Recursive types and pattern-matching in Java

Rémi Forax & Gilles Roussel
`http://vodka.univ-mlv.fr/~forax`
`http://monge.univ-mlv.fr/~roussel`

Institut Gaspard Monge,
Université de Marne-la-Vallée

Abstract. *Recursive types definitions* and *pattern-matching* are two useful built-in features of functional languages. No such mechanism is present in the Java language. In this article, we investigate different implementations to support these features in Java. First, we review ways to define recursive types. Then, we expose several approaches allowing to simulate pattern-matching on structures of these types. Finally, we present re-use techniques for algorithms featuring this mechanism.

1 Introduction

Describing algorithms on recursive structures is among the fundamental concepts of computer science. Strongly typed functional languages propose type-constructors and pattern-matching mechanisms which make the expression of such algorithms easier. These constructions are not classical idioms of object-oriented languages. As a matter of fact, these notions are often opposed to the object notion, but we assert that they are complementary [12, 1].
In this article, we review different implementations to support these features in Java.
First, we study different implementations for recursive type definitions. These types allow to type-check algorithms over recursive data structures.
Then, we examine different techniques to specify algorithms over recursive structures, just as if pattern-matching was featured in the language. More or less, these solutions simulate CLOS multi-methods [2] or Dylan functions [8] late binding according to method arguments types.
Finally, we study the specification of algorithms, loosely coupled with the data structures, in order to ease their re-use.
This work brought us to propose :

- a definition for recursive types based on inner interfaces.
- a `PatternMatcher` generic component to implement late binding according to method argument types.
- two main techniques to implement loosely coupled algorithms:
  - one based on interface adapter
  - another based on default traversal mechanism whose most general implementation is the new *Codec Pattern*

In section 2, we study a classical object-oriented way to implement recursive types and algorithms on recursive data structures. In section 3, we present an implementation of recursive types based on inner interfaces and propose methods to simulate pattern-matching. In section 4, we use previous results to implement loosely coupled algorithms.

2 Concrete recursive types

A classical way [12] to represent recursive types in a Java like object-oriented language is drawn from the functional languages type constructors. The type is implemented by a *base class* `A`, and each constructor of the type is represented by a *constructor class* `B` that inherits from the base class `A`.
Thanks to the subtyping mechanism, the reference to objects of constructor classes (`B`) can be used wherever a `A`-typed reference is expected.
In order to illustrate this method we will use a simple example of recursive type: a chained list of integers. In typed functional languages, the definition of the recursive type `List` is something like:

```
List int = Nil | Cons of int * List int;
```

which means: a list of integers is either, an empty list of name `Nil` or a pair of name `Cons` including an integer and a list of integers.
In an object-oriented language, the product '`*`' can be implemented by fields in the constructor classes and the alternative '`|`' by the inheritance mechanism, with several sub-classes which inherit from a single class. Therefore, the type `List` can be represented in Java by the three following classes.

```
abstract class List { }
class Nil extends List { }
class Cons extends List {
  int value;
  List tail;
  public Cons(int value,List tail) {
    this.value=value;
    this.tail=tail;
  }
}
```

The class `List` is the base class. Variables of this type can contain references to instances of the `Nil` and `Cons` constructor classes.
Nevertheless, this interpretation of functional type constructors in Java allows the type to be extended by adding new classes inheriting from (or implementing) base class `List`. For instance, one could add a `ConsDouble` class containing a `double` instead of an `int` and inheriting form `List`. Then, a variable of type `List` could contain a reference to an heterogeneous list. It constitutes a major difference with the functional approach where the type is homogeneous and not extensible [5].

2.1 Algorithms on recursive structures

Now that we have seen how recursive types are classically described with classes, we study how to specify algorithms on recursive structures (instances of these classes).
A first approach consists in declaring an abstract method (algorithm) in the base class and implements in each constructor sub-class its proper behavior.
Using this approach, let us consider the implementation of a function `sum` computing the sum of integers contained in a list. The base class `List` will now specify an abstract `sum()` method which will be implemented in each constructor class.

```
abstract class List {
  public abstract int sum();
}
class Nil extends List {
  public final int sum() {
    return 0;
  }
}
class Cons extends List {
  int value;
  List tail;
  public Cons(int value,List tail) {
    this.value=value;
    this.tail=tail;
  }
  public final int sum() {
    return value+tail.sum();
  }
}
```

With this method, it is easy to add (resp. remove) other constructor classes: the parts of the algorithm that are specific to each constructor are independent from each other.
On the other hand, this implementation is not exactly suitable for adding (resp. removing) a new algorithm within this structure, as it would be then necessary to alter each class. This forbids separate compilation of each different algorithm.

