Paisley: A Pattern Matching Library for Arbitrary Object Models

Baltasar Trancón y Widemann[1,2] and Markus Lepper[2]
Baltasar.Trancon@uni-bayreuth.de

[1] University of Bayreuth, Germany
[2] <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 neatless as possible and without the need for the programmers to leave the environment they are used to. Here we present a tool for the implementation of *pattern matching* as fundamental means of automated data extraction from complex and arbitrarily shaped models 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 writeable, 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 when creating compound data, and when extracting their components: They support pattern matching directly in the front-end syntax of function definition. Object oriented languages mostly offer the very desirable feature of free compositionality only when creating objects, but totally lack a corresponding idiom for extraction. Instead, explicit getter methods, type casts, assignments to temporary variables and explicit case distinction have to be applied in an imperative way of programming, which is deeply inadequate to the purpose of mere search and inquiry.

Obviously it is desirable to enrich object oriented programming practice by techniques from more declarative languages, together with the corresponding

supporting infrastructure. If this is done in a smooth and natural way, it will make program source texts more easily writeable and better readable and maintainable.

There are adequate techniques, like the *visitor pattern* and the rewriter pattern, which introduce a more declarative style of writing in object oriented data evaluation. In [7], 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-less" way of formulating queries and is too heavy weighted and too loosely defined for the purpose of point-wise extraction of details, which *are known* to exist locally.

Instead, we try to port a concept well-proven in very diverse other disciplines into the object oriented paradigm, namely *pattern matching*; The Paisley library presented herein is a generic programming aid for data extraction by pattern matching that unifies desirable features of declarative paradigms with an 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.

2 Standards of Pattern Matching

Pattern matching, in the wide sense, plays an important role in many different kinds of progamming 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 not an issue, since patterns refer to character strings only. Theoretical foundation is sound; nondeterminism is supported by interspered "dot-star-terms" and must be treated specially when compiling. Recently sound semantics have been defined even for backward group references [1]. Focus is oftenly on "processing at line rate", meaning real-time filtering of high frequent network traffic. In this field compilation to specially designed automata is the technique of choice. This is a field esp. *not covered* by our approach.

Functional programming with algebraic datatypes. Inverse constructors are a central means of data extraction and equational function definition in functional programming languages (Hope, Haskell, ML, Opal), and shares 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, esp. by the construct of "case classes", see [5].

XML processing with XPath and XSLT. XPath pattern definition has become practically relevant as a central part of the XSL transformation system. There is no commonly agreed theory for the full scope of the pattern language, but for restricted subsets [2]. Nondeterminism is supported by the "`for-each`" statement of the XSLT language, although control is limited to complete enumeration.

Logic programming with goals & unification. Logic programming languages (most famous: Prolog) offer a distinct quality by making nondeterminism, unification of variables and values, and automated exploration of solution spaces 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 among nodes. In graph transformation systems, graph patterns feature prominently as the left hand sides of rewrite rules. The pervasive nondeterminism in graphs is often handled by explicit control flow. See for instance the "Rule Application Control Language" of GrGen.NET [3]. Pattern notations take a vast number of theoretically and pragmatically different forms in the multitude of existing systems. For instance, the query language of GrGen, GReQL [4], offers regular path expressions to express complex patterns.

3 Design of Paisley Pattern Matching

3.1 Requirements. Static Typing

Porting pattern matching to an object oriented environment is not trivial. On one hand, there are semantic problems to be solved, on the other hand there is a multitude of theoretically possible implementation techniques. The abovementioned work from the Scala context gives a good survey on different strategies, even with experimental evaluation [5]. At the end of this article we will apply their criteria to our solution.

Our approach is distinguished by a carefully selected canon of rigorous design requirements. The main focus is on *strict typing.* This is maintained by type relations of different kinds, which are mapped to the host languages type system, thus always guaranteed:

1. *Pattern and data* The type of data which can be matched against a given pattern is described by a type argument to the patterns' type.
2. *Pattern on components vs. pattern on collection* 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.
3. *Pattern combinators respect data types.* The pattern combinators from the Paisley library require compatible types of the patterns' targets.

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 the bound datum by a fully typed getter interface.

In detail, our requirements are ...

1. *Declarative, readable, writeable, customizable.* Patterns express the programmer's intention of data extraction with as little formal noise as possible. This is the main target of all our efforts: Making program code more self-documenting and better maintainable, compared to the genuine imperative programming.
2. *Full reification: no parsing/compilation overhead at runtime.* Patterns are typed host-language objects; ill-defined usage is detected at compile time.

