluation (or a hybrid of both), allowing us to compile to either form. Both approaches have their merits— strict evaluation (eager execution) has minimal overhead and is thus usually more efficient if all results of the resulting node set are used, whereas lazy evaluation (partial on-demand computation) can avoid unneeded computations if not all result nodes are asked for. Also, lazy evaluation uses less memory on large result sets, which may improve cache behaviour.

When considering an intermediate language, a pure functional design seems to be a good "rule of thumb", as we are making all side effects and implicit value dependencies explicit. For example, Static Single Assignment (SSA) form [CFR+91], commonly used to optimise imperative code, is essentially a functional language [App98]. Explicit side effects and dependencies allow us to see opportunities for optimisation (and obstacles of such) more clearly, and to use many standard optimisation techniques for functional programming languages [App92, Jon88].

In our context, a pure functional design appears to be appropriate, as pure functional languages yield the same result when evaluated lazily and when evaluated eagerly. Thus, two of our goals were handled by this choice.

We chose as primitive types the following:

- $\mathbb{D}$: The type of IEEE 754 double-precision numbers
- $\mathbb{S}$: The type of Unicode character strings
- $\mathbb{B} \triangleq \{\mathbf{true}, \mathbf{false}\}$: The type of boolean values
- Node: The type of all nodes
- NodeSet: The type of all sets of nodes
- NCache: The type of all caches of nodesets (isomorphic to NodeSet)
- Node$^?$: The type of all nodes, lifted

All of these types have very concrete representations in practice, the sole exception being NodeSet, and NCache, for which a large number of possible implementations was conceivable. Node$^?$ represents what programmers of object-oriented languages would typically interpret as a tree node: Either a "node object", or *nothing* (e.g., `null` in Java). We found it useful to distinguish this frequently occurring type from Node, as *nothing* requires special handling.

2.1 Deforestation

For the purpose of our initial set-based semantics, we interpret NodeSet as the type of sets of nodes; in practice, we interpret it as a *stream* or *lazy list*. This interpretation allows us to make use of *deforestation* [MW92]. The idea behind deforestation is to avoid intermediate structures (lists, in our case) by composing operations performed during repeated recursion in a way that reduces the number of recursions. For example, an analogous idea in an imperative language would transform

```
FOR i := 0 TO length(l) DO
    l[i] := l[i] * 2;

FOR i := 0 TO length(l) DO
    l[i] := l[i] + 1;
```

to the semantically equivalent

```
FOR i := 0 TO length(l) DO
    l[i] := (l[i] * 2) + 1;
```

In [MW92], such optimisations are developed through recursion; to avoid recursion in our language, we instead used two "standard" higher-order functions, for which deforestation-derivable rewritings are known:

- filter, which removes all nodes that do not match a certain predicate, and
- fold, which aggregates values[1].

2.2 Common Iterations

To more efficiently support common iterations, we introduced a special primitive, ($\otimes$), to tag computations over a common NodeSet. We give the definition of this primitive and explain its usage in Section 3.9.

2.3 Tree Traversals

For expressing traversals over XML trees, we chose primitives which were present in the DOM [HHW+00] representation of XML trees, plus a repetition construct (the Kleene Star), after observing that all but one axis (cf. Section 7.2) could be expressed with these in a way that guaranteed node uniqueness and document order (some of these observations can also be found in the core XQuery semantics [DFF+05]).

DOM, or tree-structured data in general, is not necessarily an ideal representation. Consider the XPath query `child::node()[42]`, which selects the 42nd child of the context node: Such an access takes 43 steps (getting the first child and 42 iterations of getting the next sibling) in a tree representation, but only one (checked) array read if XML trees are represented as arrays. In our design, however, we decided not to rely on a vector-based representation; applied to such a representation, our design should be modified.

[1] Functional programmers will note that filter can be expressed in terms of fold, but only if list construction is allowed.

2.4 Namespaces

AXIL provides no specific facilities to simplify namespace support. However, given representational support (as provided e.g. by the W3C DOM representation [HHW+00]), it is easy to provide run-time facilities which properly deal with namespaces, i.e., which translate namespace prefixes into the associated namespace URIs. For any given XPath query, any such association must be known before we begin executing the query, i.e. they are passed as parameters into the AXIL expression; there, they can be used directly as part of name comparisons, being compared with the result of the namespace-uri function.

3 Syntax and Semantics of AXIL

AXIL provides what amounts to a small number of predefined higher-order functions. To avoid having to worry about the semantics of higher-order functions in general, we treat them as special constructs. This gives us a relatively simple type system, with which we begin our exposition. After explaining our type system, we move on to give syntax and semantics of the language itself. Note that the semantics we are defining here is completely set-based; since the data we are dealing with can safely be assumed to be finite, we need not worry about non-termination and, therefore, domain constructions. However, as we will discuss later, set-based semantics are not the only useful interpretation of the language.

3.1 Syntax of Types

We define AXIL in terms of a standard monomorphic type system, over the productions defined for the nonterminal T below:

$$\begin{array}{rcl} (T_a) & ::= & \mathbb{B} \mid \mathbb{D} \mid \mathbb{S} \mid \textsc{Node} \mid \textsc{Node}^? \mid \textsc{NodeSet} \\ (T_p) & ::= & (T_a) \mid (T_a) \times (T_p) \\ (T) & ::= & (T_p) \mid (T_p) \to (T_p) \end{array}$$

Recall that our primitive types, (T_a), were outlined in Section 2.

Here, the type constructor $(\times)$ constructs tuple types, and $(\to)$ constructs function types; typing rules are given in Figure 1.

Note that certain primitives do benefit from polymorphism; to allow for this, we introduce special typing rules to deal with them.

