Paisley: A Pattern Matching Library for Arbitrary Object Models

Baltasar Trancón y Widemann
Technische Universität Ilmenau
baltasar.trancon@tu-ilmenau.de

Markus Lepper
`<semantics/>` GmbH

Abstract: Professional development of software dealing with structured models requires more systematic approach and semantic foundation than standard practice in general-purpose programming languages affords. One remedy is to integrate techniques from other programming paradigms, as seamless as possible and without forcing programmers to leave their comfort zone. Here we present a tool for the implementation of *pattern matching* as fundamental means of automated data extraction from models of arbitrary shape and complexity in a general-purpose programming language. The interface is simple but, thanks to elaborate and rigorous design, is also light-weight, portable, non-invasive, type-safe, modular and extensible. It is compatible with object-oriented data abstraction and has full support for nondeterminism by backtracking. The tool comes as a library consisting of two levels: elementary pattern algebra (generic, highly reusable) and pattern bindings for particular data models (specific, fairly reusable, user-definable). Applications use the library code in a small number of idiomatic ways, making pattern-matching code declarative in style, easily writable, readable and maintainable. Library and idiom together form a tightly embedded domain-specific language; no extension of the host language is required. The current implementation is in Java, but assumes only standard object-oriented features, and can hence be ported to other mainstream languages.

1 Introduction

Declarative (functional or logical) languages are more or less equally powerful both when creating compound data, and when extracting their components: Term/pattern constructors on the right/left hand side of definition equations, respectively, offer balanced syntactic support with clean, inverse algebraic semantics. Object-oriented languages, by contrast, mostly offer the desirable feature of *compositionality* only when creating objects, but lack a corresponding primitive idiom for extraction. Instead, explicit getter methods, type casts, assignments to local variables and explicit case distinction have to be applied in an imperative programming style, which is inadequate to the purpose of mere querying.

Obviously it is desirable to enrich object-oriented programming practice by techniques from more declarative styles, together with the corresponding supporting infrastructure. "Declarative" in this context means that access operations form an algebra and entail semantic properties by induction in their structure. If this is done in a smooth and natural way, it will make program source texts more efficient and enjoyable to write, as well as better readable and maintainable.

There are techniques, like the *visitor* and *rewriter* patterns, which introduce a more declarative style of writing in object-oriented data evaluation. In [LT11], we have demonstrated how visitor-based extraction can be optimized using a combination of static and dynamic analyses. However, this technique corresponds to a more global, "point-free" way of formulating queries and is too heavy-weight and semantically too loosely defined for the purpose of point-wise extraction of details, for which local access operations *are known*.

Here, we investigate the import of a concept well-proven in all kinds of programming styles into the object oriented paradigm, namely *pattern matching*: The Paisley library presented below is a generic programming aid for data extraction by pattern matching that unifies desirable features of declarative paradigms with a pure object-oriented approach to data abstraction. It comes in two parts: a basic library and a programming idiom that uses the library operations as its core vocabulary. Problem-specific composite operations can be provided by the user by extending the library cleanly through subclassing. Our implementation is hosted in Java, but nothing prevents the same technique to be used in other strongly typed object-oriented environments such as C++ or .NET.

The present article extends an earlier tool demonstration paper [TL12] with more recent features, additional technical details and a comparative evaluation of the design. Detailed API documentation and downloads are available online at [TL11].

2 Standards of Pattern Matching

Pattern matching, in the wide sense, plays an important role in many different kinds of programming environments. But a close look shows that the techniques applied in the various fields differ substantially regarding theoretical foundation and expressiveness, the treatment of nondeterminism, type discipline, etc. These are the relevant role models, positive or negative, for our approach:

String Processing with Regular Expressions Here typing is a trivial matter, since patterns refer to character strings only. Theoretical foundation is sound; recently sound semantics have been defined even for backward group references [BC08]. Nondeterminism is either resolved locally by various flavours (greedy, reluctant etc.) of operators, or exposed to the user as global search. Focus is often on performance-critical applications, such as real-time filtering of high-frequency network traffic, making compilation to specially designed automata the technique of choice. Consequently, object-oriented data models play no role in those scenarios, and our approach does not apply.

Functional Programming with Algebraic Datatypes Inverse constructors are central to data extraction and equational function definition in functional programming languages (Hope, ML, Haskell, etc.), and share the full type discipline of the language. Nondeterminism arises not within one pattern, but rather between overlapping patterns of equations, and is usually resolved implicitly by a first-fit rule. In the context of the multi-paradigm language Scala, pattern matching has moved closer to object-oriented programming, by virtue of the `case class` construct and the `unapply` magic method; see [EOW07].