   A counter example is the interface of the "`javal.util.regex`" package: Regular expressions must be defined as *textual strings*, which are analyzed and compiled *at run-time*. This kind of design flaw is, what our design property forbids: The adequate interface would have been a collection of *typed* constructors/factory methods, the use of which cannot fail at run-time. (Such a string encoded interface can always be provided *additionally*, for all those programmers who never make any mistake when counting backslash-escaped escape-sequences of back-slash escaped back-slashes !-)
3. *Statically type-safe variables.* No need to down-cast variable bindings.
4. *Statically type-safe patterns.* Detect ill-typed pattern matching attempts as often as possible.
5. *No language extension: independent of host compiler/interpreter.* Solution can be used with off-the-shelf programming platforms.
6. *No assumptions on host language beyond standard OOP.* Solution can be re-implemented in any standard object oriented programming language.
7. *No adaptation of model datatypes required.* Data models can be developed without pattern matching in mind; no source access required. This is demonstrated in the example in section 4.1, where a third party data type from a binary library, with an exceptionally ugly interface, is niecely abstracted into our framework.
8. *Support for multiple views per type.* This even sharpens the preceding requirement: Different collections of pattern definitions can expose different structural aspects of data models.
9. *Support for continuation-style nondeterminism.* Patterns are normal objects with some inner state, which *locally* and completely memorizes the current backtracking situtation. The call for 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.* This implies no central storage or control mechanism, and lazy exploration of alternatives.

3.2 Basic Implementation Technique: DSL by Library / API

Pattern matching directives can be seen as a *domain specific language* / DSL, we want to integrate into a general purpose programming language, here: Java. For this there exist some well-known basic philosphies:

The requirements (2) to (4), for static type safety and reification rule out mere textual encodings, as critized above. On the other hand, the requirement (5) for compiler independence rule out implicit compile-time generation of pattern matching code.

As possible solution remains a *generative* approach, where DSL front-end syntax is translated into source code, in a dedicated pre-processing step. This approach is used by many others of the authors' tools, e.g. umod.

Here we prefer an API and library based implementation: patterns are constructed at run-time, in terms of host language objects, which do carry certain *semantics*. We prefer this approach because it is more lightweight and flexible. (Of course, whenever appropriate, complex fragments of code atop of this library can still be *generated* from a more concise domain-specific notation, as for instance done by our umod tool [7]).

3.3 An Imperative View to 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 extensional equality, 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 classes of data-extraction operations, namely

1. *testing*, classifying objects as either acceptable or not,
2. *projection*, descending into the structure of the sub-objects,
3. *binding*, assigning data to variables.

The design of our library is such that these three concerns are separated as much as possible, but can be composed as freely as possible.

3.4 The `Pattern` interface

(For more details of the subjects of this and the following sections, please also refer to the online-api-doc in [10, eu/bandm/tools/paisley/package-summary.html].)

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 data `(? extends A)x` by calling `p.match(x)`. This returns a Boolean value indicating the success of the match.

```
abstract class Pattern<A> {
  public abstract boolean match(A target);
  public boolean matchAgain();

  public Pattern<Object> forInstancesOf(Class<? extends A> cls);
  public static <A> Pattern<A> eq(A constant);
  public static <A> Pattern<A> equal(A constant);

  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);

}


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);
}

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);
}
```

Fig. 1. Interface synopsis (Core)

In case the result is true, all variables in the pattern will be bound, iff they occur in a branch of a disjunction which contributed to the match. In case the result is false, no variable does have any guaranteed meaning.

In case the result is true, the parameterless method `p.matchAgain()` can be called. This is how *nondeterminism* is realized, how the fact is signalled that a certain pattern matches a certain datum *in more than one way*.

The call of `matchAgain()` causes a new match attempt of the same datum. The result is again a Boolean, reflecting a match result which is guaranteed to be different from any previous result. Meaning and possible reaction are the same as with the first match, so `matchAgain()` can be called as long as its result is true.

The default implementation of `matchAgain` always returns **`false`**, specifying a deterministic pattern.

Iteration over all 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 Re-usage and re-entrance

Any pattern can be re-used with the same or some different datum, but not in parallel.

3.6 Predefined Tests and Combinators

The Paisley library offers factory methods for patterns which wrap diverse tests, and for combining patterns.