3.2 Language Syntax

We begin the value syntax with atomic values (i.e., constants and variables):

$$
\begin{array}{rcl}
(V) & ::= & (V_{\mathbb{B}}) \mid (V_{\mathbb{D}}) \mid (V_{\mathbb{S}}) \mid (\textsc{Name}) \mid (\mathbb{V}) \\
(V_{\mathbb{B}}) & ::= & \texttt{true} \mid \texttt{false} \\
(V_{\mathbb{D}}) & ::= & 0.0 \mid 1.0 \mid \ldots \\
(V_{\mathbb{S}}) & ::= & \varepsilon \mid \texttt{"foo"} \mid \ldots \\
(\textsc{Name}) & ::= & a \mid b \mid c \mid \ldots \\
(\mathbb{V}) & ::= & \$i \mid \$j \mid \ldots
\end{array}
$$

We explicitly distinguish between XPath-style variables passed in from an external environment (production $(\mathbb{V})$), and names defined within AXIL (production $(\textsc{Name})$), as these may be treated differently by AXIL implementations.

$$
\begin{array}{rcl}
(Mult) & ::= & + \mid * \\
(IName) & ::= & \square \mid (\textsc{Name}) \\
(ISeq) & ::= & (IName) \mid (IName), (ISeq) \\
(Input) & ::= & (IName) \mid \langle\ (ISeq)\ \rangle \\
(NKind) & ::= & \textsc{Node} \mid \textsc{Elt} \mid \textsc{Attr} \mid \textsc{Text} \mid \texttt{PI} \\
 & \mid & \textsc{Namespace} \mid \textsc{Comment} \\
(LR) & ::= & \leftarrow \mid \rightarrow \mid \divideontimes
\end{array}
$$

$$
\begin{array}{rcl}
(Fun) & ::= & (Builtins) \\
 & \mid & \mathsf{filter}^{(LR)}[ntInput \Rightarrow (Expr)] \\
 & \mid & \mathsf{guard}^{(LR)}[ntInput \Rightarrow \langle (Expr), (Expr) \rangle] \\
 & \mid & \mathsf{fold}^{(LR)}[(Input) \Rightarrow (Expr)](V) \\
 & \mid & (Fun) \circ (Fun) \\
 & \mid & (Fun) \bullet (Fun) \\
 & \mid & (Fun)^{(Mult)} \\
 & \mid & (Fun) \otimes (Fun) \\
(Expr) & ::= & (V) \\
 & \mid & (Fun) \triangleleft (Expr) \\
 & \mid & [(Expr)]_{(T_p)}^{(T_p)} \\
 & \mid & (Fun)(Expr) \\
 & \mid & \textbf{let}\ (Input) = (Expr)\ \textbf{in}\ (Expr)
\end{array}
$$

Here, nonterminal (*Fun*) describes functions in the intermediate language; the production

$$(\mathit{Expr}) \quad ::= \quad (\mathit{Fun})(\mathit{Expr})$$

then describes function application.

Nonterminal (*Builtins*) refers to a pre-defined set of function symbols. We list a representative subset of these in figure 2, along with their types. Note that none of the omitted functions deals with NodeSets. Since several XPath 1.0 functions are described in terms of sets of nodes, we encode these functions in fold constructions (cf. Section 3.8).

Note that function symbols set in parentheses will be used in infix notation here; thus, we write $\texttt{true} \vee \texttt{false}$ instead of $(\vee)(\texttt{true}, \texttt{false})$.

3.3 Set-based Denotational Semantics

We now define the various constructions listed in our grammar in terms of set theory, giving examples for some of the more involved constructions.

Identifying our types with their obvious set-theoretic interpretations, we define the set of values of our language as

$$\textsc{Val} \triangleq \textsc{Node} \uplus \textsc{NodeSet} \uplus \mathbb{D} \uplus \mathbb{S} \uplus \mathbb{B}$$

We will interpret $\textsc{Node}^?$ as the subset of NodeSet with no more than one node (interpreting occurrences of *nothing* as $\emptyset$).

3.3.1 The Interpretation Function

An environment $V : \textsc{Env}$, where

$$\textsc{Env} \triangleq (\mathbb{V} \uplus \textsc{Name}) \to \textsc{Val}$$

is a mapping from names to values. This environment initially carries all bindings for free variables in our expression; later updates are denoted as $V[n \mapsto S]$, with the intended semantics specified by the following two formulae:

$$V[n \mapsto S](n) = S \tag{1}$$

$$x \neq n \quad \Longrightarrow \quad V[n \mapsto S](x) = V(x) \tag{2}$$

Our interpretation function,

$$[\![-]\!] : (\mathit{Expr}) \times \textsc{Env} \to \textsc{Val}$$

interprets expressions $E : (\mathit{Expr})$ in terms of a value environment $V : \textsc{Env}$, denoted $[\![E]\!](V)$. For simplicity, we shall write $[\![E]\!]$, omitting the environment, if the environment is not relevant for the given context. In this case, the same environment can be assumed to be applied to all interpretations in the same context implicitly.

For brevity, we shall omit or only sketch interpretations of some constructs we consider obvious.

$$\frac{}{E, n : \tau \vdash n : \tau} \qquad \frac{E \vdash x_1 : \tau_1 \quad E, n : \tau_1 \vdash x_2 : \tau_2}{E \vdash \textbf{let } n = x_1 \textbf{ in } x_2 : \tau_2}$$

$$\frac{E \vdash f : \tau \to \textsc{Node}^?}{E \vdash \triangleright f : \tau \to \textsc{NodeSet}} \qquad \frac{E \vdash f : \tau_1 \to \tau_2 \quad E \vdash x : \tau_1}{E \vdash fx : \tau_2}$$

$$\frac{v \in \mathcal{S} \quad \mathcal{S} \in \{\mathbb{D}, \mathbb{B}, \mathbb{S}\}}{E \vdash v : \mathcal{S}} \qquad \frac{(f : t) \in \textsc{Builtins}}{E \vdash f : t}$$