XML Navigation and Deconstruction For this purpose the XPath [CD99] pattern notation is the basis for the majority of transformation systems, like XSL-T, Xquery, XJ, Xact, JDOM and more. There is a complete theory for a subset of XPath excluding data value comparisons [GL06]. Alternative formalisms are less popular but extant; e.g. XDuce [HP00] is a functional language with patterns serving as types, with full static checking, and with regular expressions over patterns to match heterogeneous lists.

Logic Programming with Goals and Unification Logic programming languages (Prolog etc.) offer a distinct quality by making nondeterminism, unification of terms with free variables, and exhaustion of solution spaces (encapsulated search) first-class constructs of the language. They are usually weakly typed, but theoretically well explored.

Model Query and Transformation In dedicated model query languages pattern matching is a central functionality as well: the evaluation of a query delivers a subset of model nodes. Selection criteria range from simple checks on attribute values to complex relational constraints. In graph transformation systems, graph patterns feature prominently as the left hand sides of rewrite rules. The pervasive nondeterminism in graphs is often resolved by explicit control flow. See for instance the "Rule Application Control Language" of GrGen.NET [BGJ11]. Pattern notations take a vast number of theoretically and pragmatically distinct forms in the multitude of existing systems. For instance, the query language GReQL [EB10] offers regular path expressions to express complex patterns.

3 Design of Paisley Pattern Matching

3.1 Requirements

Porting pattern matching to an object-oriented environment is not a trivial task. On one hand, there are semantic problems to be solved, mostly pertaining to the impedance mismatch between object interfaces and algebraic pattern calculi. On the other hand, there is a multitude of theoretically possible implementation techniques. The Scala paper [EOW07] gives a good survey on different strategies, complemented with experimental evaluation. At the end of this article we will apply their criteria to our solution.

The Paisley approach is distinguished by a carefully selected canon of rigorous design requirements:

1. *Declarative, readable, writable, customizable.* Patterns express the programmer's intention of data extraction with as little formal noise as possible. This improves significantly over standard imperative/object-oriented patterns in terms of self-documentation and maintainability.

2. *Full reification: no parsing/compilation overhead at runtime.* Patterns are typed host-language objects; ill-defined usage is detected at compile time. This makes our approach diametrically opposed to dynamic notations, in particular traditional regular expression libraries such as `java.util.regex`.

3. *Statically type-safe variables.* No need to down-cast variable bindings or check their types at runtime.

4. *Statically type-safe patterns.* Detect ill-typed pattern matching attempts as often as possible.

5. *No language extension: independent of host compiler/VM.* Solution can be used transparently with off-the-shelf programming platforms and runtime environments.

6. *No assumptions on host language beyond standard OOP.* Solution can be reimplemented in any standard object oriented programming language. Custom extensions can use the full power of the host language, at the user's discretion and risk.

7. *No adaptation of model datatypes required.* Applies equally to data models from third-party repositories or developed without pattern matching in mind; no source access required.

8. *Support for multiple views per type.* Different collections of patterns can expose different structural aspects of a data model. Sharpens the preceding requirement.

9. *Support for continuation-style nondeterminism.* Patterns are ordinary objects with hidden inner state which *locally* and completely memorizes the current backtracking situation. Access to successive matches should be postponable indefinitely, even across serialization and de-serialization of all objects involved.

10. *Nondeterminism incurs no significant cost unless actually used.* Implies absence of central storage or control mechanisms, and lazy exploration of alternatives.

From the programming language perspective, the main focus is on *strict typing*. This is enforced by type relations of different kinds which are mapped to the type system of the host language, and thus inherit its checking and diagnostics facilities (for API see Fig.1):

1. *Pattern and data.* The type of data which can be matched against a given pattern is described by a parameter of the patterns' type: An instance of class `Pattern<A>` will match all instances of type `A`.

2. *Pattern lifting, contravariantly.* The type of any function which lifts a pattern on an object's field to a pattern on the object as a whole, or from a member object to a collection, etc., is always a function type between the corresponding pattern types: An access operation on class `A` that yields a subobject of type `B` induces a lifting function from `Pattern<B>` to `Pattern<A>`.

3. *Pattern combinators respect data types.* The Paisley pattern combinators require compatible types of the patterns' targets: A `Pattern<A>` and a `Pattern<B>` combined always result in a `Pattern<C>` where `C` is a subtype of both `A` and `B`.