Basic rule for the whole implementation is strict typing, as postulated above in section 3.1. In this context it is of utmost importance that most Patterns are *contravariant*: A pattern capable of matching any superclass of A is a sub-type of `Pattern<A>`, in the meaning that it may be substituted for a such.

In most places the type of a pattern required e.g. as a certain function argument is only specified modulo this contravariancy. This can (and must) be expressed in Java by wildcards with lower bounds, of the form `Pattern<? super A>`.

In the current implementation there are the classes `ReflectionPatterns`, lifting some genuine `java.lang.reflect` methods, `StringPatterns`, which lift not only the standard string predicates like `String.startsWith()`, but also the main method of the `java.util.regex` package. `PrimitivePatterns` realizes some more tests on numbers and Booleans.

`CollectionPatterns` construct patterns on collection types from patterns on the elements contained therein. For this, several variants are possible. E.g. `anyElement(Pattern)` tries all contained elements for `match()`, and afterwards for `matchAgain()`, while `get(int,Pattern)` tries to match only the one element at the given position.

(Please note: for technical reasons the distinction between `java.lang.Iterable` and array types cannot be resolved by overloading, but has been mangled into the naming, using "`Array`" as a name component.)

The class `Pattern` itself offers ...

- `any()`, delivers a pattern that always matches,
- `none()`, delivers a pattern that never matches,
- `Pattern.and(Pattern)`, conjunction of two patterns,
- `Pattern.or(Pattern)`, disjunction of two patterns,
- the static combinators `both(...)`, `either(...)`, `all(...)`, `some(...)`, allowing to combine more than two patterns conjunctively or disjunctively.
- the local function `p.noMatch()`, deriving a pattern which matches iff `p` does not match,
- and `p.uniquely()`, deriving a pattern which matches iff `p` does match and `matchAgain()` fails.
- `Pattern<A>.compareTo(A, Pattern<int>)`, deriving a pattern which first executes the `A.compare(A)` method, and then applies the `Pattern<int>` to its result.

The `EnumBranch` and the `IntBranch` classes also implement a disjunction, but additionally offer a `getKey()` method which after each successful

(re-)match tells the user which of the branches has been taken. In many situations this allows to combine case distinctions with expensive analysis common for all cases.

3.7 Specialized Pattern Libraries

The current implementation of Paisley comes with some more specialized subclasses of `Pattern`. E.g., the class `XMLPatterns` offers content extraction and navigation on all axes of a "document object model" as defined by the W3C XML DOM definition [6]. It will be used in the example in section 4.2 below.

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

3.8 Projection and search

Given a total function `f` which maps all objects of class A to one of class B. Typical example: a getter function which reads a certain field of type B from an object of class A.

Then `Pattern<B>.transform(f)` yields a pattern of type `Pattern<A>`: Given a pattern `p` which tries to match field values, the pattern `p.transform(f)` will match the containing objects.

The more general case is that of partial functions. These must be realized by a sub-class `T` of `Transform<A,B>`, which defines the `boolean isApplicable(A)` and `B apply(A)` methods explicitly.

Every instantiation of such a transform class `new T(Pattern<B>)` will act as a pattern on `A`, simply by applying the transformation and subsequent matching.

3.9 Variables

A variable is simply a pattern of subclass `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 in a contravariant way, as most other pattern factories do, see section 3.6 above.

Binding occurs as a side effect which is meaningful only for successful matches; unsuccessful matches leave the corresponding variables is unspecified state. For reasons of simplicity and efficiency, our library does not provide automatic means to detect whether a variable has been bound. When a variable shall be read which does not appear in every branch of a disjunction, the class `EnumBranch` or `IntBranch` can be employed to distinguish the cases.

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 an accident that the pattern variables `c` and `d` 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 memorized explicitly, they are not adressable via the containing pattern.
- Therefore they must be constructed first, the reference to them must be memorized, and then they can be built into a newly constructed pattern.
- Variables are simple "storages" for values, they do not provide the unifcation functionality of "logical variables".
- Therefore, in most cases a certain variable appears only once(1) in a certain pattern.
- After a successful match, variables may be bound to sub-structures of the matched 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.
- There is neither an automated initialization nor a re-initialization when calling `match()` or `matchAgain()`. Consequently, the contents of a variable are not suited to decide which branch of an alternative has been taken.

The very restricted role of variables should not be a problem from the programmers point of view, because it corresponds to the rather primitive interface of object oriented "instance fields".