3 Abstract recursive types

In the Java language, when a class is declared two features are mixed: a type declaration and a object "skeleton" definition. The declared type is not the type of objects of that class but the one of references to object of that class. This type determines the members (fields or methods) that can be accessed through a reference of that type. In other words, it defines an interface[1].
This is why we argue that recursive types should not be implemented as classes but rather than a set of Java `interface`. Since a Java `interface` does not allow non-constant field declaration, type fields are represented by methods[2].
In this approach, the list type is consists in three interfaces: a *base interface* and two *constructor interfaces*. We chose to define the interfaces `Nil` and `Cons` as inner-interface of the `List` one. In one hand,this encapsulates the type definition into a unique interface and in the other hand this constrains names of list sub-classes. Classes implementing that type must implement the `List.Nil` or the `List.Cons` interface.

```
interface List {
  interface Nil extends List { }
  interface Cons extends List {
    public abstract int getValue();
    public abstract List getTail();
  }
}
```

Moreover constructor interfaces of a recursive type could be found by reflection [10] with the method `getDeclaredClasses()` of class `Class`. Then it is possible in some implementation to verify that the list type has not been extended by `List` sub-interfaces.
It is obvious to create concrete classes, implementing these interfaces (as in section 2).
One should notice that the use of access methods relaxes the type implementation: methods can either be implemented by static fields access or by dynamic computations of the field. This allow to reuse methods working on recursive types, with a large set of recursive structures as argument. For example, the argument of a method `void m(List)` can either be a classical list of integers or a lazily defined list of all the integers, as defined below.

```
class LazyCons implements List.Cons {
  List tail;
  int value;
  public LazyCons(int value) {
    this.value=value;
  }
  public List getTail() {
    if (tail==null)
      tail=new LazyCons(value+1);
    return tail;
  }
  public int getValue() {
    return value;
  }
}
```

[1] Here, the term interface is used in a more general sense than Java keyword `interface`.
[2] In the `List` interface, we only define access methods (`get*`), the modification methods (`set*`) could be added if needed.

More generally, developers should never use classes to define types but always define the corresponding interfaces to ease reusability. Unfortunately, the use of interfaces slows down execution and multiplies the number of files to maintain. Therefore, an automated way to move from one specification to the other should be investigated (see section 4.1).

3.1 Algorithms on abstract recursive structures

Inheritance is not well suited to specify algorithms on abstract recursive types. For instance, the `sum()` method cannot be simply added to the previous `List` interface using inheritance because, in the `Cons` class, this method is called on the `getTail()` return value which is of a static type `List`. It is therefore necessary to modify the `List` interface or to use type cast and multiple implementation of interfaces as proposed below.

```
interface SumList extends List {
  public abstract int sum();
}
abstract class Cons implements List.Cons, SumList {
  public final int sum() {
    // return getValue()+getTail().sum();  type checking error!!
    return getValue()+((SumList)getTail()).sum();
  }
}
abstract class Nil implements List.Nil, SumList {
  public final int sum() {
    return 0;
  }
}
```

The cast is needed because since the Java language does not provide a way to specify contravariant return type for the method `getTail()`. In the above example, a `SumList getTail()` method cannot overwrite the `List getTail()` method defined in the `List` class.

3.2 Functional approach

In this section we investigate an alternative to inheritance that we call the functional approach. It saves the developpers to modify multiple classes to specify a new algorithm since the whole algorithm is encapsulated within a single class. We no longer view algorithms as methods of objects but rather as objects. Since algorithms studied in this article share a common behavior – i.e. they do pattern-matching – objects representing them should inherit from a common `PatternMatcher` class implementing the pattern-matching.
To maintain the modularity, the algorithm is broken down along each interface constituting the type (`List`). The algorithm part associated to each constructor interface is specified within a method `match()` taking this constructor interface as parameter. For the `List` type, this class should look like:

```
class PatternMatcher {
  public final Object match(List.Nil n)  { ... }
  public final Object match(List.Cons c) { ... }
}
```

In order to simulate the pattern-matching, a mechanism must be available to search for the appropriate `match()` method, according to the actual argument type, like multi-methods do [2]. A late binding of method has to be performed according to the type of the first argument. The idea is to write the `match(List.Cons)` and `match(List.Nil)` methods and to be able to call the `match()` method with a reference statically typed as `List`; the choice of the method being dynamic.
However, it cannot be done in the Java language because the method binding is established from the statically determined types of the argument. Late binding of methods is only performed according to the type of the target object[3].
Now we will describe two different ways to simulate this late binding.