4. *Pattern variables limit the type of their possible results.* On the construction side, a pattern variable has a type attributed with the type it can match, as any other pattern. After successful match, on the binding side, the variable offers a typed

getter interface: A `Variable<A>` is a `Pattern<A>` and yields values of static type `A`.

3.2 Basic Implementation Technique: DSL by Library + API

Pattern matching directives can be seen as a *domain specific language* (DSL) to be embedded into a general-purpose host programming language, in this case Java. For this there exist some well-known basic philosophies:

The requirements (2)–(4), for static type safety and reification rule out mere textual encodings, as criticized above. On the other hand, the requirement (5) for compiler independence rules out implicit compile-time handling of pattern matching code. Another possible solution is a *generative* approach, where DSL front-end syntax is translated into host language source code in a dedicated pre-processing step. This approach is used by many of the authors' other tools.

Here we chose instead an API and library-based implementation: Patterns are constructed at run-time, in terms of host language objects with certain *semantics*. We prefer this approach because it is far more lightweight and flexible. Of course, if appropriate, complex stereotypical code fragments on top of this library can be generated automatically from a more concise domain-specific notation, as for instance done by our umod tool [LT11].

3.3 Imperative View on Pattern Matching

The classical semantics of patterns as the inverse of constructor terms of algebraic datatypes, de-facto standard in declarative languages, does not carry over smoothly to the object-oriented paradigm, because object constructors generally lack the mathematical benevolent properties of their algebraic counterparts, namely extensionality, injectivity, disjointness and completeness.

A looser notion of pattern matching, more appropriate to the abstraction style of object orientation, is to consider it the reification and composition of certain categories of data extraction operations: *Testing* classifies objects as either acceptable or not; *projection* descends into the structure of the subobjects and extracts primitive data attributes; *binding* assigns data to variables.

These three aspects can be delimited precisely in well-written object-oriented code; they correspond to simple local idioms. A fourth aspect however, namely *logic*, is implicit and scattered across the control flow structure of code, in terms of sequences, conditionals, case distinctions, loops, etc. That the logical aspect comes with subtle and non-local ramifications should be evident to everyone who has successfully hand-coded a parser. This is a major source of difficulties in writing, reading and maintaining object-oriented code. Our central motivation behind the Paisley approach is to put logic on equals footing with the former three aspects in an object-oriented setting.

```
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);
}

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);
}

abstract class Transform<A, B> extends Pattern<A> {
  protected final Pattern<? super B> body;

  protected abstract B apply(A target);
  protected abstract boolean isApplicable(A target);
}
```

Figure 1: Interface synopsis (core)

The design of our library is such that these four concerns are separated as much as possible, but can be composed as freely as required. A notable implication is that logical structure, in particular with respect to nondeterminism, is given the most fundamental operator basis possible, namely fully compositional ad-hoc conjunction and disjunction of subpatterns, of which traditional pattern aggregation and case distinction are merely special cases.

3.4 The `Pattern` interface

The main interface of the library is the abstract base class `Pattern<A>` of patterns that can process objects of type `A`. A pattern `Pattern<A> p` is applied to some target data `x` of type `A` or any subtype by calling `p.match(x)`, returning a Boolean value indicating whether the match was successful.

In case the result is **`true`**, variables occurring in the pattern are guaranteed to be bound under conditions inductive in the logical structure of the pattern: A successful match binds variables in all branches of a conjunction, and in some branch of disjunction. Conversely, a variable is certainly bound if it occurs in all disjunctive branches or in some conjunctive branch. In case the matching result is **`false`**, variable bindings are unspecified.

After matching successfully, and using the values of bound variables, the parameter-less method `p.matchAgain()` may be called. This is how nondeterminism, that is the fact that a given pattern matches a given target *in more than one way*, is exposed at the interface.

The call of `matchAgain()` causes a new matching attempt of the same target by backtracking. The result has the same interpretation as for `match(x)`, so `matchAgain()` can be iterated as long as its result is **`true`**. The match is different in some way, in the sense the some new disjunctive branch is taken, in each successful call. The default implementation of `matchAgain()` always returns **`false`**, specifying a deterministic pattern.

Iteration over all possible matches of a nondeterministic pattern is effected simply by a **`do`** … **`while`** loop, with minimal redundancy:

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

3.5 Predefined Tests and Combinators

The Paisley library offers factory methods for patterns wrapping ubiquitous test and getter methods, and generic pattern combinators and liftings.

Basic rule for the whole implementation is strict typing, as postulated above in section 3.1. In this context it is essential to observe that all patterns except variables are *contravariant*: A pattern capable of matching any supertype `B` of `A` can act as a `Pattern<A>`, hence `Pattern<B>` should be treated as a subtype. This is expressed by library methods consistently taking parameters of wildcard types with lower bounds, in forms such as `Pattern<? super A>`.