For patterns with a single variable, bindings for all matches can be collected either eagerly or lazily with `eagerBindings` and `lazyBindings`, respectively, thus effecting fully reified encapsulated search as strongly typed objects of the Java collection framework. The iteration pattern for all matches simplifies accordingly.

```
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, these both features cannot be abstracted to more than one variable, only because Java lacks the possibility to define *ad hoc* typed n-tuple.

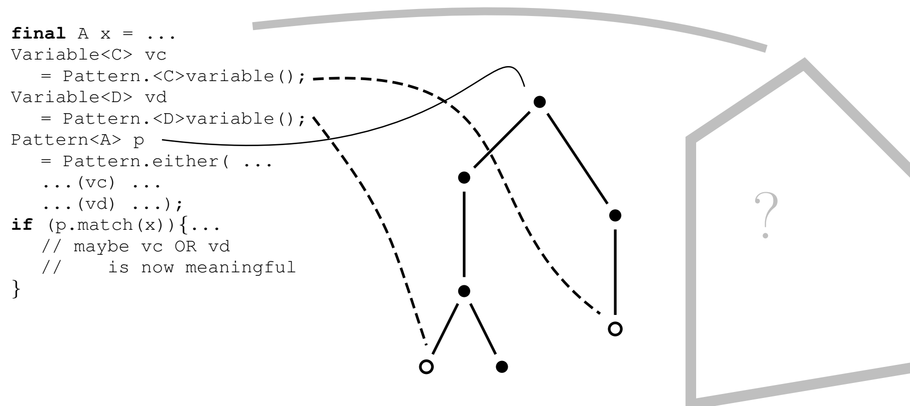

Fig. 2. Explicit references to data, variables and pattern are required.

3.10 Parametrizing Patterns with Sub-Patterns. Star and Plus Closures

Furthermore, variables are the basis for two fundamental generic transformations of patterns:

First, in the code fragment

```
Variable<V> v = Pattern.<V>variable();
Pattern<R> topPattern = ... (v) ... ;
Pattern<V> subPattern = ... ;
Pattern<R> newTopPattern =  v.bind(topPattern, subPattern);
```

`newTopPattern` will be a pattern in which every occurence of `v` is replaced by the `subPattern`. Thus variables are the central means for *paremeterizing* patterns.

As a consequence, they serve as the bridge-head of "star" or "plus" Kleene closures:

```
Variable<V> v = Pattern.<V>variable();
Pattern<V> once = ... (v) ... ;
Pattern<V> multi = v.plus(once);
```

The newly constructed top-level pattern `multi` is the "Plus" closure of the pattern `once`, insofar as every occurence of the variable `v` leads to a binding for this variable, but, as a nondeterministic alternative when calling `matchAgain()`, also leads to a new instantiation of the pattern, starting at this very point. The operator `v.star()` behaves accordingly.

3.11 Reflection on Patterns

The API of Pattern comes with a small number of reflection methods: `isDeterministic()` yields whether the pattern may match at most once. Two variants of `preserves()` report whether a given variable is not assigned during match execution.

3.12 Breaking the paradigm

While introducing declarative style of writing and thinking, the code "behind the scene" remains imperative, i.e. is executed in a certain fixed sequential order. Whether any knowledge of this order is considered part of its 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.

When using the information of sequential execution order you can realize powerful functionalities. E.g., the example in section 4.3 shows, how the behaviour of *logical variables* can be mimicked. But then you loose the declarative nature of the DSL, you loose the independence of the concrete form of the patterns beyond their matching semantics.

It must be clear to the user that these both ways of employing our tools substantially differ.

4 Examples

The following examples (source text and ready-to-run .jar files) can be retrieved from `http://bandm.eu/metatools/paisley` The examples can easily be executed following the instructions on this website.

Please refer also to the API DOC at `http://bandm.eu/metatools/docs/api/index.html`, [10].

4.1 Matching Fools Days

The Java runtime library comes with a package which models calendaric date. Its central classes, `GregorianCalendar`, `Calendar`, etc., have a rather awkward interface: properties must be identified by predefined integer constants and can only be handled by calling a single getter/setter method. Additionally there is a complex behaviour of inner state, and the the first step every developer will do when condamned to use these classes is to define a sensible façade!

The listing in figure 3 shows different user-defined subclasses of our `Pattern` class, which match any calendaric date which (a) belongs to a *First of April*, and (b) the year of which matches a further pattern.

`foolsDay1` does so by testing all relevant properties directly in the body of the `match()` method, using the host langugage logical "and".

`foolsDay2` uses the conjunctive pattern combinator instead. The `get()` method is a typical projection in the sense of section 3.8: It delivers a pattern for a `Calender` by lifting a pattern for the indicated component.

foolsDay3 uses one more intermediate level: The projection functions for certain properties are reified, and the top-level pattern gets even more readable.

The test function initTestdata() fills an array with pseudo-random calendaric data.

Then