$$\frac{E \vdash s : \textsc{Node} \to \textsc{NodeSet}}{E \vdash s^* : \textsc{Node} \to \textsc{NodeSet}} \qquad \frac{E \vdash s : \textsc{Node} \to \textsc{NodeSet}}{E \vdash s^+ : \textsc{Node} \to \textsc{NodeSet}}$$

$$\frac{E \vdash x_1 : \tau_1 \times \ldots \times \tau_k \quad E, n_1 : \tau_1, \ldots, n_k : \tau_k \vdash x_0 : \tau_0}{E \vdash \textbf{let } \langle n_1, \ldots, n_k \rangle = x_1 \textbf{ in } x_0 : \tau_0}$$

$$\frac{E \vdash s_2 : \tau_1 \to \tau_2 \quad s_1 : \tau_0 \to \tau_1}{E \vdash s_2 \circ s_1 : \tau_0 \to \tau_2}$$

$$\frac{E \vdash s_2 : \textsc{Node} \to \tau \quad \tau \in \{\textsc{Node}^?, \textsc{NodeSet}\} \quad s_1 : \textsc{Node} \to \textsc{Node}^?}{E \vdash s_2 \bullet s_1 : \textsc{Node} \to \tau}$$

$$\frac{E \vdash x_1 : \tau_1 \quad \tau_1 \in \{\textsc{NodeSet}, \textsc{Node}, \mathbb{B}, \mathbb{S}, \mathbb{D}\} \quad \tau_2 \in \{\mathbb{B}, \mathbb{S}, \mathbb{D}\}}{E \vdash [x_1]^{\tau_1}_{\tau_2} : \tau_2}$$

$$\frac{E, n : \textsc{Node}, i : \mathbb{D} \vdash x_1 : \mathbb{B} \quad \delta \in \{\leftarrow, \rightarrow, \divideontimes\}}{E \vdash \mathsf{filter}^\delta[\langle n, i \rangle \Rightarrow x_1] : \textsc{NodeSet} \to \textsc{NodeSet}}$$

$$\frac{E, n : \textsc{Node}, i : \mathbb{D} \vdash t : \mathbb{B} \quad E, n : \textsc{Node}, i : \mathbb{D} \vdash e : \textsc{NodeSet} \quad \delta \in \{\leftarrow, \rightarrow, \divideontimes\}}{E \vdash \mathsf{guard}^\delta[\langle n, i \rangle \Rightarrow \langle t, e \rangle] : \textsc{NodeSet} \to \textsc{NodeSet}}$$

$$\frac{E, n : \textsc{Node}, i : \mathbb{D}, o : \tau \vdash x_1 : \tau \quad E \vdash v : \tau \quad \delta \in \{\leftarrow, \rightarrow, \divideontimes\}}{E \vdash \mathsf{fold}^\delta[\langle n, i, o \rangle \Rightarrow x_1]v : \textsc{NodeSet} \to \tau}$$

$$\frac{E \vdash f : \textsc{Node} \to \textsc{NodeSet} \quad E \vdash n : \tau \to \textsc{NodeSet}}{E \vdash f \triangleleft n : \tau \to \textsc{NodeSet}}$$

$$\frac{E \vdash f : \tau_1 \to \tau_2 \quad E \vdash g : \tau_1 \to \tau_3}{E \vdash f \otimes g : \tau_1 \to \tau_2 \times \tau_3}$$

Figure 1: Typing and type inference rules for AXIL expressions. The environments E used here follow the usual notational convention; note that for all variables passed into an AXIL expression the environment needs to be pre-initialised with the variable's type. The set Builtins contains all initial function definitions and their type judgements.

remove-duplicates	:	$\textsc{NodeSet} \to \textsc{NodeSet}$	$(\equiv_{\mathbb{B}})$	:	$\mathbb{B} \times \mathbb{B} \to \mathbb{B}$
cache	:	$\textsc{NodeSet} \to \textsc{NCache}$	$(\equiv_{\mathbb{D}})$	:	$\mathbb{D} \times \mathbb{D} \to \mathbb{B}$
stream	:	$\textsc{NCache} \to \textsc{NodeSet}$	$(\equiv_{\mathbb{S}})$	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{B}$
cachesize	:	$\textsc{NCache} \to \mathbb{D}$	$(<)$	:	$\mathbb{D} \times \mathbb{D} \to \mathbb{B}$
concat	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	$(\leq)$	:	$\mathbb{D} \times \mathbb{D} \to \mathbb{B}$
starts-with	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{B}$	$(\geq)$	:	$\mathbb{D} \times \mathbb{D} \to \mathbb{B}$
contains	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{B}$	$(>)$	:	$\mathbb{D} \times \mathbb{D} \to \mathbb{B}$
substring-before	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	$(+)$	:	$\mathbb{D} \times \mathbb{D} \to \mathbb{D}$
substring-after	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	$(-)$	:	$\mathbb{D} \times \mathbb{D} \to \mathbb{D}$
substring	:	$\mathbb{S} \times \mathbb{D} \times \mathbb{D} \to \mathbb{S}$	$(\cdot)$	:	$\mathbb{D} \times \mathbb{D} \to \mathbb{D}$
string-length	:	$\mathbb{S} \to \mathbb{D}$	$(/)$	:	$\mathbb{D} \times \mathbb{D} \to \mathbb{D}$
normalize-space	:	$\mathbb{S} \to \mathbb{S}$	$(\%)$	:	$\mathbb{D} \times \mathbb{D} \to \mathbb{D}$
translate	:	$\mathbb{S} \times \mathbb{S} \times \mathbb{S} \to \mathbb{S}$	$(\vee)$	:	$\mathbb{B} \times \mathbb{B} \to \mathbb{B}$
floor	:	$\mathbb{D} \to \mathbb{D}$	$(\wedge)$	:	$\mathbb{B} \times \mathbb{B} \to \mathbb{B}$
ceiling	:	$\mathbb{D} \to \mathbb{D}$			
round	:	$\mathbb{D} \to \mathbb{D}$	$\uparrow$	:	$\textsc{Node} \to \textsc{Node}^?$
not	:	$\mathbb{B} \to \mathbb{B}$	$\leftarrow$	:	$\textsc{Node} \to \textsc{Node}^?$
			$\rightarrow$	:	$\textsc{Node} \to \textsc{Node}^?$
			$\swarrow$	:	$\textsc{Node} \to \textsc{Node}^?$
get-attr-node	:	$\mathbb{S} \times \textsc{Node} \to \textsc{Node}^?$	$\searrow$	:	$\textsc{Node} \to \textsc{Node}^?$
get-attr	:	$\mathbb{S} \times \textsc{Node} \to \mathbb{S}$	$\downarrow^A$	:	$\textsc{Node} \to \textsc{NodeSet}$
$\diamond$	:	$\textsc{Node} \to \textsc{Node}$	$\downarrow^N$	:	$\textsc{Node} \to \textsc{NodeSet}$
id	:	$\mathbb{S} \to \textsc{Node}^?$	$\boxdot$	:	$\textsc{Node} \to \textsc{Node}^?$
local-name	:	$\textsc{Node} \to \mathbb{S}$			
namespace-uri	:	$\textsc{Node} \to \mathbb{S}$			
name	:	$\textsc{Node} \to \mathbb{S}$			
$\textsc{test}[(NKind)]$	:	$\textsc{Node} \to \mathbb{B}$			
$\triangleright$	:	$\textsc{Node}^? \to \textsc{NodeSet}$			

