Interpreting XPath by Iterative Pattern Matching with Paisley

Baltasar Trancón y Widemann[1,2] and Markus Lepper[2]

[1] Ilmenau University of Technology
[2] <semantics/> GmbH

Abstract. The Paisley architecture is a light-weight EDSL for non-deterministic pattern matching. It automates the querying of arbitrary object-oriented data models in a general-purpose programming language, using API, libraries and simple programming patterns in a portable and non-invasive way. The core of Paisley has been applied to real-world applications. Here we discuss the extension of Paisley by *pattern iteration*, which adds a Kleene algebra of pattern function composition to the unrestricted use of the imperative host language, thus forming a hybrid object-oriented–functional–logic framework. We subject it to a classical practical problem and established benchmarks: the node-set fragment of the XPath language for querying W3C XML document object models.

1 Introduction

The Paisley embedded domain-specific language (EDSL) and library adds the more declarative style of *pattern matching* to the object-oriented context of Java programming [9,10]. Paisley offers a combination of features that is, by our knowledge, not offered by any other existing pattern matching solution for a main-stream language: it is strictly typed, fully compositional, non-invasive and supports non-determinism, full reification and persistence.

The non-deterministic aspects of Paisley have been demonstrated to provide a fairly powerful basis for embedded logic programming in the object-oriented host environment. In [11] we have discussed how to solve cryptarithmetic puzzles with the backtracking facilities of Paisley, and how to obtain non-trivial efficient search plans by object-oriented meta-programming with Paisley pattern objects.

Here we describe and evaluate recent extensions to Paisley that greatly increase its capabilities as an embedded functional-logic programming system, supporting more complex control and data flow than the combinatorial generate&test tactics of [11]. In particular, the novel contributions are:

1. in the core layer, a calculus of functions on patterns and its Kleene algebra, thus providing regular expressions of nested patterns;
2. in the application layer, a library extension that demonstrates their use by matching W3C XML Document Object Models with XPath [3] expressions;
3. a practical case study that evaluates the approach in comparison with standard on-board facilities of the host language Java.

Variants: **1** `.//a[@href]` **2** `.//p//a[@href]` **3** `.//p[.//a[@href]]`

```
void collect (int variant, Node node, Collection⟨Element⟩ results,
              boolean sideways) {
  if (node instanceof Element) {
    Element elem = (Element)node;
    if (elem.getTagName().equals(variant == 1 ? "a" : "p")) {
      if (variant > 1) {
        Collection⟨Element⟩ sub = variant == 2 ? results : new ArrayList⟨⟩();
        collect(1, elem, sub, false);
      }
      if (variant == 1 && !elem.getAttribute("href").equals("") ||
          variant == 3 && !sub.isEmpty())
        results.add(elem);                                    // solution
    }
  }
  if (node.getFirstChild() != null)
    collect(variant, node.getFirstChild(), results, true);    // depth
  if (sideways && node.getNextSibling() != null)
    collect(variant, node.getNextSibling(), results, true);   // breadth
}
```

Fig. 1. Three related XML queries. *Top:* XPath expressions; *bottom:* Java code.

2 Motivation

The basic motivation for a pattern matching EDSL is that patterns as language constructs make data-gathering code more declarative, and thus more readable, more writable and more maintainable. If done properly, this additional expressive power can be used to factor code compositionally in ways that are not straightforwardly possible in a conventional object-oriented approach, where the logic and data flow of a complex query often appear as cross-cutting concerns.

As an example that highlights the issues of logical compositionality, consider a family of three related XPath expressions, depicted in Fig. 1, that select nodes from XHTML documents. The first selects, from the context node and its descendants, all `<a>` elements (hyperlink anchors) that have a `href` (target) attribute specified. The second selects only those `<a>` elements with `href` attributes that are nested within `<p>` elements (paragraphs). The third selects all `<p>` elements that have some `<a>` element with `href` attribute nested within.

The task is to implement the specified search procedures, as object-oriented queries of the standard W3C X(HT)ML Document Object Model. Evidently, one wishes to implement most of the operational code generically, with reuse and minimal adaptation for each of the three variants. Additionally, the second and third variant should be able to use the first recursively. To emphasize the role of reuse in this example, Fig. 1 gives a unified implementation, where the identifier variant is grayed out to emphasize that all its occurrences serve only the static choice between the variants. The common algorithm is to traverse nodes

by depth-first search, and add all valid matches, possibly repeatedly, to a results collection supplied by the caller. Two further improvements of the solution are suggested as exercises to the reader:

1. Of course, code that works for these three, but no other XPath expressions is hardly generic. Abstract to a large, preferrably unbounded, set of useful XPath expressions. Refactor the code to separate commonalities and individual degrees of freedom, as cleanly as possible.
2. The use of a Collection for matches implies eager evaluation. This is not efficient for the recursive calls from variant 3 to variant 1, where searching can be aborted after the first successful match (which falsifies sub.isEmpty()). Refactor the code to allow for lazy evaluation, as transparently as possible.[3]

See section 5 and Fig. 4 below for our proposed solution. We do not claim that these refactoring steps are infeasible in any particular previous framework. But we shall demonstrate that the Paisley approach naturally gives both pragmatic design guidelines, and a concrete notation for the adequate, abstract, modular and efficient expression of the underlying algorithm and its instantiations.