```
anyArrayElement(foolsDay(any())).match(x));
```

answers the question whether any of the dates matches a fools day, while

```
anyArrayElement(foolsDay(eq(1984))).match(x));
```

says whether the fools day of the year 1984 is contained in the data.

The following code iterates over all matches and prints them to stdout:

```
final Variable<Integer> year = new Variable<Integer>();
final Pattern<Calendar[]> p = anyArrayElement(foolsDay(year));
if (p.match(x)) do
  System.out.print(year.getValue() + "␣␣");
while (p.matchAgain());
```

Please note that the "year" argument to the fools day pattern is itself a pattern, not a simple integer. This pattern can in turn be arbitrarily complicated. E.g., the following retrieval delivers all fools days from the test data which fall into even-numbered years:

```
final Variable<Integer> year = new Variable<Integer>();
final Pattern<Integer> even
 = new Atomic<Integer>(){
     public boolean match(Integer i){
       return (i%2)==0;
    }};

final Pattern<Calendar[]> p2
  = anyArrayElement(foolsDay(year.and(even)));
System.out.println("even␣years:␣" + year.eagerBindings(p2, x));
```

4.2 Matching XHTML Data

In this example we analyze the XHTML document from http://www.w3.org/TR/2002/REC-xhtml1-20020801. This document is itself (self-application !-) an XHTML document. As such it contains a very ugly construction, called "definition list". This is an element with tag dl containing an *alternating sequence* of "definition terms" and "definition definitions", standing for the components of a *glossary*, namely terms and a their explanation. The ugly fact is that the relation between both is only by position, and is not reified/tagged.

Paisley comes with a library eu.bandm.tools.paisley.XMLPatterns, which deals with XML Document Object Models, as defined bie W3c [6]. This library contains patterns to filter for elements, attributes, attribute contents, and to follow all *axes* defined in the standard, see the declarations in figure 4.

Figure 5 shows the routines needed to identify the dt/dd pairs.

Assumed that doc contains the document object of the above-mentioned XHTML specification document, in any implementation adhering to the Java anguage binding of W3C XML DOM. Then the following code can be applied:

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

//...

  final Variable<Element> dt = new Variable<Element>();
  final Variable<String> href = new Variable<String>();
  final Pattern<Document> p =
    root(descendantOrSelf(
           defPair(dt, descendant(
                         anchorHRef(all(href,
                                        localURL(false)
                                        )))
                   )));
  if (p.match(doc)) do {
    System.out.println(String.format("%s␣refers␣to␣%s",
                                     dt.getValue().getTextContent(),
                                     href.getValue()));
  } while (p.matchAgain());
//..
```

The printout shows every *external* link appearing in the text of a dd entry, preceded by the dt term that text is related to, see figure 6.

Please note the different levels of non-determinism, unrolled "behind the scene": All immediately adjacent pairs dt/dd are found, at arbitrary deep nesting level in the document, and in each such dd contents all a-elements are found with a non-local URI.

4.3 Mimicking logical variables

As mentioned above in section 3.12, it is possible (but not always advisable !-) to take into account the sequential order of evaluation to achieve powerfuls effects, as in the following code:

```
final Variable<Element> dt = new Variable<Element>();
final Variable<String> href = new Variable<String>();
final Pattern<String> containsKeyword = new Atomic<String>(){
  public boolean match(String x) {
    return textContainsDefTerm(x, dt.getValue().getTextContent());
  }
};
final Pattern<Document> p =
  root(descendantOrSelf(
         defPair(dt, descendant(
                       anchorHRef(all(href,
```

```
                                        localURL(false),
                                        containsKeyword
                                        )))
                    )));
    // etc.
```

In this version the pattern "containsKeyword" refers to the variable dt and thus to the <dt> element which has been matched by that variable in the same match attempt as the pattern is called. This corresponds to the current "terminus under explanation", the explanation text of which contains the matched <a>. Consequently, in contrast to the first example, here only those URIs will be matched which themselves do contain the keyword *verbatim*, modulo case. Now only those lines from figure 6 appear which are marked with ⋆.