$(\otimes)$	:	$(\tau_0 \to \tau_1) \times (\tau_0 \to \tau_2) \to (\tau_0 \to \tau_1 \times \tau_2)$
$(\boxplus)$	:	$\textsc{NodeSet} \times \textsc{NodeSet} \to \textsc{NodeSet}$
$\textsc{order}^{\leftarrow}$	:	$\textsc{NodeSet} \to \textsc{NodeSet}$
$\textsc{order}^{\rightarrow}$	:	$\textsc{NodeSet} \to \textsc{NodeSet}$
$\textsc{nub}$	:	$\textsc{NodeSet} \to \textsc{NodeSet}$

Figure 2: An excerpt of the set BUILTINS of function symbols paired with their types (33 names were omitted). Note that function symbols written in parentheses will be written in infix notation.

3.4 Semantics of let and variables

The purpose of a **let** construct is to bind a name to a value. We use this construction in three cases:

- To bind names to subexpressions which have been hoisted during an optimisation phase,
- To give a name to a subexpression which has been identified as a common subexpression during optimisation, and
- To deconstruct tuples (which arise by use of the $\otimes$ operator, described below).

The semantics of this construction (for single name bindings) are as follows:

$$[\![\textbf{let } n = S \textbf{ in } E]\!](V) \triangleq [\![E]\!](V[n \mapsto [\![S]\!](V)])$$

i.e., we evaluate S, bind the name n to the result, and proceed to evaluate E with this new name binding. The deconstructing **let** binding for tuples can be defined analogously.

3.4.1 Semantics of Arrows

Arrows represent our navigational primitives:

- $\uparrow$ (n): Select the parent node of n, if present
- $\rightarrow$ (n): Select the next sibling (next-greater sibling in document order) of n, if present
- $\swarrow$ (n): Select the first child (least child in document order) of n, if present

Arrows $\leftarrow$ and $\searrow$ have analogous meanings. Note that all arrows may fail to return a result; in that case, their result is $\emptyset$. Otherwise, it is the singleton containing the result.

For attribute and namespace nodes, we use the primitives $\downarrow^A$ and $\downarrow^N$, respectively, which yield the set of all attribute nodes of n ($\downarrow^A$ (n)), or the set of all namespace nodes of n ($\downarrow^N$ (n)).

Another navigational primitive, though visually not an arrow, is $\diamond(n)$, which selects the root node of the document to which n belongs.

3.5 Semantics of Composition

Function composition ($\circ$) is quite standard. However, we also often have to compose a NODE$^?$-yielding function with a computation which requires a NODE as input; this is expressed by $\bullet$.

$$\begin{aligned} [\![f \circ g]\!] &\triangleq \lambda x.[\![f]\!]([\![g]\!](x)) \\ [\![a \bullet b]\!] &\triangleq \lambda n.\{n'' | n' \in [\![b]\!](n), n'' \in [\![a]\!](n')\} \end{aligned}$$

It is worth pointing out that $\bullet$ is precisely the monadic *bind* operation on the monad of partiality: If the first function (a) yields no result, we skip

the second function (b) and immediately yield *nothing* (or equivalently an empty set, if the second function yields a NodeSet— recall that we interpret both concepts identically).

As an example, consider $\uparrow \bullet \uparrow$: This function implements the XPath expression `parent::*/parent::*`, yielding no result if either no parent exists, or if no grandparent exists ("if the parent has no parent").