3 Basic Matching with Paisley

The design principles, semantics and APIs of Paisley patterns have been discussed in detail in [9,10,11]. The EDSL is extremely lightweight: It requires no language or compiler extension; API calls and programming idioms are sufficient for its full expressive power. The core API is summarized in Fig. 2.

The root of the pattern hierarchy is the abstract base class Pattern⟨A⟩ of patterns that can process objects of some arbitrary type A. A pattern Pattern⟨A⟩ p is applied to some target data x of type A by calling p.match(x), returning a **boolean** value indicating whether the match was successful.

Complex patterns are composed from application-layer building blocks that implement classifications and projections, the reification of instance tests and getter methods, respectively. These can be defined independently for arbitrary (public interfaces of) data models, requiring no intrusion into legacy code. Paisley comes with predefined bindings for common Java data models, such as the Collection framework; more can be added by the user. Each projection x from type A *to* B induces by contravariant lifting a construction of type Pattern⟨A⟩ *from* Pattern⟨B⟩, conveniently implemented as a method Pattern⟨A⟩ getX(Pattern⟨B⟩ p).

Information is extracted from a successfully matched pattern via embedded pattern variables. A pattern Variable⟨A⟩ v matches any value A x, and stores a reference to x that can be retrieved by invoking v.getValue(). Variables behave like imperative rather than logical variables; subsequent matches merely overwrite the stored value, no unification is implied.[4]

[3] This task is inherently much more difficult in conventional programming languages than in logic and lazy functional approaches; cf. [5].

[4] This design choice enables the use of Paisley for general, object-oriented data models, where stable notions of equality, let alone an induction principle, cannot be taken for granted. See section 4 for the handy implications of the imperative perspective.

```
abstract class Pattern⟨A⟩ {
  public abstract boolean match(A target);
  public boolean matchAgain();
  public static ⟨A⟩ Pattern⟨A⟩
    both(Pattern⟨? super A⟩ first, Pattern⟨? super A⟩ second);
  public static ⟨A⟩ Pattern⟨A⟩
    either(Pattern⟨? super A⟩ first, Pattern⟨? super A⟩ second);
  public Pattern⟨A⟩ someMatch();
}

class Variable⟨A⟩ extends Pattern⟨A⟩ {
  public A getValue();
  public ⟨B⟩ List⟨A⟩ eagerBindings(Pattern⟨? super B⟩ root, B target);
  public ⟨B⟩ Iterable⟨A⟩ lazyBindings(Pattern⟨? super B⟩ root, B target);
  public ⟨B⟩ Pattern⟨B⟩ bind(Pattern⟨? super B⟩ root, Pattern⟨? super A⟩ sub);
  public Pattern⟨A⟩ star(Pattern⟨? super A⟩ root);
  public Pattern⟨A⟩ plus(Pattern⟨? super A⟩ root);
}
```

Fig. 2. Interface synopsis (core).

Patterns can be composed conjunctively and disjunctively by the binary combinators both and either, or their n-ary variants all and some (not shown), respectively. As in traditional logic programming, disjunction is realized as backtracking: After each successful match for a pattern p, p.matchAgain() can be invoked to backtrack and find another solution. Solutions can be exhausted, in an imperative style of encapsulated search, by the following programming idiom:

```
if (p.match(x)) do
  doSomething();
while (p.matchAgain());
```

When *not* using an exhaustive loop, the computation of alternative solutions can be deferred indefinitely; the required state for choice points is stored residually in the instances of the logical combinators themselves.

The search construct can be abbreviated further to a functional style if the desired result of each match is the value stored in a single pattern variable v. Invoking v.eagerBindings(p, x) or v.lazyBindings(p, x) returns objects that give the value of v for all successive matches, collected beforehands in an implicit loop, or computed on iterator-driven demand, respectively. The latter can also deal with infinite sequences of solutions.

For cases where only the satisfiability of a pattern p, but not the actual sequence of solutions, is of concern, one can invoke p.someMatch(), yielding a wrapper that cuts all alternatives after the first solution. Thus, laziness is exploited for efficiency and, when used in conjunction with other patterns with multiple relevant solutions, spurious multiplication of solution sets is avoided.

4 Advanced Pattern Calculus

4.1 Pattern Substitution and Iteration

As discussed in [10], Paisley patterns and their variables obey an algebraic calculus where most of the expected mathematical laws hold, despite the low-level imperative nature of their implementation.

A variable v known to occur[5] in a pattern p can be substituted by a subpattern q, by invoking v.bind(p, q), mnemonically read as $(\lambda v.\, p)q$. The implementation delegates to p and q sequentially, and hence does not require access to the internals of either operand.