So this technique of cross-refering to already bound pattern variables can realize a behaviour like "unification", as known from true "logic variables".

5 Conclusion

5.1 Related Work

A theoretically elegant design of pattern matching capabilities for Java, JMatch, is presented in [8]. 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, requiring CPS transformation of certain program parts. 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 [9] incorporates a powerful pattern matching idiom with clean semantics and user-defined extensibility, via singleton objects and the unapply method [5]. 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 of the "star-closure" significant weaknesses.

5.2 Applying the Evaluation Criteria of Emir, Odersky and Williams

The paper on Scala of these authors [5] presents an evaluation grid of nine criteria. Letting out the last three (which deal with concrete performance measuring and cannot easily be reconstructed) we come the result that Paisley corresponds to the "extractor" solution presented in this paper, with some significant differences:

Conciseness of the framework Overhead is required for the Paisleys projection operations, see section 3.8 above. They correspond to the "extractors" in Scala, and are a little more verbose than those, caused by the host language's

syntax. Furthermore, the chain of delegation to embedded patterns must always be written down explicitly, while in many situations a call to "unapply()" will silently be inserted in the Scala approach.

But many of these projections are pre-defined in the more special libraries of Paisley, most of them parametric, and ready-to-reuse by the programmer.

Conciseness of shallow matches. Conciseness of deep matches
The syntax of a concrete application of a complex Paisley pattern has least possible snytactic noise.

But some noise is indeed generated by the fact that the "right hand side" is not simply written in-line, but has to be coded explicitly, after the match has succeeded, maybe involving explicit unpacking of bound variables.

Maintainability: representation independence
No internal representation at all must be revealed, because only the functional interfaces must be implemented. (Nevertheless, in most cases the extractors will follow the internal structure "voluntarily".)

Maintainability: extending (data) variants. Maintainability: extending patterns
The data and the pattern world may grow arbitrarily without affecting the behaviour of older class definitions and patterns. Since patterns are only defined by their functional interface, arbitrary new variants can be added, and existing combinations can freely be abstracted at any time.

This goes far further than the Scala "extractors": Since being transparent to non-determinisum, also arbitrary disjunctions and even encapsulated search can be abstracted to one single pattern component.

References

1. Becchi, M., Crowley, P.: Extending finite automata to efficiently match perl-compatible regular expressions. In: Proceedings of the 2008 ACM CoNEXT Conference. pp. 25:1–25:12. CoNEXT '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1544012.1544037
2. Benedikt, M., Fan, W., Geerts, F.: XPath satisfiability in the presence of DTDs. J. ACM 55(2), 1–79 (2008)
3. Blomer, J., Geiß, R., Jakumeit, E.: The GrGen.NET User Manual (2011), http://www.grgen.net
4. Ebert, J., Bildhauer, D.: Reverse engineering using graph queries. In: Schürr, A., Lewerentz, C., Engels, G., Schäfer, W., Westfechtel, B. (eds.) Graph Transformations and Model Driven Engineering, vol. 5765. Springer Verlag (2010)
5. Emir, B., Odersky, M., Williams, J.: Matching objects with patterns 4609 (2007)
6. Hors, A.L., Hégaret, P.L., Wood, L., Nicol, G., Robie, J., Champion, M., Byrne, S.: Document Object Model (DOM) Level 2 Core Specification Version 1.0. W3C Recommendation, http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/
7. Lepper, M., Trancón y Widemann, B.: Optimization of visitor performance by reflection-based analysis. In: Cabot, J., Visser, E. (eds.) Theory and Practice of Model Transformations. Lecture Notes in Computer Science, vol. 6707. Springer (2011)