[3] We call this late binding: the *classical* late binding.

3.3 Visitor pattern

A first general solution to solve this problem is known as the *Visitor Pattern* [6]. This pattern is very close to the functional approach and we present it on the list example.
First, the recursive type must be defined. Each of its classes must implement an `void accept(Visitor)` method. Thus, the type list can no longer be specified using interfaces. A way to specify algorithms over interfaces is proposed in a following section.

```
abstract class List {
  public void accept(ListVisitor v);
}
class Nil extends List {
  public void accept(ListVisitor v) {
    v.visit(this);
  }
}
class Cons extends List {
  int value;
  List tail;
  public Cons(int value,List tail) {
    this.value=value;
    this.tail=tail;
  }
  public void accept(ListVisitor v) {
    v.visit(this);
  }
}
```

Then, the `sum` algorithm on the list is now specified within a single `SumVisitor` class implementing the interface `ListVisitor`. For each `List` sub-class, there is a method `visit()`, i.e. `visit(Cons)` and `visit(Nil)`.

```
interface ListVisitor {
  public void visit(Cons c);
  public void visit(Nil n);
}
class SumVisitor implements ListVisitor {
  int sum=0;
  public final void visit(Nil n) { }
  public final void visit(Cons c) {
    sum+=c.value;
    c.tail.accept(this);
  }
}
```

In order to invoke the appropriate method according to the dynamic type of the first argument, the classical Java late binding mechanism (according to the target object) is used. More precisely, method overwriting, here of the `accept(ListVisitor)` method, is used. When `target.accept(v)` is executed, the virtual machine selects dynamically the appropriate `accept()` method according to the target object type (standard late binding). As each `accept()` method statically knows the object class in which it is defined, it can call the appropriate `visit()` method by taking `this` (statically typed) as its argument.
This exchange between target object and argument object is not very natural. It forces the user to call the `accept()` method on the list, rather than the `visit()` method as shown in the next example.

```
  public final static void main(String[] argv) {
    List l=new Cons(2,new Cons(3,new Nil()));
    SumVisitor visitor=new SumVisitor();
```

```
    l.accept(visitor);
    System.out.println("sum :"+visitor.sum);
  }
```

3.4 A reflection mechanism

In the previous approach, the method overwriting of `accept()` allows to associate a `visit()` method corresponding to the actual type of an object. We introduce here another construction which saves the need for implementing `accept()` methods in recursive types.
A similar approach is described in the article *The Essence of the Visitor Pattern* [14] although in that case it was limited to one argument and mixed with a default traversal feature (see in section 4.2).
The main idea of the construction we propose is to use a `match(List)` method which dynamically invokes the appropriate `match()` method, according to the dynamic type of its argument thanks to the standard reflection mechanism of Java. The `getClass()` method of `java.lang.Class` allows to determine the actual type of the argument. Then the `getMethod()` method finds the appropriate `match()` method. Eventually, the resulting `match()` method is invoked with the `invoke()` method.
As a matter of fact, we adopt a general method where late binding can be performed on several arguments (passed as an object array). The key part of this approach is the `PatternMatcher` class. It is a generic component which performs the pattern-matching whatever is the parameter type used.
The inheritance of this component ease the specification of algorithms that use pattern-matching on a recursive structure. Unlike *Visitor Pattern*, this method does not make mandatory, for each type, the writing of an interface containing a method for each constructor class.
The following code is a naive implementation for the generic `PatternMatcher` component. A more generic class could take as argument the name of the matching method and the `Class` instance in which the method should be looked for.

```
class PatternMatcher {
 public final Object match(Object[] l) {
   Class lClass[]=new Class[l.length];
   for(int i=0;i<l.length;i++)
     lClass[i]=l[i].getClass();
   return match(l,lClass);
 }
 public final Object match(Object[] l,Class[] c) {
   try {
     Method m=getClass().getMethod("match",c);
     return m.invoke(this,l);
   }
   catch(Exception e){
     throw new NoMatchingMethodException(e.getMessage());
   }
 }
}
```