Substitution can also be given a recursive twist; a pattern p with a "hole" v can be nested. The patterns v.star(p) and v.plus(p) correspond to the * and $^+$-closures, respectively, of the search path relation between p and v, in the sense that the usual recursive relations hold,

$$\begin{aligned} \mathsf{v.star(p)} &\equiv \mathsf{either(v,\ v.plus(p))} \\ \mathsf{v.plus(p)} &\equiv \mathsf{v.bind(p,\ v.star(p))} \end{aligned}$$

which is already almost the complete, effective implementation, up to lazy duplication of v.plus(p) for each iteration level that is actually reached. Note that iteration depth is conceptually unbounded, and solutions are explored in depth-first pre-order, due to the way either is used. Thus patterns form a Kleene algebra up to equivalence of solution *sets* but not *sequences*.

Using these iteration operators, complex nondeterministic pattern constructors can be defined concisely. For instance, the contravariant lifting of the multi-valued, transitive *descendant* projection in XML document trees, applied to a pattern p, becomes simply

```
v.bind(v.plus(child(v)), p)
```

where Variable⟨Node⟩ v is fresh, and the multi-valued projection Pattern⟨Node⟩ child(Pattern⟨Node⟩) is implemented in terms of the org.w3c.dom.Node getter methods getFirstChild() and getNextSibling(). Note how the two sources of nondeterminism, regarding horizontal (child) and vertical (plus) position in the document tree, respectively (cf. the distinct recursive calls in Fig. 1), combine completely transparently.

4.2 Pattern Function Abstraction

Functions taking patterns to patterns have so far featured in two important roles. In operational form, implemented as lifting methods, they are the basis for contravariant representation of projection patterns. In algebraic form, as bind-based abstractions from a variable, they enable pattern composition and iteration. Being so ubiquitous in the Paisley approach, they deserve their own

[5] The actual condition is definite assignment rather than occurrence, for technical reasons.

```
abstract class Motif⟨A, B⟩ {
  public abstract Pattern⟨B⟩ apply(Pattern⟨? super A⟩);
  public static  ⟨A⟩ Motif⟨A, A⟩ id();
  public         ⟨C⟩ Motif⟨C, B⟩ on(Motif⟨C, ? super A⟩ other);
  public static  ⟨A⟩ Motif⟨A, A⟩ star(Motif⟨A, ? super A⟩ f);
  public static  ⟨A⟩ Motif⟨A, A⟩ plus(Motif⟨A, ? super A⟩ f);
  public Iterable⟨A⟩ lazyBindings (B target);
  public List⟨A⟩ eagerBindings(B target);
}

class Variable⟨A⟩ extends Pattern⟨A⟩ {
  // ...
  public ⟨B⟩ Motif⟨A, B⟩ lambda(Pattern⟨B⟩ root);
}
```

Fig. 3. First-class pattern function (motif) interface.

reification. Fig. 3 shows the relevant API. An instance of class Motif⟨A, B⟩ is the reified form of a function taking patterns over A to patterns over B, or by contravariance, the pattern representation of an access operation that takes data of type B to data of type A, by its apply method.

The Motif class provides the algebra of the category of patterns, namely the identical motif id() and the type-safe composition of motifs by the on(Motif) instance method. The variable-related operations star/plus and lazy-/eagerBindings each have a point-free counterpart in Motif, which create anonymous variables to hold intermediate results on the fly. For instance, the XML *descendant* pattern defined above can be expressed less redundantly in point-free form as

```
plus(child).apply(p)
```

given a child access reification Motif⟨Node,Node⟩ child.

Conversely, motif abstraction is defined for pattern variables, which obeys the beta reduction rule:

$$\mathsf{v.lambda(p).apply(q)} \equiv \mathsf{v.bind(p,\ q)}$$

There is often a trade-off in convenience between functions in operational and reified form, that is as either pattern lifting methods or motifs. Since there is no automatic conversion between methods and function objects in Java prior to version 8, our strategy in the Paisley library is to provide both redundantly.

5 Motivation Revisited

Figure 4 repeats the XPath examples expressions, now contrasting the domain-specific XPath code with the general-purpose Java/Paisley code. The latter has been underlined for easy reference. The solution makes full use of the shared

Variants: **1** `.//a[@href]` **2** `.//p//a[@href]` **3** `.//p[.//a[@href]]`

```
Pattern⟨Node⟩ dslash(String name, Pattern⟨Element⟩... constraints) {
  return descendantOrSelf(element(both(tagName(name), all(constraints))));
}

Variable⟨Element⟩ outerVar = new Variable⟨⟩();
Pattern⟨Node⟩ outerForm = dslash("p", outerVar);
Variable⟨Element⟩ innerVar = new Variable⟨⟩();
Pattern⟨Node⟩ innerForm = dslash("a", innerVar, attr("href", neq("")));

Iterable⟨Element⟩ collect1(Document root) {
  return innerVar.lazyBindings(innerForm, root);
}
Iterable⟨Element⟩ collect2(Document root) {
  return innerVar.lazyBindings(outerVar.bind(outerForm, innerForm), root);
}
Iterable⟨Element⟩ collect3(Document root) {
  return outerVar.lazyBindings(outerVar.bind(outerForm, innerForm.someMatch()), root);
}
```

Fig. 4. Three related XML queries revisited (from Figure 1). *Top:* XPath expressions; *middle:* Java code template; *bottom:* template instantiations.