8. Liu, J., Myers, A.C.: JMatch: Iterable abstract pattern matching for Java. In: Practical Aspects of Declarative Languages. Lecture Notes in Computer Science, vol. 2562. Springer (2003)
9. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. artima, 2 edn. (2010)
10. Trancón y Widemann, B., Lepper, M.: The BandM Meta-Tools JAVA API Doc. `http://bandm.eu/metatools/docs/api/index.html` (2010)
11. Trancón y Widemann, B., Lepper, M.: Paisley: Pattern matching à la carte. LNCS, vol. 7307. Springer (2012)

```
static Pattern<Calendar> foolsDay1(Pattern<? super Integer> year) {
  return new Unary<Integer, Calendar>(year) {
    public boolean match(Calendar c) {
      return c.get(Calendar.MONTH)        == Calendar.APRIL
        &&   c.get(Calendar.DAY_OF_MONTH) == 1
        && getBody().match(c.get(Calendar.YEAR));
    }
  };
}

// --------------------------------------------------------------

static Pattern<Calendar> get(final int field,
                             Pattern<? super Integer> value) {
  return new Transform<Calendar, Integer>(value) {
    public Integer apply(Calendar c) {
      return c.get(field);
    }
  };
}

static Pattern<Calendar> foolsDay2(Pattern<? super Integer> year) {
  return all(get(Calendar.MONTH,        eq(Calendar.APRIL)),
             get(Calendar.DAY_OF_MONTH, eq(1)),
             get(Calendar.YEAR,         year));
}

// --------------------------------------------------------------

static Pattern<Calendar> day(Pattern<? super Integer> day) {
  return get(Calendar.DAY_OF_MONTH, day);
}

static Pattern<Calendar> month(Pattern<? super Integer> month) {
  return get(Calendar.MONTH, month);
}

static Pattern<Calendar> year(Pattern<? super Integer> year) {
  return get(Calendar.YEAR, year);
}

static Pattern<Calendar> foolsDay3(Pattern<? super Integer> year) {
  return all(month(eq(Calendar.APRIL)),
             day  (eq(1)),
             year (year));
}

// --------------------------------------------------------------

private static final double MSEC_PER_YEAR
  = 365.0 * 24.0 * 60.0 * 60.0 * 1000.0;
private static final double INCREMENT = 0.123456789;

final static Calendar[] x = new Calendar[5000];

private static void initTestdata(){
  for (int i = 0; i < x.length; i++) {
    x[i] = Calendar.getInstance();
    x[i].setTimeInMillis((long)(i * INCREMENT * MSEC_PER_YEAR));
  }
}
```

Fig. 3. Example 1: Matching all Fools' Days

```
import org.w3c.dom ;

static Pattern<Node>    ancestor(Pattern<? super Node> body)
static Pattern<Node>    ancestorOrSelf(Pattern<? super Node> body)
static Pattern<Element>         attr(String name)
static Pattern<Element>         attr(String namespaceURI, String localName)
static Pattern<Element>         attrNode(String name, Pattern<? super Attr> body)
static Pattern<Element>         attrNode(String namespaceURI,
                                         String localName, Pattern<? super Attr> body)
static Pattern<Element>         attrValue(String name, Pattern<? super String> body)
static Pattern<Element>         attrValue(String namespaceURI, String localName,
                                          Pattern<? super String> body)
static Pattern<Node>    child(Pattern<? super Node> body)
static Pattern<Node>    comment()
static Pattern<Node>    descendant(Pattern<? super Node> body)
static Pattern<Node>    descendantOrSelf(Pattern<? super Node> body)
static Pattern<Node>    element(Pattern<? super Element> body)
static Pattern<Node>    followingSibling(Pattern<? super Node> body)
static Pattern<Node>    localName(String localName)
static Pattern<Node>    name(String namespaceURI, String localName)
static Pattern<Node>    namespaceURI(String namespaceURI)
static Pattern<Node>    nextSibling(Pattern<? super Node> body)
static Pattern<Node>    parent(Pattern<? super Node> body)
static Pattern<Node>    precedingSibling(Pattern<? super Node> body)
static Pattern<Node>    previousSibling(Pattern<? super Node> body)
static Pattern<Document>        root(Pattern<? super Element> body)
static Pattern<Element>         tagName(String tagName)
static Pattern<Node>    textContent(Pattern<? super String> body)
```