In this class, we have defined two `match()` methods. The first one identifies argument types with argument classes. In the general case this is not true since arguments of primitive type are wrapped into objects. Thus, the second method provide a way to pass arguments as objects together with their type.
To illustrate the use of this generic component, we present the `Merger` algorithm. It performs a pattern-matching on its two arguments in order to merge two lists alternately. Specifying the algorithm computing the sum of the list integers would have been as simple.

```
class Merger extends PatternMatcher {
```

```
  public final List match(Nil n,Cons c) {
    return c;
  }
  public final List match(Nil n1,Nil n2) {
    return n1;
  }
  public final List match(Cons c,Nil n) {
    return c;
  }
  public final List match(Cons c1,Cons c2) {
    List tail=c1.getTail();
    c1.setTail(c2);
    c2.setTail(match(tail,c2.getTail()));
    return c;
  }
  public final List match(List l1,List l2) {
    return (List)match(new Object[]{l1,l2});
  }
}
```

The last `match()` method specializes the generic `PatternMatcher` for the specific type `List`.

3.5 Algorithm optimization

The main problem with this implementation, also observed by *Palsberg & Jay*, is its lack of speed. In fact, for each class, the algorithm has to process a method search. One possible optimization consists in calculating in advance the set of classes for which a `visit()` method is available and save this information in an hash-table.
The benchmark chosen to validate this implementation, is the one described in the article *The Essence of the Visitor Pattern* [14]. The aim is to calculate the sum of the list integers. The list is composed of 2000 links. The test was made on a 64 Megabytes SPARCstation 10. The compiler and the run-time engine were taken from the jdk1.1.6.

Implementation	processing time
redefining method	0.02 min
Visitor Pattern	0.03 min
`PaternMatcher`	1.18 min
Optimized `PaternMatcher`	0.15 min

Optimizing the `PaternMatcher` implementation allows to significantly boost its speed. We estimate that more thorough optimizations should lower its execution time to that of the *Visitor Pattern* since optimizations performed to boost the classical late binding [11] could be applied for parameter late binding. As a corollary, the reflection mechanism implemented in the *JVM* could be improved if including these optimizations.

3.6 Extending the reflection search method

The dynamic method search mechanism searches for the methods according to its exact profile. In the previous example, the `PatternMatcher` performs the method search according to the concrete class rather than its interface. That is to say that the Java dynamic reflection method search does not perform dynamically type unification as it is done at compile time for method matching.
In fact, the `PatternMatcher` and the *Visitor Pattern* do not follow the same semantics. The `PatternMatcher` implementation does not work with a recursive type defined using interfaces as advocated.
We propose a new implementation of the `java.lang.Class.getMethod()` which performs a dynamic type unification. We think this method should be included in Java reflection API.

```
public Method getBestMethod(String name,Class types[])
```

```
    throws NoSuchMethodException, SecurityException {
    Method methods[]=getMatchMethods(name,types);
    Method bestMethod=methods[0];
    Class bestMethodTypes[]=bestMethod.getParameterTypes();
    for(int i=1;i<methods.length;i++) {
      int j=0;
      int value=0;
      Class pTypes[]=methods[i].getParameterTypes();
      for(;j<pTypes.length;j++) {
        int diff=(bestMethodTypes[j].isAssignableFrom(pTypes[j]))?1:
          (pTypes[j].isAssignableFrom(bestMethodTypes[j]))?-1:0;
        if (diff==0 || (value!=diff && value!=0))
          throw new TwoMethodMatchException(
            "two methods "+name+" match parameters");
        value=diff;
      }
      if (value==1) {
        bestMethod=methods[i];
        bestMethodTypes=pTypes;
      }
    }
    return bestMethod;
  }

  private Method[] getMatchMethods(String name,Class types[])
    throws NoSuchMethodException, SecurityException {
    Vector vector=new Vector();
    Method methods[]=getMethods();
    for(int i=0;i<methods.length;i++)
    if (methods[i].getName().equals(name)) {
      Class pTypes[]=methods[i].getParameterTypes();
      if (types.length==pTypes.length) {
        int j=0;
        for(;j<types.length && pTypes[j].isAssignableFrom(types[j]);j++);
        if (j==types.length)
          vector.add(methods[i]);
      }
    }
    if (vector.size()==0)
      throw new NoSuchMethodException("method "+name+" not found");
    return (Method[])vector.toArray(new Method[vector.size()]);
  }
```