In the current implementation there are static factory classes `ReflectionPatterns`, lifting some Java reflection operations such as `isInstance` or `getAnnotation`, as well as `StringPatterns` which lifts standard string operations like `startsWith`, but also the interface of the `java.util.regex` package. `PrimitivePatterns` wraps Java primitive types and some core methods such as `equals` or `compareTo`.

`CollectionPatterns` lifts patterns on elements to patterns on collections. This can be used as a controlled source of nondeterminism: Search patterns such as constructed by `anyElement(Pattern)` try all contained elements for `match()`/`matchAgain()`, while deterministic patterns such as constructed by `get(int, Pattern)` try to match only the one element at the given position. Variants for array types are also provided.

The class `Pattern` itself provides a framework of logical core combinators: binary operators `both` for conjunction and `either` for disjunction, to be discussed in detail in section 3.8 below; the constant patterns `any()` and `none()`, matching everything and nothing, respectively, as base cases; n-ary vararg combinator variants for convenience.

Modifications of the solution space of patterns are implemented as instance methods. `p.noMatch()` yields a pattern that matches if `p` itself has no solution. The match is deterministic and binds no variables; compare to negation-by-failure in logic programming. `p.uniquely()` matches iff `p` itself matches with exactly one solution, that is `p.matchAgain()` fails immediately. The match binds all variable also bound by `p`.

3.6 Variables

A pattern variable is simply a pattern of class `Variable<A>` that matches always, and binds to the matched object for later retrieval via the `getValue()` method. The variable interface is unique in the sense that its type parameter occurs in a return type, so it does not behave contravariantly as other pattern constructs do; cf. the preceding section 3.5.

The basic idiom of pattern matching is thus:

```
Variable<C> vc = new Variable<C>();
Variable<D> vd = new Variable<D>();
Pattern<A> p = myPattern(vc, vd);   // known to bind vc AND vd
if (p.match(x))
  doSomething(vc.getValue(), vd.getValue());
```

It is not by accident that the pattern variables `vc` and `vd` in this example have local declarations with precise static type (first two lines): This style enables the full use of static type information for bound values, even if the matching pattern has been constructed from generic building blocks that are defined independently of the type of occurring variables.

Figure 2 shows how variables are used in a Paisley compound pattern:

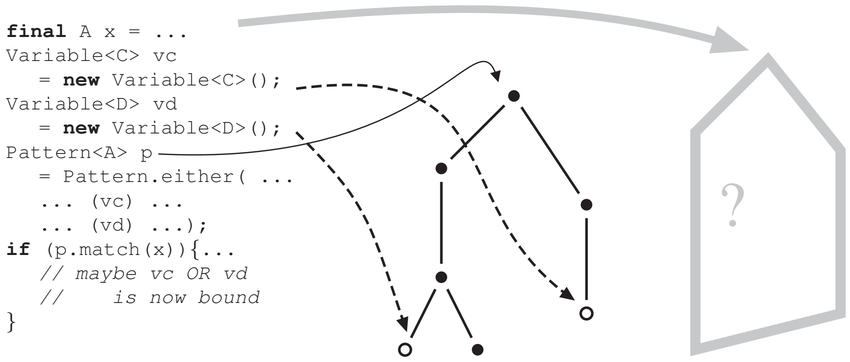
References to variables must be retained explicitly; they are not accessible via the containing pattern, as this would break compositionality. Therefore they must be constructed first, retaining a reference, before being incorporated into a newly constructed pattern. In order to safeguard against race conditions, it is good practice to give pattern variables local visibility only.

Variables are in the imperative style, that is simple, mutable containers for a single value; they do not provide either the unification functionality or the single-assignment/backtracking access mode of logical variables. Therefore, in most cases each variable appears exactly once in a given pattern, complex disjunctions aside.

After a successful match, variables may be bound to subobjects of the matched target datum. Whether a certain variable is bound or not may depend on the chosen alternative of a disjunction. The user is fully responsible for reading only bound variables.

Since variables have no distinguished initial "unbound" state, there is no *dynamic* check whether a variable has been bound by the most recent matching attempt: unsuccessful matches leave the occurring variables in unspecified state. Fortunately, the *static* effect of a pattern on given variables can be inferred inductively from its logical structure; the inference rules are available both as user documentation and runtime queries; details are beyond the scope of this article.