3.5.1 Semantics of Exponentials

The Kleene Plus (and Kleene Star) are used to compute the transitive (and reflexive) closure of a function. Formally,

$$\begin{aligned} [\![f^*(n)]\!] &\triangleq \bigcup_{i \in \mathbb{N}} [\![f]\!]^i([\![n]\!]) \\ [\![f^+(n)]\!] &\triangleq \bigcup_{i \in \mathbb{N} \setminus \{0\}} [\![f]\!]^i([\![n]\!]) \end{aligned}$$

where

$$\begin{aligned} f^0(n) &\triangleq n \\ f^i(n) &\triangleq f(f^{i-1}(n)) \end{aligned}$$

In practice, we use them exclusively for arrows. For example,

$$\rightarrow^+ (n)$$

computes all right siblings of n (and thus effectively implements the `following-sibling` axis). Combined with composition, we can implement the `child` axis as

$$\rightarrow^* \bullet \swarrow$$

It is also possible to nest exponentials— using this, we can describe the `descendant-or-self` axis as

$$(\rightarrow^* \bullet \swarrow)^*$$

The observant reader may have noticed that these expressions would not typecheck in the type system outlined before. According to Figure 1, any function f an exponential is applied to must have a type

$$f : \text{Node} \rightarrow \text{NodeSet}$$

Thus, for type-correctness, we must lift the result with the $\triangleright$ construction, which semantically acts as the identity; we chose to omit this detail to simplify our initial explanation. The reason for this restriction is not evident from our set-theoretical semantics, but may become clear to the reader when considering that a compiler would typically want to interpret Node$^?$s very differently from NodeSets.

3.5.2 Semantics of Other Nodeset Operations

Two other operations over NODESETs are required in certain situations. NODESET composition is straightforward:

$$[\![S_1 \boxplus S_2]\!] \quad \triangleq \quad [\![S_1]\!] \cup [\![S_2]\!]$$

To understand the need for our second primitive, note that path composition cannot be implemented by straightforward function composition:

$$(\triangleleft \rightarrow)^* \circ (\triangleleft \rightarrow)^*$$

not only does not typecheck (since $(\triangleleft \rightarrow)^* : \text{NODE} \rightarrow \text{NODESET}$) but also cannot be assigned a useful meaning: Since the right-hand side of the function composition computes a set of nodes but the left-hand side expects a single node, we need to explain more precisely what is supposed to happen in this kind of composition. Thus, we define a third composition operator:

$$[\![(f \triangleleft S)(n)]\!] \quad \triangleq \quad \{[\![f(n)]\!] | n \in [\![S(n)]\!]\}$$

The operator $\triangleleft$ thus allows us to expand a node set in a pointwise fashion.

Note that the following useful equality holds:

$$f \triangleleft (\triangleright g) \quad = \quad f \bullet g$$

Three other built-in functions, $\text{ORDER}^{\leftarrow}$, $\text{ORDER}^{\rightarrow}$ and NUB, which represent sorting in reverse document order, sorting in document order, and duplicate removal (respectively) are trivial in set-based semantics and can thus be treated as instances of the identity function.

3.6 Semantics of filter

Our navigational primitives are sufficient for indiscriminately traversing over XML trees. However, they give us no means of expressing node tests or predicates. For this purpose, we use filter, which is parameterised by a predicate and applies this predicate to all nodes of a node set passed to it, eliminating all nodes which fail the predicate. For example, we can now express `./book` as the following construction[2]:

$$\begin{aligned} \mathsf{filter}^{*}\,[\langle n, \square\rangle \Rightarrow \mathsf{name}(n) = \text{“book”}] \\ \circ((\triangleright \rightarrow)^{*} \bullet \swarrow) \end{aligned}$$

The second line simply describes “get-all-children”, but the first line expresses our filter construction. First, note our predicate, which is the entire expression in brackets:

$$[\langle n, \square\rangle \Rightarrow \mathsf{name}(n) = \text{“book”}]$$

[2]For proper semantics, we must also test for the correct node type. We omit this test for brevity.

Since the predicate needs to be parameterised by the node we are analysing, we first bind the node to a name, "n" in our example, as expressed in the tuple to the left of the double arrow. Here, the tuple also contains a "□": We can optionally bind a name to our current position within the node set (which is useful e.g. if we want to pick precisely the 23rd node), and if this information is of use to us, we must specify a name here. "□" indicates that we do not care about this information (which is a useful hint for compilation).

Also note the exponent attached to filter: This may be $\divideontimes$ (as in our example), which indicates that the order in which the input node set is processed is irrelevant, or one of $\rightarrow$ and $\leftarrow$, which indicates a request to iterate through the nodes in document order or in reverse document order (respectively). As mentioned before, we can bind a name to our current position in the node set— however, this position is meaningless unless we specify such an order, and we disallow this binding if our iteration order is $\divideontimes$.

To describe the semantics of filter (and fold), we need to define an auxiliary function, $\mathsf{pick}_\delta(-) : \textsc{NodeSet} \rightarrow \textsc{Node}$, which retrieves the "next" element out of the node set in the prescribed order. Its definition employs the Axiom of Choice in the case of $\delta = \divideontimes$, which we denote by a function $\mathsf{choice} : \textsc{NodeSet} \rightarrow \textsc{Node}$:

$$\begin{aligned}\mathsf{pick}_{\rightarrow}(S) &= \mathsf{inf}_{\preceq}(S)\\ \mathsf{pick}_{\leftarrow}(S) &= \mathsf{sup}_{\preceq}(S)\\ \mathsf{pick}_{\divideontimes}(S) &= \mathsf{choice}(S)\end{aligned}$$

where $\mathsf{inf}_{\preceq}$ and $\mathsf{sup}_{\preceq}$ yield the singleton containing the least element in document order or the greatest element in document order (respectively), if any such element exists, and $\emptyset$ otherwise.

In the next step, we will make use of a function $\mathbb{D}(-) : \mathbb{N} \rightarrow \mathbb{D}$ which maps natural numbers to their closest equivalents in $\mathbb{D}$; this step is necessary since an infinite subset of $\mathbb{N}$ cannot be represented in $\mathbb{D}$, but we would still like to provide positional information as accurately as possible.