The `getMatchMethods()` method constructs the array of methods well typed according to arguments. Then, the `getBestMethod()` method looks for the most precise method. It performs this search dynamically with the same algorithm used by Java compilers statically. Moreover, the Java compilers use argument static types whereas the `getBestMethod()` method uses their real dynamic types to allow late binding.
In the ambiguous cases the `getBestMethod()` method throws a `TwoMethodMatchException`. A simpler but less general method search, used by parasitic multi-methods [13], consists in choosing the first matching method.
If no method matches argument types a `NoSuchMethodException` is thrown by the `getMatchMethods()`. This feature is not desirable in a strongly typed language. Implementing a default behavior [14] (see. section 4) solves this problem.

Default parameter

This version allows the `List` base interface to be used as a parameter type. It behaves as a default parameter type (the functional '_'). Writing a `Merger` becomes even simpler since it is not necessary to write a method `match()` for each combination of constructor interfaces.

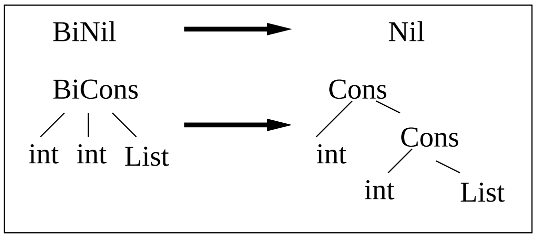
```
class Merger extends PatternMatcher {
  public final List match(Nil n, List l) {
    return l;
  }
  public final List match(Cons c,Nil n) {
    return c;
  }
  public final List match(Cons c,Cons c2) {
    return new Cons(c.getValue(),new Cons(c2.getValue(),
      match(c.getTail(),c2.getTail())));
  }
  public final List match(List l1,List l2) {
    return (List)match(new Object[]{l1,l2});
  }
}
```

4 Loosely coupled algorithms

The aim of this section is to investigate several techniques allowing to describe a "translation" between two similar, but non isomorphic, recursive types, so that algorithms specified for one type can be applied on the other one. Then with this approach, algorithms can be specified on their natural recursive type and reuse everywhere they make sense.

4.1 Interface adapter

In section 2 we saw that using interfaces allows the structure to abstract from its internal representation. Moreover, they also allow to specify loosely coupling of algorithms. To do so, an adapter class performs the "type translation". We believe that implementation of such adapter should use the `PatternMatcher` generic component, since it needs to perform pattern-matching on the input recursive structure.
To illustrate this technique, we present an example where objects of class `BiList` are considered as objects of class `List`. A `BiList` object is a list where each link contains two values.



The idea of our implementation is to (lazily) construct the `List`, step by step, using a pattern-matching over the `BiList`. A `BiCons(a,b,tail)` is translated into `Cons1(a,Cons2(b,tail))` and `BiNil` into `Nil`. Classes `Cons1` and `Cons2` implements `List.Cons`, `Nil` implements `List.Nil`. To reuse a algorithm specified for a `List` one should call:
    `Algo.match(BiListToListAdapter.match(biList))`

```
class BiListToListAdapter extends PatternMatcher {
  public final List match(BiList l) {
   return (List)match(new Object[]{l});
```

```
  }
  public final List match(BiList.BiCons bicons) {
    return new Cons1(bicons);
  }
  public final List match(BiList.BiNil binil) {
    return new Nil();
  }
  static class Nil implements List.Nil { }

  class Cons1 implements List.Cons {
    public Cons1(BiList.BiCons bc) {
      this.bc=bc;
    }
    public final int getValue() {
      return bc.getValue1();
    }
    public final List getTail() {
      if (cons2==null)
        cons2=new Cons2(bc);
      return cons2;
    }
    Cons2 cons2;
    BiList.BiCons bc;
  }

  class Cons2 implements List.Cons {
    public Cons2(BiList.BiCons bc) {
      this.bc=bc;
    }
    public final int getValue() {
      return bc.getValue2();
    }
    public final List getTail() {
      if (list==null)
        list=match(bc.getTail());
      return list;
    }
    List list;
    BiList.BiCons bc;
  }
}
```

It is not easy to implement such adapters by hand. We are investigating the automatic generation of adapter classes inspired from other languages approaches [7, 4]. The common idea of these techniques is to deduce, from type definitions, a mapping between types.
Moreover, the use of an adapter generates a intermediate data structure that could be discard. We hope to adapt functional deforestation methods [3, 15] to statically eliminate the adapter implementation.