common structure of the three variants, and achieves complete separation of concerns:

- The search procedure implied by the operator `//` is not spread out as recursive control flow, but reified and encapsulated as the descendantOrSelf pattern lifting, predefined in the XPathPatterns static factory class of Paisley, discussed in detail in the following section.
- XPath subexpressions are denoted concisely and orthogonally.
  - The generic form `.//`*tag*... that reoccurs in all variants is abstracted as the extensible pattern construction dslash.
  - The particular inner and outer forms, `.//a[@href]` and `.//p`, respectively, are defined once and for all, independently of each other.
- The semantics of XPath queries are effected concisely and orthogonally.
  - The general principle of encapsulated search is expressed naturally by a call to lazyBindings. The third variant concisely prunes the recursively encapsulated search *after the first solution*, via the pattern modifier someMatch().
  - The particular configuration of subexpressions for all variants boils down to a simple choice of the bound variable and pattern-algebraic composition. Note that both are reified and hence first-class citizens of Java. Thus, each variant boils down to a one-line expression, where the points of variability are ordinary subexpressions that can be chosen statically (as shown) or dynamically, with arbitrarily complex meta-programming.

```
enum Axis {
  public Motif⟨? extends Node, Node⟩ getMotif();
}

abstract class Test {
  public abstract Motif⟨? extends Node, Node⟩ getMotif();
}

abstract class Predicate {
  public abstract boolean accepts(NodeSet context, int position, Node candidate);
  public Motif⟨? extends Node, Node⟩ apply(Motif⟨? extends Node, Node⟩ base);
}

class NodeSet {
  private Collection⟨? extends Node⟩ elems;
  public NodeSet(Iterable⟨? extends Node⟩ elems);
  public int size();
  public Iterable⟨Node⟩ filter(Predicate pred);
}

class Path {
  Path(Motif⟨? extends Node, Node⟩ motif);
  public Motif⟨? extends Node, Node⟩ getMotif();
  public Path step(Axis axis, Test test,  Predicate... predicates);
}
```

Fig. 5. XPath base classes in Paisley.

6 The Paisley XPath Interpreter

As a case study of real-world data models and queries, we have implemented the *navigational* (proper path) fragment of the XPath 1.0 language as a Paisley pattern combinator library. The complete implementation consists of the factory classes XMLPatterns for generic DOM access and XPathPatterns for XPath specifics, with currently 433 and 282 lines of code, respectively. Given an XPath parser, it can be extended to a full-fledged interpreter, and hence a non-embedded DSL, by a straightforward mapping of abstract syntax nodes to pattern operations.

6.1 Language Fragment

The XPath 1.0 language comes with many datatypes and functions that do not contribute directly to its main goal, namely the addressing of nodes in an XML document. For simplicity, we restrict our treatment of the language to a fragment streamlined for that purpose. Typical uses of XPath within XSLT or XQuery [2] conform to this fragment. We conjecture that the missing features can be added without worsening essential operational complexity and runtime performance.

We omit external variables and functions, and all operations on the datatypes of strings, numbers and booleans, as well as intangible document nodes such as comments and processing instructions. The supported sublanguage is reflected one-to-one by API operations based in the class hierarchy depicted in Fig. 5. (Operation signatures are given in Fig. 11 in the appendix.) Its focus is on so-called path expressions of the general syntactic form

$$\mathit{path} ::= \mathit{abs\_path} \mid \mathit{rel\_path} \qquad \mathit{abs\_path} ::= \texttt{/}\,\mathit{rel\_path}^{?}$$
$$\mathit{step} ::= \mathit{axis}\ \texttt{::}\ \mathit{test}\ (\texttt{[}\ \mathit{predicate}\ \texttt{]})^{*} \qquad \mathit{rel\_path} ::= (\mathit{rel\_path}\ \texttt{/}\ )^{?}\,\mathit{step}$$

Here *axis* is one of the twelve XPath document axes, omitting the deprecated namespace declaration axis. The nonterminal *test* is the tautological test `node()`, the text node test `text()` or an explicit node name test. For *predicate*, used to filter selected nodes, we accept not the full XPath expression language, but only

$$\mathit{predicate} ::= \mathit{path} \mid \mathit{integer} \mid \texttt{not}\ \mathit{predicate} \mid \texttt{(}\ \mathit{predicate}\ (\texttt{and} \mid \texttt{or})\ \mathit{predicate}\ \texttt{)}$$

where a *path* predicate holds if and only if it selects at least one node (existential quantification). Positive and nonpositive *integer* predicates `[`$\pm i$`]` select the i-th node from the candidate sequence, and the $(n - i)$-th node, respectively, from the sequence of n candidates in total. The former is a valid abbreviation for `[position()==i]` in standard XPath; we add the latter as an analogous abbreviation for `[position()==last()-i]`. Logical connectives on predicates are defined as usual.