The advantage of this form of variable binding is that initialization costs are minimal and patterns can be reused without special preparation. It also implies that it is transparent to the user which branch of a disjunction has been taken; observed values of variables cannot be used to reconstruct the information. While this is the desired abstraction in most cases, special "marked" forms of disjunction (implemented by classes `IntBranch` and `EnumBranch`) can be used to retain the information for complex nested searches.



```
final A x = ...
Variable<C> vc
   = new Variable<C>();
Variable<D> vd
   = new Variable<D>();
Pattern<A> p
   = Pattern.either( ...
   ... (vc) ...
   ... (vd) ...);
if (p.match(x)){...
   // maybe vc OR vd
   //    is now bound
}
```

Figure 2: Explicit references to data, variables and pattern are required.

The restricted role of variables, although rather poor from a high-level declarative viewpoint, mimics closely the behaviour of local variables and fields in object-oriented programming, and should therefore feel natural to the programmer.

3.7 Encapsulated Search

For patterns with a single variable, bindings for all matches can be collected eagerly or lazily with `eagerBindings(Pattern)` and `lazyBindings(Pattern)`, respectively, thus effecting fully reified encapsulated search as strongly typed objects of the Java collection framework. The iteration pattern for all matches of pattern `p` for target `x` simplifies accordingly, with operational semantics equivalent to the loop given in section 3.4:

```
for (C c : vc.lazyBindings(p, x))
  doSomething(c) ;
```

The alternative is to calculate all possible matches before any processing, i.e. eagerly. This works of course only if the number of possible matches is finite:

```
List<A> list = vc.eagerBindings(p, x);
```

In the current implementation, simultaneous encapsulated search for multiple variables is not implemented, because of the lack of support for ad-hoc tuple types in Java.

3.8 Backtracking and Reentrance

The paradigmatic platform for backtracking nondeterminism is of course the Prolog language. Implementation hints can be gleaned from its operational semantics, in particular Warren's Abstract Machine (WAM) [War83].

The WAM uses to less than four categories of stack space (of which call stack and choice stack are interleaved to form the so-called local stack) to control nondeterminism and variable bindings. Fortunately for us, the simpler nature of imperative variables in Paisley and the underlying Java Virtual Machine (JVM), where no variable bindings, let alone object allocations, need to be undone by backtracking, allows a significant reduction of complexity: Besides the regular call stack, only a choice stack for yet unexplored disjunctive branches is required. And because the latter is strictly local to each matching, it can be implemented decentrally and transparently, distributed and hidden in the pattern combinator instances themselves. In contrast to an interleaving implementation, this requires no privileged access to the JVM stack, and is hence easily portable to other platforms.

The disjunctive pattern `either(p, q)` behaves as `p` initially, but switches to `q` after solutions to `p` are exhausted. The conjunctive pattern `both(p, q)` fixes a solution for `p` and produces solutions for `q`, but retries `p` and resets `q` whenever solutions to the latter are exhausted. These modes of operation subsume plain Boolean combination for deterministic argument patterns, and result in the concatenation and Cartesian product (in lexical order), respectively, of solutions for independent argument patterns. Overall behaviour can be more complex if `q` depends on `p` via observation of variable bindings or side effects; see section 3.12 below.

Note that a naive implementation of conjunction in terms of sequential matching attempts would not result in a full implementation of backtracking, but rather in the partial backtracking found in many simple search algorithms: The first argument pattern is committed to a solution before the second is tried; the latter is not reiterated for alternative solutions of the former. Contrast with the **`else`** branch in the code of Figure 3 below.

To illustrate the operation of the backtracking mechanism in detail, Figure 3 shows parts of the implementation of the `both` combinator. The local choice stack segment records whether the `left` argument pattern has succeeded and a reference to the matched target. The `match` method attempts to pair a solution for each argument pattern, and sets up the choice stack for later retrieval by `matchAgain()` as a side effect. The implementation of `matchAgain()` is almost identical, except that the grey statements are omitted and the underlined function head and recursive call are replaced by `matchAgain()`. The `either` combinator implements disjunction in an analogous manner.

As a consequence of the residual implementation of the choice stack, patterns are not thread safe: They can be reused sequentially, as required for `both` (see Figure 3), but not concurrently. On the upside, this allows for a simple, local and therefore highly efficient implementation of the cut: Patterns implement a method `cut()` that discards unused solutions with minimal effort, and a method `clear()` that additionally purges obsolete references from the choice stack, in order to control heap space usage even if pattern objects remain live for long times. Both methods descend recursively to subpatterns.