4.2 Default traversal

Another approach, introduced by Lieberherr [9], as *adaptive programming* introduced the notion of default traversals on every structure subtypes which do not have a specified visit method. The writing is therefore more independent from the algorithm and the structure, if the structure is modified, the description of the algorithm remains the same.
The version proposed by Lieberherr is static. All classes are available when traversals are defined. Here, we study two methods which use the Java reflection mechanism to introduce a dynamic default traversal.

4.3 Walkabout pattern

A first method proposed by *J. Palsberg & C.B. Jay* for defining this default traversal i.e. unspecified `match()`, consists in executing a visit in all the fields of non-primitive type of the class.
In the chained list example, only the `Cons` class has to contain a visit method because the traversal of `Nil` is the default one. In fact, the *Walkabout Pattern* will not visit the `Nil` since it does not contain fields.

```
class Sum extends Walkabout {
  int sum=0;
  public final void match(Cons c) {
    sum+=c.getValue();
    match(c.getTail());
  }
}
```

```
class Walkabout extends PatternMatcher {
  public final void match(Object l) {
    if (l!=null) {
      try {
        match(new Object[]{l});
      }
      catch(NoMatchingMethodException e) {
        Field fields[]=l.getClass().getFields();
        for(int i=0;i<fields.length;i++)
          if (!(fields[i].getType().isPrimitive()))
            try {
              match(fields[i].get(l));
            }
            catch(IllegalAccessException e2) {
            }
      }
    }
  }
}
```

4.4 Codec pattern

The *Walkabout* takes advantages of the `PatternMatcher` dynamic method search according to the classes and interfaces of the hierarchy. A natural improvement of the *Walkabout Pattern* consists in using the inheritance tree, not only for dynamic method search, but also to specify the traversal on the structure.
The basic idea of the *Codec Pattern*, is to enable the specification of algorithm fractions, not only for terminal (final) classes of the inheritance hierarchy, but also for all internal classes of the hierarchy. This implies to extend the *Walkabout Pattern* default traversal with a traversal of the inheritance hierarchy.
The traversal of the inheritance tree visits each class twice: once in the direction of the inheritance, and once in the opposite direction. Two *wrappers* (`match()`methods) can be specified for each class; `prefix()` is called when going down in the tree and `suffix()` is called when going up. These wrappers overwrite `prefix()` and `suffix()` default methods. The default `prefix()` method does nothing. The `suffix()` default method traverses fields declared (not inherited) in the class like *Walkabout Pattern*[4].
The implementation of the *Codec Pattern* is much more complex and space consumer than the others, this is why we only present an example using this component.
To illustrate the use of this component, we slightly modify the implementation of the list by inserting an abstract class `Item` between the interface `List` and the `Cons` links in order to add a `Link` link.

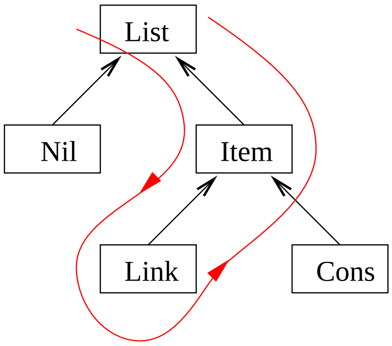


Fig. 1. Method call flow

The figure 1 shows, for a `Link` object, the call flow of wrappers in the *Codec Pattern*.
The above example describes a *PrintVisitor* class which traverses a list and displays the content of each link by using the `Codec` component. The example `RevPrintVisitor` produces the reversed list in relation to the `PrintVisitor`.

```
class PrintVisitor extends Codec {
  public final void prefix(Cons c) {
    System.err.println(
      "Cons.value "+c.getValue());
  }
  public final void prefix(Link l) {
    System.err.println(
      "Link.color "+l.getColor());
  }
  public final void prefix(Nil n) {
    System.err.println("Nil");
  }
  // public final void suffix(Item i) {
  //   visit(i.list);
  // }
}
```

```
class RevPrintVisitor extends Codec {
  public final void suffix(Cons c) {
    System.err.println(
      "Cons.value "+c.getValue());
  }
  public final void suffix(Link l) {
    System.err.println(
      "Link.color "+l.getColor());
  }
  public final void suffix(Nil n) {
    System.err.println("Nil");
  }
  public final void prefix(Item i) {
    visit(i.getTail());
  }
}
```

[4] It has to be noted that the fields traversal order is not, *a priori*, the same than the *Walkabout Pattern*.