Various abbreviation rules apply; for instance, the shorthand `.//a[@href]` expands to the verbose form `self::node()/descendantOrSelf::node()/child::a[attribute::href]`. This translates to the following semantic object:

```
relative().step(Axis.self, node())
          .step(Axis.descendantOrSelf, node())
          .step(Axis.child, name("a"),
               exists(relative().step(Axis.attribute, "href")))
```

The relation between XPath predicates and candidate sequences, misleadingly called "node-sets" in the standard, is rather idiosyncratic (they can not be modeled adequately as plain sets) and the major challenge in this case study. A node-set is implicitly endowed with either of two predefined orders, namely *forward* or *reverse document order*. These orders are loosely specified by a pre-order traversal of nodes, up to attribute permutation. A predicate filters the nodes in a node-set not purely by point-wise application, but may depend on some context, namely the position of the node in the node-set, starting from one, and the total number of members. This information is available in XPath via the "functions" `position()` and `last()`, respectively. It is realized in the API by the parameter position of method Predicate.accepts and the method size() of class NodeSet, respectively, to be supplied from the method filter of class NodeSet.

6.2 Pattern-Based Interpreter Design

The node-extracting semantics of the XPath language can be rendered naturally in the Paisley framework by contravariant lifting. An XPath expression can be

$$\begin{aligned}
\mathsf{ancestor} &\equiv \mathsf{plus}(\mathsf{parent}) & \mathsf{descendant} &\equiv \mathsf{plus}(\mathsf{child}) \\
\mathsf{ancestorOrSelf} &\equiv \mathsf{star}(\mathsf{parent}) & \mathsf{descendantOrSelf} &\equiv \mathsf{star}(\mathsf{child}) \\
\mathsf{followingSibling} &\equiv \mathsf{plus}(\mathsf{nextSibling}) & \mathsf{precedingSibling} &\equiv \mathsf{plus}(\mathsf{previousSibling}) \\
\mathsf{following} &\equiv \mathsf{ancestorOrSelf.on}(\mathsf{followingSibling}).\mathsf{on}(\mathsf{descendantOrSelf}) \\
\mathsf{preceding} &\equiv \mathsf{ancestorOrSelf.on}(\mathsf{precedingSibling}).\mathsf{on}(\mathsf{descendantOrSelf}) \\
\mathsf{self} &\equiv \mathsf{id}
\end{aligned}$$

Fig. 6. Non-primitive XPath axes.

```
class Path {
  // ...
  public Path step(Axis axis, Test test,  Predicate... predicates) {
    Motif⟨? extends Node, Node⟩ r = axis.getMotif().on(test.getMotif());
    for (Predicate p : predicates)
      r = p.apply(r);
    return new Path(getMotif().on(r));
  }
}
```

Fig. 7. Implementation of composite path expressions.

applied to any node of an XML document, here implemented as DOM, and extracts some other nodes, possibly of a more special type, such as elements or attributes. This gives rise to a semantic type Motif⟨? **extends** Node, Node⟩.

Except for the context-sensitivity of predicates, all language constructs could simply been given motif semantics and lumped together by composition.

XPath axes are all defined concisely in terms of motifs. Primitive DOM access operations from factory class XMLPatterns directly define the attribute, child and parent axes. Given the additional primitives next-/ previousSibling, which are only implicit in the XPath standard, all other axes are definable in terms of elementary motif algebra; see Figure 6. Node tests are straightforward applications of test pattern lifting.

Atomic path expressions are realized by the document root access motif and the identity motif, for absolute and relative paths, respectively. Composite path expressions conceptually compose their constituents left to right. The exception are predicates, which are implemented as motif transforms in order to deal with context sensitivity. These are applied in order to the step basis, that is the composition of axis and test, for local filtering, and the result is then composed with the front of the path expression; see Fig. 7.

The real challenge is the implementation of node-set predicates: On the one hand, context information about relative element position and total node-set size must be provided, which transcends the context-free realm of pattern and motif composition. On the other hand, elements should be enumerated lazily, in

```
class Predicate {
  // ...
  public Motif⟨Node, Node⟩ apply(final Motif⟨? extends Node, Node⟩ base) {
    return new Motif⟨? extends Node, Node⟩() {
      public Pattern⟨Node⟩ apply(Pattern⟨? super Node⟩ p) {
        return new MultiTransform⟨Node, Node⟩(p) {
          protected Iterable⟨Node⟩ apply(Node n) {
            return new NodeSet(base.lazyBindings(n)).filter(Predicate.this); // [*]
  }}; }}; }
}

class NodeSet {
  // ...
  public NodeSet(Iterable⟨? extends Node⟩ elems) {
    this.elems = cache(elems) ;
  }
  public Iterable⟨Node⟩ filter(final Predicate pred) {
    return new Iterable⟨Node⟩() {
      public Iterator⟨Node⟩ iterator() {
        return new FilterIterator⟨Node⟩(elems.iterator()) {
          int i = 0;                                                    // [*]
          protected boolean accepts(Node candidate) {
            return pred.accepts(NodeSet.this, ++i, candidate);          // [*]
  }}; }}; }
}
public static Predicate exists(final Path cond) {
  return new Predicate() {
    public boolean accepts(NodeSet context, int position, Node node) {
      return cond.getMotif().apply(any()).match(node);                  // [*]
    }
  };
}
public static Predicate index(final int i) {
  return new Predicate() {
    public boolean accepts(NodeSet context, int position, Node node) {
      return position == (i > 0 ? i : context.size() − i);              // [*]
    }
  };
}
```

Fig. 8. Implementation of context-sensitive predicate filtering. See text for underlining and [∗].

order to make tests for non-emptiness efficient and avoid scanning for unneeded solutions. Obviously, evaluation must switch transparently from lazy to eager strategy if the size of the node-set is observed. And lastly, for elegance reasons, we strive for an implementation that is as declarative as possible, with very limited amounts of specific imperative coding.