3.9 Specialized Pattern Libraries

The current implementation of Paisley comes with a few more application-specific groups of pattern combinators. In particular, the static factory class `XMLPatterns` supports content extraction and navigation along all axes of a W3C XML document object model

```
private Pattern left, right;                       // pattern tree
private A       target_save;                       // local choice stack
private boolean left_matched;                      // local choice stack

public boolean match(A target) {
  if (left_matched = left.match(target)) {
    target_save = target;                          // for use by matchAgain()
    if (right.match(target)) return true;
    else
      while (left_matched = left.matchAgain())
        if (right.match(target_save)) return true;
    target_save = null;                            // solutions exhausted
  }
  return false;
}
```

Figure 3: Implementation of backtracking (excerpt)

(DOM [HHW+00]). It shows compositional abstraction through pattern lifting, and non-invasive "patternification" of an existing, abstract data model, namely the standard Java package `org.w3c.dom`. Showcase examples using this and other pattern factories are included in the Paisley download package at [TL11]. Figure 4 shows an excerpt that deals with "glossary entries" in an XHTML document, that is adjacent *term–description* (`<dt>`–`<dd>`) pairs in a *definition list* (`<dl>`). In particular, the self-contained example code extracts from a document the full relation between terms and *hyperlinks*, that is *anchors* (`<a>`) with a `href` attribute, contained in the respective following descriptions. Note how the implementation is, mandatory formal noise of Java aside, hardly more complicated than the prose description.

This example shows the classical way of using patterns, where the variables appear in leaf position, and the context is specified ("generate"). Paisley supports the symmetric situation where the content of the bound structures is narrowed further ("test"). For instance, extending the pattern `r` to

```
  glossaryPair(both(dt, textContent(startsWith("c"))),
               descendant(anchorWithHRef
                            (both(href, not(startsWith("#")))) ))
```

will match only glossary entries starting with lower case "c" and anchors to non-local hrefs.

For other, user-defined tasks the implementation strategy is similar: encapsulating all dirty details of testing, iterating, backtracking and cutting into library patterns, thus creating a clean basis on which the operational code can be formulated in an intentional, declarative way.

```
import eu.bandm.tools.paisley.*;
import static eu.bandm.tools.paisley.Pattern.*;
import static eu.bandm.tools.paisley.XMLPatterns.*;
import org.w3c.dom.*;

class XML_example {
  final static String XHTMLNS = "http://www.w3.org/1999/xhtml";
  static Pattern<Node> xhtmlElement(String localName,
                                    Pattern<? super Element> element) {
    return element(both(name(XHTMLNS, localName), element));
  }
  static Pattern<Node> glossaryPair(Pattern<? super Element> dt,
                                    Pattern<? super Element> dd) {
    return both(xhtmlElement("dt", dt),
                nextSibling(xhtmlElement("dd", dd)));
  }
  static Pattern<Node> anchorWithHRef(Pattern<? super String> href) {
    return xhtmlElement("a", attrValue("href", href));
  }
  final Variable<Element> dt = new Variable<Element>();
  final Variable<String> href = new Variable<String>();
  final Pattern<Element> r =
    glossaryPair(dt, descendant(anchorWithHRef(href)));
  final Pattern<Document> p = root(descendantOrSelf(r));

  { // ... let "doc" be a w3c dom representation of an XHTML document
    if (p.match(doc)) do {
      System.out.println("Glossary␣entry␣\""
                         + dt.getValue().getTextContent()
                         + "\"␣refers␣to␣\"" + href.getValue() + "\"");
    } while (p.matchAgain());
  }
}
```

Figure 4: XHTML Glossary Entry Example

3.10 Projection and Testing

The base case of data extraction is a conceptual total function $f : A \to B$; the archetypal example being a getter method in class `A` which reads some member field of type `B`. This induces a function that maps each pattern `p` of type `Pattern<B>` to a pattern of type `Pattern<A>` that matches a target x by having `p` match $f(x)$. Given a suitable reification `f` of f, this can be written in Paisley simply as `transform(f, p)`.

The more general case is that of partial functions, where an undefined result causes the matching to fail. These are realized conveniently as subclasses `T` of `Transform<A, B>`, which implement the **`boolean`** `isApplicable(A)` and `B apply(A)` methods explicitly, with the obvious semantics.

In principle, `Transform` is a complete basis for projection and testing, that is for all functional pattern components except variables. Of course, hand-coded projections/tests can be defined by the user, where more convenient or efficient for the application at hand.

3.11 Pattern Substitution and Closure

Apart from extracting data from a match, variables also play a meta-level role as hooks for pure pattern algebra, thus enabling powerful generic abstractions and transformations.