We are now ready to define FILTER:

$$\mathsf{filter}^{\delta}[\langle n, i\rangle \Rightarrow P](S) \quad \triangleq \quad F(S, 1)$$

where

$$\begin{aligned}F(\emptyset, c) &\triangleq \emptyset\\ F(S, c) &\triangleq \mathbf{let}\ \ e = \mathsf{pick}_\delta(S)\\ &\qquad\quad\ \ S' = F(S \setminus \{e\}, c+1)\\ &\qquad \mathbf{in}\ \ C(S', e, [\![P[\ n \leftarrowtail e,\\ &\qquad\qquad\qquad\quad i \leftarrowtail \mathbb{D}(c)]]\!])\\ C(S', e, \mathbf{true}) &\triangleq S' \cup \{e\}\\ C(S', e, \mathbf{false}) &\triangleq S'\end{aligned}$$

3.7 Semantics of guard

Guard is a generalisation of both filter and $\triangleleft$: The operation filters all incoming nodes, but rather than yielding *nodes* which succeed in the test, it returns the union of all *nodesets* generated from these nodes.

$$\mathsf{guard}^{\delta}[\langle n, i\rangle \Rightarrow \langle P, E\rangle](S) \quad \triangleq \quad F(S, E, 1)$$

where

$$\begin{array}{rcl}
F(\emptyset, E, c) & \triangleq & \emptyset \\
F(S, E, c) & \triangleq & \textbf{let} \quad e = \mathsf{pick}_{\delta}(S) \\
 & & \qquad\; S' = F(S \setminus \{e\}, c+1) \\
 & & \textbf{in} \quad C(S', \; [\![E[\; n \hookleftarrow e, \\
 & & \qquad\qquad\qquad i \hookleftarrow \mathbb{D}(c)]]\!], \\
 & & \qquad\qquad [\![P[\; n \hookleftarrow e, \\
 & & \qquad\qquad\qquad i \hookleftarrow \mathbb{D}(c)]]\!]) \\
C(S', E, \textbf{true}) & \triangleq & S' \cup E \\
C(S', E, \textbf{false}) & \triangleq & S'
\end{array}$$

3.8 Semantics of fold

fold, which functional programmers also know as reduce or foldl (for "fold-left"), allows a value to be aggregated over a node set. For example, the XPath `sum` function can be implemented follows:

$$\mathsf{fold}^{\divideontimes}[\langle n, \Box, o\rangle \Rightarrow o + [n]_{\mathbb{D}}^{\textsc{Node}}](0.0)$$

(here, $[n]_{\mathbb{D}}^{\textsc{Node}}$ is an explicit coercion from a Node to a $\mathbb{D}$, which works as described for singleton node sets in the XPath semantics).

In this example, we will return 0.0 whenever the NodeSet passed to this construction is empty. Otherwise, we will, for each node, evaluate the body once; during the first iteration, o is bound to 0.0, afterwards it is bound to the result of the previous iteration, and finally the result of the last iteration is returned.

Note that $\divideontimes$ and $\Box$ have the same meanings as for filter before.

fold also commonly arises when performing comparisons over node sets. In this case, XPath specifies existential semantics: If the specified condition holds for any element of the node set, the entire condition is true. We can e.g. express our earlier predicate `[./author="Franz Kafka"]` as follows:

$$\begin{array}{l}
\mathsf{fold}^{\divideontimes}[\langle n, \Box, o\rangle \Rightarrow o \vee ([n]_{\mathbb{S}}^{\textsc{Node}} \equiv_{\mathbb{S}} \text{“Franz Kafka”})](\textbf{false}) \\
\quad \circ \mathsf{filter}^{\divideontimes}[\langle n, \Box\rangle \Rightarrow \mathsf{name}(n) \equiv_{\mathbb{S}} \text{“author”}] \circ ((\triangleright \rightarrow)^{*} \bullet \swarrow)
\end{array}$$

Formally, we can give the semantics of fold as

$$\mathsf{fold}^{\delta}[\langle n, i, o\rangle \Rightarrow f](v)(S) \quad \triangleq \quad R(v, S, 1)$$

where

$$
\begin{array}{lcll}
R(v_1, \emptyset, c) & \triangleq & v_1 & \\
R(v_1, S, c) & \triangleq & \textbf{let} & e = \mathsf{pick}_\delta(S) \\
 & & & v_2 = [\![[\ n \leftarrowtail e, \\
 & & & \qquad i \leftarrowtail \mathbb{D}(c), \\
 & & & \qquad o \leftarrowtail v_1]]\!] \\
 & & & v_3 = R(v_2, S \setminus \{e\}, c+1) \\
 & & \textbf{in} & v_3
\end{array}
$$

3.9 Semantics of $\otimes$

The $\otimes$ construction allows us to express common traversal. First, observe its trivial definition, which is reminiscent of the categorial product:

$$(f_1 \otimes f_2)(S) \quad \triangleq \quad \langle f_1(S), f_2(S) \rangle$$

Semantically, $\otimes$ seems to add nothing. However, consider $\otimes$ being applied e.g. to two filter constructions: In this case, it will expect one NodeSet as input, and yield the result of applying both filter constructions to the same set. Since this can be implemented by having the bodies of both filters test the same same node in sequence, only one iteration over the node set is needed— we use this intuition as a justification to make this class of common traversals explicit through the $\otimes$ operator. As concrete example, consider two queries, `./book` and `./article`, which we can translate into

$$\begin{array}{r}
\mathsf{filter}^*\,[\langle n, \square \rangle \Rightarrow \mathsf{name}(n) = \text{“book”}] \\
\circ((\triangleright \rightarrow)^* \bullet \swarrow)
\end{array}$$

and

$$\begin{array}{r}
\mathsf{filter}^*\,[\langle n, \square \rangle \Rightarrow \mathsf{name}(n) = \text{“article”}] \\
\circ((\triangleright \rightarrow)^* \bullet \swarrow)
\end{array}$$

We can now merge these two queries into

$$\begin{array}{rr}
 & (\mathsf{filter}^*\,[\langle n, \square \rangle \Rightarrow \mathsf{name}(n) = \text{“book”}] \\
\otimes & \mathsf{filter}^*\,[\langle n, \square \rangle \Rightarrow \mathsf{name}(n) = \text{“article”}]) \\
 & \circ((\triangleright \rightarrow)^* \bullet \swarrow)
\end{array}$$

and deconstruct them via **let**.