The solution is depicted in Figure 8. Generic Paisley API method calls are underlined for easy reference. The action of a predicate on a base motif requires control over the solution node-set. Hence it needs to intercept both the pattern parameter at the motif level and the target node parameter at the pattern level, by means of two nested anonymous classes. Then a lazy disjunction is spliced in, which enumerates the solutions of the base motif, wraps them in a local node-set and filters them context-sensitively.

The counterpart on the node-set side of the implementation works as follows: It wraps the lazy enumeration of candidate nodes is a collection, via the auxiliary method cache (definition not shown). This collection caches enumerated items for repeated access, and forces eager evaluation if its size() method is called.

The actual filtering operation yields a lazy enumeration that intercepts iterator creation, again by means of two nested anonymous classes. The iterator of the cached candidate collection is overwritten by an instance of the auxiliary abstract class FilterIterator⟨A⟩ (definition not shown) that in-/excludes elements ad-hoc, determined by the result of its method **boolean** accepts(A). This acceptance test is then routed back to the context-sensitive acceptance test of the given predicate, by addition of a single minuscule piece of explicit imperative (stateful) programming, namely a counter i for the relative position. Note that, Java formal noise apart, the actual problem-specific code consists of five single-line statements, marked with [∗].

Existential predicates forget their results, hence the invocation of the catch-all pattern any(). They prune the search after the first hit, as witnessed by the absence of a call to matchAgain() on the freshly created pattern. Eager evaluation is only ever triggered by index predicates via a call to context.size(). Logical predicate connectives are defined point-wise (not shown).

7 Experimental Evaluation

We have tested the performance and scalability of our implementation using material from XMark [8], a benchmark suite for XML-based large databases. The homepage of XMark [7] offers a downloadable tool to generate pseudo-random well-typed XML files according to a published DTD, with a linear size parameter. We have used the tools to produce test data files for size parameter values from 0.04 to 0.40 in increments of 0.04, where 0.01 corresponds roughly to 1 MiB of canonical XML. From the 20 published XQuery benchmark queries we have chosen as test cases the four where the entire logic is expressed in XPath rather than XQuery operations; see Fig. 9 (and Table 1 in the appendix).

For each pair of data file and query expression, we processed the document with lazyBindings of the XPath motif, and computed running times for extracting

Query	XPath Expression
Q01	`/site/open_auctions/open_auction/bidder[1]/increase/text()`
Q06	`//site/regions//item`
Q15	`/site/closed_auctions/closed_auction/annotation/description/parlist/listitem/parlist/listitem/text/emph/keyword/text()`
Q16	`/site/closed_auctions/closed_auction[annotation/description/parlist/listitem/parlist/listitem/text/emph/keyword/text()]`

Fig. 9. XPath expression test cases from the XMark benchmark.

the *first* solution and *all* solutions (in a loop), as well as the *effective* time (total time divided by number of solutions).

Timing values were obtained as real time with System.nanoTime(), median value of ten repetitions, with interspersed calls to System.gc(). In order to compensate deferred computation costs in the DOM implementation, we reused the same document instances for all successive queries. All experiments were performed on an Intel Core i5-3317U quad-core running at 1.70 GHz, with 8 GiB of physical memory, under Ubuntu 14.04 LTS (64bit), and Oracle Java SE 1.8.0_05-b13 with HotSpot 64bit Server VM 25.5-b02 and 800 MiB heap limit.

Our first quantitative goal, beyond highlighting the elegance and effectiveness of Paisley-style pattern *specification*, is to demonstrate the efficiency payoff of lazy pattern *execution*. Our second quantitative goal comes back to methodological arguments from the motivation section of this paper. Pattern matching has been hailed as a declarative tool that brings expressiveness for data querying extremely close to the hosting programming language environment. Tools for non-embedded domain-specific languages may be more powerful in many respects, but the burden of proof is on them that this power is worth the trouble incurred by the impedance mismatch with ordinary host code. Nevertheless, we should verify that the benefits of tight embedding produced by our methodological approach are not squandered by the implementation.

To that end, we repeated our experiments with the next-closest tool at hand, namely the Java on-board XPath implementation accessible as javax.xml.xpath.∗.[6] We compared both preparation and running times of Paisley XPath patterns with XPathExpression objects obtained via XPath.compile(String). The first solution and all solutions were obtained by calling XPathExpression.evaluate with the type parameter values NODE and NODESET, respectively.