The traversal executed by the *Codec Pattern* is quite close to the one executed by the *tree-traversal* of Lieberherr [9]. The propagation during the visit is executed in a similar way. However, the *Codec Pattern* stays blind as for the direction it has to take, regarding the *tree-traversal* which benefits a static analysis of the class graph.
Our actual implementation turns out to be much more complex than the simple version of the `PatternMatcher` and its performances regarding size and speed are unattractive. We estimate that thorough optimizations shall allow to obtain acceptable performances.

5 Conclusions

In this article, we propose several implementations, featuring a reflection mechanism, which ease the use of recursive types in Java. These implementations cover recursive type definition, pattern-matching and re-use of algorithms by mimicking features of typed functional languages.
The main difference between these two approaches (functional and object-oriented) remains the check of type. In Java, type-check is performed dynamically and may throw exceptions at runtime, which can be qualified of "heretic" for strongly typed languages. This loosely type-check is issued from the intensive use of the Java reflection mechanism.
In addition to the technical aspect of implementing pattern-matching in Java, this article shows that reflection allows to define new mechanisms; in that case, the late binding of the method according to the argument types, without altering the language. This track is quite interesting, as it could lead to a language whose base mechanisms would be extremely poor (do not contain idiomatic constructions) and easy to master, and where programmers would add in a very simple way other mechanisms according to their needs, thanks to generic components by using the reflection mechanism.

Acknowledgments

We would like to thank Didier Parigot and Loic Correnson for our discussions, and also Etienne Duris and Marc Dalesme for their help regarding the writing of this article.

References

1. Luca Cardelli, Peter Wegner *On Understanding Types, Data Abstraction, and Polymorphism* Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985.
2. L. G. DeMichiel, R. P. Gabriel. *Common lisp object system overview* ECOOP'87, LNCS 276 pp 151-170, 1987.
3. Philip Wadler. *Deforestation: Transforming Programs to Eliminate Trees* In Harald Ganzinger, editor, European Symposium on Programming (ESOP '88), LNCS 300 , pages 344-358, Nancy, March 1988. Springer-Verlag.
4. Carole Le Bellec, Martin Jourdan, Didier Parigot and Gilles Roussel. *Specification and Implementation of Grammar Coupling Using Attribute Grammars* In Proceedings of Programming Language Implementation and Logic Programming (PLILP '93), LNCS 714, pages 123-136, Tallinn, August 1993.
5. Pascal Andr et Jean-Claude Royer. *La modlisation des listes en programmation par objets.* Dans JFLA'94, P. Cointe, C. Queinnec et B. Serpette (diteurs), Collection Didactique de l'INRIA, pages 259-285, Noirmoutier, Janvier 1994.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software* Addison Wesley, 1995.
7. Johann Jeuring. *Polytypic pattern matching* Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming and Computer Architecture, pages 238–248, San Diego 1995.
8. Andrew Shalit *The Dylan Reference Manual* Addison-Wesley, September 1996
9. Karl J. Lieberherr. *Adaptive Object-Oriented Software : The Demeter Method with Propagation Patterns* PWS Publishing Company, Boston, 1996.

10. James Gosling, Bill Joy, Guy Steele. *The Java Language Specification* The Java Series, Addison Wesley, 1996.
11. Tim Lindholm, Franck Yellin. *The Java Virtual Machine Specification* The Java Series, Addison Wesley, 1996.
12. Martin Odersky and Philip Wadler. *Pizza into Java: Translating theory into practice* 24th ACM Symp. on Principles of Programming Languages, 1997.
13. John Boyland and Giuseppe Castagna *Parasitic Methods: An Implementation of Multi-Methods for Java.* OOPSLA '97 Atlanta, ACM 0-89791-000-0/97 USA 1997.
14. Jens Palsberg and C. Barry Jay. *The Essence of the Visitor Pattern* In Proceedings of COMPSAC'98, 22nd Annual International Computer Software and Applications Conference, pages 9-15, Vienna, Austria, August 1998.
15. Loic Correnson, Etienne Duris, Didier Parigot and Gilles Roussel. *Declarative Program Transformation: a Deforestation case-study.* International Conference on Principles and Practice of Declarative Programming (PPDP'99), Paris, October 1999.