Fig. 4. The W3C XML DOM Pattern Library

```
private static final String XHTMLNS = "http://www.w3.org/1999/xhtml";

/** Matches Nodes which are Elements from the xhtml namespace with
    the given local name, and then match the subPattern.
 */
private static Pattern<Node> xhtml(String localName,
                                   Pattern<? super Element> subPattern) {
  return element(both(name(XHTMLNS, localName), subPattern));
}

private static Pattern<Node> defTerm(Pattern<? super Element> dtPattern) {
  return xhtml("dt", dtPattern);
}
private static Pattern<Node> defDescription(Pattern<? super Element> ddPattern) {
  return xhtml("dd", ddPattern);
}

/** Matches an "xhtml:dt" element immediately followed by an
   "xhtml:dd" element, iff both match the resp. pattern argument.
 */
private static Pattern<Node> defPair(Pattern<? super Element> dtPattern,
                                     Pattern<? super Element> ddPattern) {
  return both(defTerm(dtPattern), nextSibling(defDescription(ddPattern)));
}

/** Matches an "xhtml:a" element iff its href attribute matches the String pattern.
 */
private static Pattern<Node> anchorHRef(Pattern<? super String> href) {
  return xhtml("a", attrValue("href", href));
}

/** Matches a String iff it is/is not a "local" uri
 */
private static Pattern<String> localURL(final boolean local) {
  return new Atomic<String>() {
    public boolean match(String x) {
      return x.startsWith("#") == local;
    }
    @Override public String toString() {
      return "localURL(" + local + ")";
    }
  };
}

/** Aux function: returns whether a given text contains a given keyword,
    possibly after stripping some framing brackets.
 */
private static boolean textContainsDefTerm (final String text,
                                            final String defWord){
  final String keyword = defWord.startsWith("[")
    ? defWord.substring(1,defWord.length()-1) : defWord ;
  return text.toUpperCase().contains(keyword.toUpperCase());
}
```

Fig. 5. Example 2: Matching the XHTML XHTML specificaton

```
"This version:" refers to "http://www.w3.org/TR/2002/REC-xhtml1-20020801"
"Latest version:" refers to "http://www.w3.org/TR/xhtml1"
"Previous version:" refers to "http://www.w3.org/TR/2000/REC-xhtml1-20000126"
"Diff-marked version:" refers to "xhtml1-diff.html"
"Well-formed" refers to "http://www.w3.org/TR/REC-xml#sec-well-formed"
"[CSS2]" refers to "http://www.w3.org/TR/1998/REC-CSS2-19980512"  ★
"[CSS2]" refers to "http://www.w3.org/TR/REC-CSS2"  ★
"[DOM]" refers to "http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001"  ★
"[DOM]" refers to "http://www.w3.org/TR/REC-DOM-Level-1"  ★
"[DOM2]" refers to "http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113"
"[DOM2]" refers to "http://www.w3.org/TR/DOM-Level-2-Core"
"[HTML]" refers to "http://www.w3.org/TR/1999/REC-html401-19991224"  ★
"[HTML]" refers to "http://www.w3.org/TR/html401"  ★
"[RFC2045]" refers to "http://www.ietf.org/rfc/rfc2045.txt"  ★
"[RFC2046]" refers to "http://www.ietf.org/rfc/rfc2046.txt"  ★
"[RFC2046]" refers to "http://www.ietf.org/rfc/rfc2046.txt"  ★
"[RFC2119]" refers to "http://www.ietf.org/rfc/rfc2119.txt"  ★
"[RFC2376]" refers to "http://www.ietf.org/rfc/rfc2376.txt"  ★
"[RFC2396]" refers to "http://www.ietf.org/rfc/rfc2396.txt"  ★
"[RFC2854]" refers to "http://www.ietf.org/rfc/rfc2854.txt"  ★
"[RFC3023]" refers to "http://www.ietf.org/rfc/rfc3023.txt"  ★
"[RFC3066]" refers to "http://www.ietf.org/rfc/rfc3066.txt"  ★
"[RFC3236]" refers to "http://www.ietf.org/rfc/rfc3236.txt"  ★
"[XHTML+MathML]" refers to "http://www.w3.org/TR/MathML2/dtd/xhtml-math11-f.dtd"
"[XHTMLMIME]" refers to "http://www.w3.org/TR/2002/NOTE-xhtml-media-types-20020801"
"[XHTMLMIME]" refers to "http://www.w3.org/TR/xhtml-media-types"
"[XHTMLMOD]" refers to "http://www.w3.org/TR/2001/REC-xhtml-modularization-20010410"
"[XHTMLMOD]" refers to "http://www.w3.org/TR/xhtml-modularization"
"[XML]" refers to "http://www.w3.org/TR/2000/REC-xml-20001006"  ★
"[XML]" refers to "http://www.w3.org/TR/REC-xml"  ★
"[XMLNS]" refers to "http://www.w3.org/TR/1999/REC-xml-names-19990114"
"[XMLNS]" refers to "http://www.w3.org/TR/REC-xml-names"
"[XMLC14N]" refers to "http://www.w3.org/TR/2001/REC-xml-c14n-20010315"
"[XMLC14N]" refers to "http://www.w3.org/TR/xml-c14n"
```

Fig. 6. The results of Matching the XHTML XHTML specificaton.