3.10 Node-Caches

We use Node caches, type NCache, to represent *cached* nodesets. Generation for such nodesets is enforced to avoid multiple iterations over the same nodeset. Semantically, the only nodeset operations are *cache* and *stream*, which are defined as

$$(\mathit{stream} \circ \mathit{cache})\ s = s$$

and *cachesize*, which is

$$(\mathit{cachesize} \circ \mathit{cache})\ s = \#s$$

In practice, *cachesize* is a very efficient function to compute the cardinality of the nodeset represented by a node cache; of course, its efficiency comes at the price of first representing the nodeset as a cache.

3.11 List-based semantics

While set semantics give a good intuition for the intended meaning of the language, a model which more closely captures implementation issues is provided by list-based semantics. As these are more involved, we will only briefly sketch them here; in general, they are analogous to the set-based semantics but require two additional primitives (which we listed in Figure 2, but did not explain before):

- NUB: Eliminates duplicates in a given node stream while preserving its order
- ORDER$^{\delta}$: Orders a given node stream in document order ($\delta = \rightarrow$) or reverse document order ($\delta = \leftarrow$).

The list semantic description of exponentials, however, deserves some mention here (using list comprehension and $+\!\!+$ for list concatenation):

$$[\![f^*]\!]_l(n) \triangleq [n] +\!\!+ [n''|n' \leftarrow [\![f]\!]_l(n), n'' \leftarrow f^*(n')]$$

An interesting property of these semantics is that in the case of nested exponents, e.g.

$$((\triangleright \leftarrow)^* \bullet \swarrow)^*$$

they yield a depth-first pre-order traversal, which in turn gives us document order for this important case. To see this, observe that every returned list element of the inner exponent ($(\triangleright \leftarrow)^*$) is returned to the outer exponent (into its n'); the outer exponent then immediately recurses on it (in $n'' \leftarrow f^*(n')$).

4 Code Generation from AXIL

Since the semantics of AXIL are pure (in a functional sense), both strict and lazy implementations of the language are conceivable. Our current implementation is strict; it generates nested loop code from a graph representation of AXIL. In this graph representation, multiple uses of the same value are made explicit to allow memoisation. Furthermore, we have eliminated all occurrences of ($\circ$) and ($\bullet$) and, instead, use explicit name bindings, i.e. the AXIL expression $(\triangleright \rightarrow)^* \bullet \swarrow$ becomes (essentially)

$$\lambda n.(\lambda m. \triangleright (\rightarrow (m)))^*(\swarrow (n))$$

In XJ, which provides the context for our current implementation, it is possible for XPath query results to be immediately coerced to a primitive type; in particular, coercions of queries on attribute node sets filtered by name are common. We import all coercions into the actual AXIL representations, which allows us to perform optimisations on them during a later re-writing phase— in this particular example, the re-writing allows

us to reduce the (rather complex) AXIL expression to a single function evaluation.

Other re-writings simplify AXIL terms; for example, our current backend implements the $-^+$ exponent by expanding any expression f^+ to the equivalent of $f^* \triangleleft f$.

Also, fold and filter constructions are not annotated with ordering constraints in our current representation; the reason for this is that they need not sort their inputs. However, they may be annotated with early-termination predicates, which allow us to re-write the previous fold example (3.8) as follows:

$$\mathsf{fold}^{\divideontimes} \left[\begin{array}{lll} \langle n, \square, o \rangle \Rightarrow & & \\ \quad \textbf{yield} & [n]_{\mathbb{S}}^{\textsc{Node}} \equiv_{\mathbb{S}} \text{“Franz Kafka”} & \\ \quad \textbf{until} & [n]_{\mathbb{S}}^{\textsc{Node}} \equiv_{\mathbb{S}} \text{“Franz Kafka”} & \end{array} \right] (\textbf{false}) \\ \circ \mathsf{filter}^{\divideontimes} [\langle n, \square \rangle \Rightarrow \mathsf{name}(n) \equiv_{\mathbb{S}} \text{“author”}] \circ ((\triangleright \rightarrow)^* \bullet \swarrow)$$

These early termination annotations (which we omitted in our previous exposition for brevity, as they are not required for the set-based semantics) are essential for efficiency. In the above, both occurrences of the predicate would share their code and only be evaluated once.

5 Comparison and Results

To evaluate the usefulness of AXIL for immediate code generation, we implemented it as part of the XJ [HRS+05] compiler, together with a prototype backend which evaluates AXIL queries strictly. XJ currently (v1.1) uses the Xalan DOM implementation to represent XML trees; our AXIL implementation reflects this.

To compare our performance, we ran our implementation on three micro-benchmarks and one XJ sample program, and compared results with XJ v1.1, XJ using the XPath engine of Xalan for query evaluation, and a manual implementation of all benchmarks (listed as “Manual DOM”) which we had manually fine-tuned to the specific problem at hand. All versions were run on Sun’s JDK 1.5.0, build 64, running on GNU/Linux on top of a single-CPU 2.4 GHz Intel Pentium 4 machine with 2GB of memory on a document of about 2^{15} nodes. All queries were executed at least 200 times, with the first 100 runs being discarded in an attempt to minimise caching and JITting artifacts.

As shown in Figure 3, our AXIL performance is reasonably close to the performance of our hand-optimised code. Our implementation has to deal with some overhead caused by XJ semantics which we cannot eliminate without more sophisticated program analyses, causing some of the overhead observed in this figure. While more analyses will be required before a final judgement can be made, our prototype system seems to perform well enough to make it practical; in particular, it performs consistently better than the existing XJ v1.1 evaluation system.