The Paisley approach fares well, even as a non-embedded DSL. The Paisley preparation process (a simple recursive descent parser for full XPath generated with ANTLR[7], followed by direct translation of abstract syntax nodes to pattern operations) is consistently faster than on-board compilation to XPathExpression objects, although the task may have been sped up marginally by considering

[6] In our Java environment, com.sun.org.apache.xpath.internal.jaxp.XPathImpl.
[7] http://www.antlr.org/

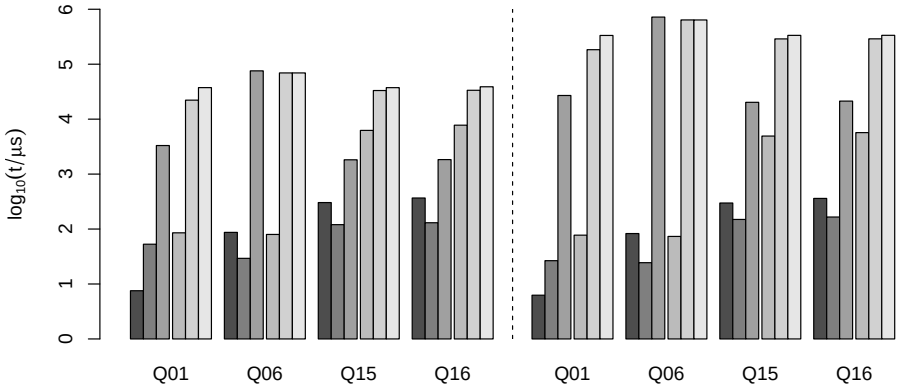

Fig. 10. Running times for Paisley and Java on-board XPath implementations. *Left* – size parameter 0.04; *right* – size parameter 0.40. Colors: *dark to light* – Paisley eff/-first/all; on-board eff/first/all. Logarithmic scale; lower end of scale arbitrary.

only a subset of the language after parsing. (See Table 2 in the appendix for details.) When patterns are constructed in embedded DSL style, using the API rather than textual input, static safety is improved, and even the small overhead eliminated, at the same time.

The differences in running times are so drastic that they can only be visualized meaningfully in logarithmic scale. Proportions range from on-board facilities being 13 % faster for all solutions of Q06 at size 0.04, to being over 26 000 times slower for the first solution of Q06 at size 0.40. With the exception of the exhaustion of "brute-force" case Q06, Paisley is 1–2 (all solutions) or 2–4 (first solution) orders of magnitude faster, respectively; see Fig. 10. Accessing only the first solution with the on-board tools is particularly disappointing, being only marginally faster than exhausting all solutions. It appears that our motivation is confirmed, and conventionally developed tools are ill-suited to scalable lazy evaluation, where external demand governs internal control. (More detailed results are given in Fig. 12 the appendix.)

8 Conclusion

The experimental figures for the on-board tools have been obtained without any tweaking of features; hence there is large uncertainty in the amount of possible improvements. Therefore the comparison should not be taken too literally. But we feel that it is fair in a certain sense nevertheless: Our Paisley implementation has been obtained in the straightest possible manner, also without any tweaking. Its advantage lies thus chiefly in the fact that we have chosen the application domain, namely the specified XPath fragment, and tailored the tool design to avoid any complexity inessential to the task at hand. It is this light-weight flexibility by effective manual programming we wish to leverage with the Paisley

approach – the capabilities of existing, more heavy-weight tools with respect to automated, adaptive specialization are generally no match.

8.1 Related Work

Related work with regard to language design and implementation, and to object-oriented-logic programming, has been discussed in [9,10] and [11], respectively.

Purely declarative accounts of XPath, although theoretically interesting, have little technical impact on our main concern, the concrete embedding in a mainstream programming language. A few random examples: In [6], XPathLog is presented, adding variable binding capabilities and sound Herbrandt semantics to XPath, for a full-fledged logic programming language. In [4], an algorithmic analysis of XPath in terms of modal logic is given. The language fragment and experimental approach used there is a model predecessor for our own work, including the particular benchmark. In the recent paper [1], the implementation of XPath in the Haskell-like functional-logic programming language TOY is discussed. A combinatorial approach to XPath constructs very similar to Paisley is taken, which appears to corroborate our claims of a natural design.

References

1. Almendros-Jiménez, J., Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: XPath query processing in a functional-logic language. ENTCS 282, 19–34 (2012)
2. Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML Query Language (Second Edition). W3C, http://www.w3.org/TR/2010/REC-xquery-20101214/ (2010)
3. Clark, J., DeRose, S.: XML Path Language (XPath) Version 1.0. W3C, http://www.w3.org/TR/1999/REC-xpath-19991116/ (1999)
4. Franceschet, M., Zimuel, E.: Modal logic and navigational XPath: an experimental comparison. In: Proc. Workshop Methods for Modalities. pp. 156–172 (2005)
5. Hughes, J.: Why Functional Programming Matters. Computer Journal 32(2), 98–107 (1989)
6. May, W.: XPath-Logic and XPathLog: A logic-based approach for declarative XML data manipulation. Tech. Rep. 149, Institut für Informatik, Albert-Ludwigs-Universität Freiburg (2001)
7. Schmidt, A.: XMark – an XML benchmark project (2009), http://www.xml-benchmark.org/
8. Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: A benchmark for XML data management. In: Proc. 28th VLDB. pp. 974–985. Morgan Kaufmann (2002)
9. Trancón y Widemann, B., Lepper, M.: Paisley: pattern matching à la carte. In: Proc. 5th ICMT. LNCS, vol. 7307, pp. 240–247. Springer (2012)
10. Trancón y Widemann, B., Lepper, M.: Paisley: A pattern matching library for arbitrary object models. In: Proc. 6th ATPS. LNI, vol. 215, pp. 171–186. Gesellschaft für Informatik (2013)
11. Trancón y Widemann, B., Lepper, M.: Some experiments on light-weight object-functional-logic programming in Java with Paisley. In: Declarative Programming and Knowledge Management. LNCS, vol. 8439. Springer (2014), in press