The most basic one is substitution, which enables pattern parametrization: In the code fragment

```
Variable<V> v = new Variable<V>();
Pattern<R> top = ... (v) ... ;
Pattern<V> sub = ... ;
Pattern<R> newTop =  v.bind(top, sub);
```

`newTop` denotes a pattern in which every occurrence of `v` is replaced by a reference to `sub`. The implementation works non-invasively by recursively nested matching; hence, even hand-coded patterns which hide their logical structure (if any) can safely be substituted into. Duplication of substitution replacements is avoided by exploiting sequential reusability of patterns. As a moniker for the programming interface, read `v.bind(p, q)` as the lambda calculus expression $(\lambda v.p)q$.

In extension of the quantifier perspective on patterns, variables also serve as the bridgehead of “star” or “plus” Kleene closures:

```
Variable<V> v = new Variable<V>();
Pattern<V> once = ... (v) ... ;
Pattern<V> some = v.star(once);
Pattern<V> more = v.plus(once);
```

The newly constructed patterns `some`/`more` are the star/plus closures, respectively, of the path relation between `once` and `v`, insofar as they obey the expected mutually recursive behavioral equivalence relation

$$\texttt{some} \equiv \texttt{either(v, more)} \qquad \texttt{more} \equiv \texttt{v.bind(once, some)}$$

Using these, the complex pattern constructor `XMLPatterns.descendantOrSelf(p)` featured in Figure 4 can be defined concisely as

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

where `v` is a fresh variable, and the primitive `child` is implemented in terms of the canonical `org.w3c.dom.Node` getters `getFirstChild()` and `getNextSibling()`. Note how the two sources of nondeterminism, regarding horizontal (`child`) and vertical (`star`) position in the document tree, respectively, combine completely transparently.

3.12 Breaking the Paradigm

The “pragmatic philosophy” of Paisley is to leverage a declarative style of writing and thinking, while the code “behind the scenes” remains genuinely imperative, with no intermediate transformation.

Whether any knowledge of the implied control flow is considered part of pattern semantics, is a fundamental decision on the level of "coding style guidelines", and its consequences must be considered carefully. Here clearly a Rubicon would be crossed.

On the upside, by taking advantage of sequential execution order, powerful functionalities like non-linear patterns can be implemented. The following pattern p, built using the Paisley XML facilities discussed in section 3.9, matches all XHTML anchors which contain their target URL literally in their text content:

```
Variable<String> v = new Variable<String>();
Pattern<Element> p =
  all(name(XHTMLNS, "a"),
      attrValue("href", v),
      descendant(textContent(contains(v.getValue()))));
```

On the downside, it is evidently easy to accidentally break the declarative matching semantics, resulting in all the kinds of subtle and hard-to-debug program behaviour we set out to avoid with our design in the first place. Extending the library in this way, while technically supported and perhaps pragmatically valid, is certainly a different use case altogether; the two should be distinguished carefully for fundamental software engineering reasons.

4 Conclusion

4.1 Related Work

A theoretically elegant design of pattern matching capabilities for Java, JMatch, is presented in [LM03]. While it has had much impact, and is cited heavily by later work, there are severe drawbacks: The approach assumes a perspective on pattern matching that is very much like logical programming. As a result, their nondeterminism is rather heavy-weight: It requires CPS transformation of certain program parts, and hence interferes severely with apparent control flow, making program understanding and debugging forbiddingly difficult. Furthermore, the solution is a host language extension and requires a special academic compiler. All such experiments are eventually doomed to oblivion unless some big vendor adopts the technology.

As mentioned above, the multi-paradigm language Scala [OSV10] incorporates a powerful pattern matching idiom with clean semantics and user-defined extensibility, via singleton objects and the `unapply` method [EOW07]. Being part of the core Scala design, it is better integrated with the host language than our approach can ever hope to be. On the other hand, we find the lack of nondeterminism and pattern algebra significant weaknesses.

4.2 Comparative Evaluation Framework

Concerning the evaluation of the design of different pattern matching mechanisms, a paper [EOW07] from the Scala context introduced a grid of nine criteria. Omitting the last three, which deal with concrete performance measuring and cannot easily be reconstructed, we find that Paisley corresponds largely to the "extractor" type of solution discussed in that paper, with some notable differences.

Conciseness of the Framework Programming overhead is required for the Paisley projection operations, see section 3.10 above. They correspond to the "extractors" in Scala, and are a little more verbose than those, obviously due to the less expressive host language syntax. Furthermore, the chain of delegation to embedded Paisley patterns must always be written down explicitly, whereas in many situations a call to `unapply` will silently be inserted in the Scala approach.