Benchmark	Xalan	XJ v1.1	AXIL	Manual DOM
`./country`	9.9s	0.037s	0.024s	0.021s
`descendant-or-self::node()`	27.8s	4.27s	0.68s	0.46s
`./country[@population >= 1000000] [position() > (last() div 2)]`	23.8s	0.36s	0.084s	0.047s
Mondial	>24h	40.7s	5.7s	3.6s

Figure 3: Performance of a prototype AXIL-based compiler backend for XJ

6 Related Work

Translating XPath into other "intermediate" languages for the purpose of optimisation or compilation has previously been done in [Pan04], who translated XPath queries into SQL queries, and by Helmer and Kanne [HKM02], for the Natix XML database. As noted by the latter, pipelining– which corresponds to our deforestation process– is highly important in computing results efficiently (by avoiding intermediate results). Unlike our work, both systems directly apply databases for query processing. Core XQuery [DFF$^+$05] is also an intermediate language, statically typed and with explicit coercions, but captures far more than XPath 1.0, while (as argued previously) not allowing some traversal merges to be expressed. In general, AXIL can be thought of as being at a slightly lower level than core XQuery.

Among non-XPath alternative query languages, XAL [FHP02] also uses a functional scheme with higher-order constructions, specifically also with a Kleene-star "meta-operation", though they rely on a "projection" operation to filter nodes by name and kind, rather than relying on a general-purpose filter operation.

7 Future Work

To validate the usefulness of AXIL, further optimisation steps will need to be implemented on it. For example, we do not currently have a satisfactory scheme for the elimination of sorting and duplicate removal primitives, though we hope to adopt Hidders and Michiels scheme from [HM03].

7.1 Complete Schema types

Some optimisations require more type information than we currently provide. Consider, for example, a document which consists of "book" elements, which in turn contain (exclusively) "author" elements, which have no element children of their own. When asked to search for all books in the document, an overeager programmer might specify the query as `.//book`, asking for a recursive traversal of the entire tree— which would clearly touch far more nodes than necessary. With enough type information, however, we could infer that the query need not descend and can be expressed as the AXIL equivalent of `./book`.

7.2 The preceding Axis

Almost all XPath axes can be represented concisely with the primitives developed previously— the sole exception to this is the ordering of the `preceding` axis, which we represent as

$$\text{ORDER}^{\leftarrow} \circ (\leftarrow^* \bullet \searrow)^* \triangleleft \leftarrow^+ \triangleleft \uparrow^*$$

Here, an explicit ordering operation is required, if the result of the operation needs to be ordered (i.e., if the result or a node set derived from the result is passed into a fold or filter which makes use of its index parameter).

While the $\leftarrow^+ \triangleleft \uparrow^*$ fragment does generate its results in reverse document order, as desired, the fragment $(\leftarrow^* \bullet \searrow)^*$ does not: In order to achieve this, we would need a post-order depth-first traversal, but our semantics for $-^*$ yield a pre-order traversal. We currently use the above, inefficient solution, as the preceding axis is not used very frequently in our experience; to provide a more satisfactory solution, a new exponential primitive would be required.

8 Conclusion

AXIL is an XPath intermediate language suitable for a number of optimisations. Being a pure functional language, it can be compiled both into strict and into lazy code, or into a hybrid of both. We have described the syntax and relevant parts of the AXIL language, given a rationale for its current form, and sketched its current prototype implementation, implemented within the XJ compiler. Initial performance results seem to indicate that XJ is suitable for efficient compilation, though more analyses will need to be performed before a conclusion can be drawn.

References

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[App98] Andrew W. Appel. SSA is Functional Programming. *ACM SIGPLAN Notices*, 33(4):17–??, 1998.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath), Version 1.0. W3C Recommendation, `http://www.w3.org/TR/xpath`, 1999.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[Cla99] James Clark. XSL Transformations (XSLT) Version 1.0. W3C Recommendation, `http://www.w3.org/TR/xslt`, 1999.

[DFF+05] Denise Draper, Peter Fankhauser, Mary Fernández, et al. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft, `http://www.w3.org/TR/xquery-semantics`, 2005.

[FHP02] Flavius Frasincar, Geert-Jan Houben, and Cristian Pau. XAL: An Algebra For XML Query Optimization. In Xiaofang Zhou, editor, *Thirteenth Australasian Database Conference (ADC2002)*, Melbourne, Australia, 2002. ACS.

[HHW+00] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, et al. Document Object Model (DOM) Level 2 Core Specification. W3C Recommendation, `http://www.w3.org/TR/DOM-Level-2-Core`, 2000.

[HKM02] Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Optimized Translation of XPath Expressions into Algebraic Expressions Parameterized by Programs Containing Navigational Primitives. In *Proceedings of the International Conference on Web Information Systems Engineering (WISE)*, 2002.

[HM03] J. Hidders and P. Michiels. Avoiding Unnecessary Ordering Operations in XPath. In *Proceedings of the 9th International Workshop on Database Programming Languages (DBPL)*, Potsdam, Germany, 2003.

[HRS+05] Matthew Harren, Mukund Raghavachari, Oded Shmueli, et al. XJ: Facilitating XML Processing in Java. In *14th International World Wide Web Conference (WWW2005)*. ACM Press, 2005.

[Jon88] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1988.

[MW92] Simon Marlow and Philip Wadler. Deforestation for Higher-Order Functions, 1992.

[Pan04] Tadeusz Pankowski. Processing XPath expressions in relational databases. In P. van Emde Boas, J. Pokorny, M. Bielikova, and J. Stuller, editors, *SOFSEM 2004: Theory and Practice of Computer Science*, pages 265–276. Springer-Verlag, 2004.

[PW99] Philip Wadler. A formal semantics of patterns in XSLT. *Markup Technologies*, 16, 1999.

[Wad99] Philip Wadler. Two semantics for XPath. `http://homepages.inf.ed.ac.uk/wadler/topics/xml.html`, 1999.