A Appendix: Supplementary Material

Table 1. Size of input files and solution spaces for XPath expression test cases from the XMark benchmark.

	Size		# Solutions			
Parameter	**File** (kiB)	**# Nodes**	**Q01**	**Q06**	**Q15**	**Q16**
0.04	4 648	191 083	439	870	6	5
0.08	9 212	379 719	884	1 740	17	15
0.12	13 696	569 434	1 301	2 604	29	28
0.16	18 100	748 766	1 732	3 480	32	29
0.20	22 964	946 554	2 142	4 350	37	34
0.24	27 396	1 129 553	2 610	5 214	44	40
0.28	31 976	1 315 902	3 045	6 090	62	58
0.32	36 616	1 502 611	3 453	6 960	60	53
0.36	41 076	1 690 597	3 857	7 824	61	54
0.40	45 600	1 877 979	4 310	8 700	68	59

Table 2. Preparation times for Paisley and Java on-board XPath implementations.

		Q01	**Q06**	**Q15**	**Q16**
Paisley	(μs)	56.40	101.73	89.26	154.32
On-Board		233.44	204.73	162.61	215.92
Ratio		4.14	2.01	1.82	1.40

```
enum Axis {
  ancestor, ancestorOrSelf, attribute,  child,  descendant, descendantOrSelf,
  following,  followingSibling,  parent,  preceding, precedingSibling,  self
  // ...
}

public static Test node  ();
public static Test text  ();
public static Test name  (String tagName);

public static Predicate exists  (Path cond);
public static Predicate index  (int i);

public static Predicate not  (Predicate p);
public static Predicate and  (Predicate p, Predicate q);
public static Predicate or  (Predicate p, Predicate q);

public static Path absolute  ();
public static Path relative  ();
```

Fig. 11. XPath operations as combinators in the Paisley application-layer library.

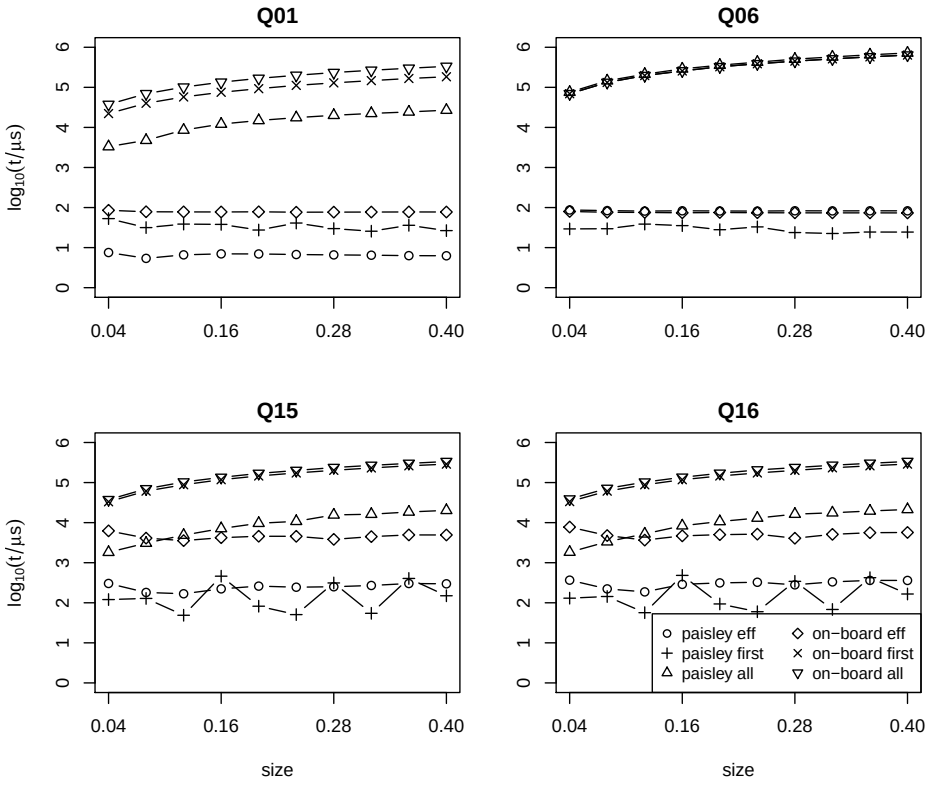

Fig. 12. Running times for Paisley and Java on-board XPath implementations. Logarithmic scale; lower end of scale arbitrary.