The disadvantages of Paisley are offset to some degree by the superior abstraction capabilities, whereby for many applications predefined, highly generic library building block for patterns are provided ready-to-use. Additionally, the Paisley approach does not share the weakness of Scala pattern matching that descent into the structure of an object is reified, and hence boxed and unboxed, at every level of `unapply`.

Conciseness of Shallow/Deep Matches The syntax of a concrete application of a complex Paisley pattern has least possible syntactic noise. A source of noise that cannot be evaded by the nature of our design is the absence of a *rule* concept in Paisley: There is no equivalent of the Scala matching block that follows a `match` operator. Instead, the scheduling of alternative patterns (rule left-hand sides) and the association of corresponding reactions (rule right-hand sides) is expressed in the hosting object-oriented style.

Maintainability: Representation Independence No internal representation at all need be revealed, because only the functional testing/projection interfaces have to be implemented. See the XML DOM example cited in Section 3.9 above. Nevertheless, in many cases data abstraction is trivial, so the extractors will follow the internal structure naturally.

Maintainability: Extending (Data) Variants / Patterns The data and the pattern world may grow arbitrarily without mutually affecting the behaviour of older model class definitions and patterns. Since patterns are only defined by their terse functional interface (see Section 3.4), arbitrary new variants can be added, and existing combinations can freely be abstracted at any time.

In terms of compositionality, this goes far beyond Scala extractors: Being transparent regarding nondeterminism, also arbitrary disjunctions and even encapsulated searches can be abstracted to self-contained pattern components.

References

[BC08] Becchi, M.; Crowley, P.: Extending finite automata to efficiently match Perl-compatible regular expressions. In Proc. 2008 ACM CoNEXT Conference (CoNEXT '08). ACM, New York, 2008; S. 25:1–25:12.

[BGJ11] Blomer, J.; Geiß, R.; Jakumeit, E.: The GrGen.NET User Manual. http://www.grgen.net, 2011.

[CD99] Clark, J.; DeRose, S.: XML Path Language (XPath) Version 1.0. W3C, http://www.w3.org/TR/1999/REC-xpath-19991116/, 1999.

[EB10] Ebert, J.; Bildhauer, D.: Reverse Engineering Using Graph Queries. In (Schürr, A.; Lewerentz, C.; Engels, G.; Schäfer, W.; Westfechtel, B. Hrsg.) Graph Transformations and Model Driven Engineering, LNCS 5765. Springer Verlag, 2010.

[EOW07] Emir, B.; Odersky, M.; Williams, J.: Matching Objects with Patterns. In (Ernst, E. Hrsg.) Proc. 21st ECOOP, LNCS 4609. Springer, 2007.

[GL06] Genèves, P.; Layaïda, N.: A System for the Static Analysis of XPath. ACM Trans. Inf. Sys. 24, 2006.

[HHW$^+$00] Le Hors, A.; Le Hégaret, P.; Wood, L.; Nicol, G.; Robie, J.; Champion, M.; Byrne, S.: Document Object Model (DOM) Level 2 Core Specification Version 1.0. W3C, http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/, 2000. Recommendation.

[HP00] Hosoya, H.; Pierce, B.C.: XDuce: A Typed XML Processing Language. In Proc. 3rd Workshop on the Web and Data Bases (WebDB 2000), S. 226–244. Springer Verlag, 2000.

[LM03] Liu, J; Myers, A.C.: JMatch: Iterable Abstract Pattern Matching for Java. In Practical Aspects of Declarative Languages, LNCS 2562. Springer, 2003.

[LT11] Lepper, M.; Trancón y Widemann, B.: Optimization of Visitor Performance by Reflection-Based Analysis. In (Cabot, J.; Visser, E. Hrsg.) Theory and Practice of Model Transformations, LNCS 6707. Springer Verlag, 2011.

[OSV10] Odersky, M.; Spoon, L.; Venners, B.: Programming in Scala. artima, 2nd edition, 2010.

[TL11] Baltasar Trancón y Widemann and Markus Lepper. *Paisley Download and Documentation Page*. http://www.bandm.eu/metatools/docs/usage/paisley-download.html, 2011.

[TL12] Baltasar Trancón y Widemann and Markus Lepper. Paisley: pattern matching à la carte. In Proc. 5th International Conference on Model Transformation (ICMT 2012), LNCS 7307. Springer Verlag, 2012; S. 240–247.

[War83] Warren, D.: An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, 1983.