Higher Order Programming through Java Reflection

Marco Bellia[1] and M. Eugenia Occhiuto[2]

[1] Università di Pisa, Italy
[2] Università di Genova, Italy

Abstract. Higher order is considered a good methodology for program design and specification, furthermore it is fundamental for rapid prototyping. In this paper, we show different approaches to write higher order programs in Java. Drawbacks and benefits of such approaches are analyzed with reference to an example which is a typical one for higher order programming.

1 Introduction

Higher order programming is considered a characteristic of functional languages. In this class of languages, in fact, programs are functions and functions are first class values of the language. This means that functions can be passed as parameters to other functions, returned as result of the computation and furthermore, functions can be values in data structures (i.e. we can have lists, records and arrays of functions). The benefits obtained by higher order programming are in the expressivity of the code, which becomes more concise, clear and well structured and can be reused more easily. These topics are extensively discussed in the literature on functional programming. References, that are a starting point, are [Hud00, Hud89, Wad92].

The drawback of higher order is the inefficiency of the implementation, for this reason functional languages have never become an alternative to the imperative ones, instead limited higher order features have been added to imperative, first order languages to improve their expressivity. Examples in this direction are Pascal [JW75], C [KR88] and C++ [Sch95]. Such languages have been defined providing features to allow to pass programs (i.e. procedures or functions) as parameters to other programs. In fact, in this way an important abstraction mechanism is added to the language, in particular, programs are abstracted with respect to the programs they invoke in their body.

In a sense, also object-oriented languages originate from an attempt to add higher order feature to imperative languages. In fact, in this case, objects[3] are first class values of the language. Objects contain values (instance variables) and methods (instance methods), both are bound to names. Hence objects are environments, a particular kind of functions of type (Ide $\rightarrow$ {Val $\cup$ Methods}). Methods, in this case, are not themselves values, but are contained in objects

[3] There can be some exceptions: for instance, primitive values in Java

which are values, hence can be passed as parameters, returned as result and stored in data structures by passing, returning and storing the object containing the method. In this way the language provides a kind of higher order, [MW97], which helps in getting some higher order programs but does not cope with general higher order programming, i.e. functionals typically used and defined in functional languages. In this paper we address the problem of writing general higher order programs in Java. Several approaches are considered starting from those based on reflection. In Java reflection has been generally confined to event driven programming [Fla04] for instance for the implementation of graphical interfaces, in order to obtain a more compact and reusable code. In such cases, any way, resort to reflection can be replaced by anonymous inner classes without loose of generality. Higher order programs, on the contrary cannot be written using other techniques without loosing important properties as we will see in the sequel.

In Section 2 we introduce a problem and show a solution in Java using only first order methods. In Section 3 we show three solutions in pure Java defining higher order functions, all use reflection but in different ways. In Section 4 three different approaches to the problem are considered they are anonymous inner classes 4.1, delegates 4.2 and the Pizza language 4.3, a Java extension providing parametric polymorphism and higher order functions. The last section concludes the paper.

2 Why higher order programming in object-oriented languages?

Suppose we want to define collections of geometric shapes with methods to compute areas and perimeters of such shapes, and also collections of areas and perimeters. In Figure 1 we start defining generic collections of Objects in the class FList that defines lists with the usual methods insert, val and tail operators. The

```
public class FList {
    private Object elem; private FList next; private int sz;
    public FList () {sz=0; }
    public void insert (Object x) {
            if (sz==0) {elem=x; sz=1; next= new FList();}
            else{sz++; next.insert(x);}}
    public Object val () { return elem;}
    public FList tail () { return next;}
    public int size () { return sz;} }
```

Fig. 1. Class FList definition

definitions of such methods are trivial when it is clear that the empty list is an

object of type FList in which every instance variable is null except sz (representing the size of the list) which is 0. Then we define the class Shapes and several subclasses for specific geometric shapes, Figure 2. The third class ShapeList is

```
public abstract class Shape {
    public abstract double area();
    public abstract double perimeter();}
public class Rectangle extends Shape {
private double base;
private double height;
public Rectangle ( double b, double h){base=b; height=h;}
    public double area() {return base * height; }
    public double perimeter(){ return 2*base+2*height;}
public class Circle extends Shape {
    private double radius;
    public Circle(double r){radius=r;}
    public double area() { return new Double(radius *radius*PI);}
    public double perimeter() { return new Double(radius*2*PI);}}
public class ShapeList extends FList {
    public FList areaList(){
            FList L= new FList();
            if (size()!=0) {L=((ShapeList)tail()).areaList();
                    L.insert(((Shape) val()).area());}
            return L; }
    public FList perimeterList(){
            FList L= new FList();
            if (size()!=0) {L=((ShapeList) tail()).perimeterList();
                    L.insert(((Shape) val()).perimeter());}
            return L; } }
```

Fig. 2. The first order solution in Java

an extension of FList defined with two methods areaList and perimeterList, since we want to compute the list of values that are areas or perimeters of a given list of shapes.

As it is clear, examining the code, the method areaList and the method perimeterList are constituted by the same code except for the name of the method in the invocation. Such a code repetition could be avoided if the language allowed to pass methods as arguments to other methods, hence to use a higher order programming methodology to write programs. Suppose this is possible and hence we can write a higher order method, with a method as argument. In functional languages such a functional is called map and has two arguments, the first one is a function, the second one is a list, it computes the list of the values obtained applying the function to each element in the list. A first tentative definition of map is given in Figure 3. Since map is defined as an instance method of FList, it

has one parameter F, which is a method. ShapeList is still defined with its two methods but these are mere invocations of map. Allowing to pass programs (i.e. procedures, functions or methods depending on the language) as parameters to other programs, is adding an important abstraction mechanism to the language, in particular, programs are abstracted with respect to the programs they invoke in their body. In fact map is the abstraction of areaList and perimeterList with respect to the methods area and perimeter they invoke in their body. Unfortunately, this solution does not work, because val().F()) is not recognized as legal Java expression. F infact is a

```
public class FList { ...same as above for instance variables, methods etc.
      public FList map(Method F){
            FList L= new FList();
            if (size()!=0) { L=tail().map(F);
                  L.insert(val().F()));}
            return L; }}
public class ShapeList extends FList {
      public FList areaList() {
            return map(area); }
      public FList perimeterList() {
            return map(perimeter); }}
```

Fig. 3. A first higher order program

3 Solutions in Java

In this section we show how higher order functions can be defined in Java through the reflection package in java.lang. Firstly, we briefly illustrate few features, not commonly used in Java programs, that are used in this paper:

Class is the class of all classes in a Java application. It is defined in java.lang

Method is the class of all the methods both static and of instance. It is defined in java.lang.reflect

Class getClass () is the instance method of Object, which returns the class of the object.

Method getMethod(String name, Class [] parameterType) is the instance method of Class that returns the method, among those of the class , whose name is equal to name and whose parameters are equal to parameterType.

Object invoke(Object obj, Object [] args) instance method of Method, which allows the invocation of the method on which it is issued. Invoke has two parameters, obj of type Object, which is the object of the invoked method and args of type Object [] an array of objects that are the parameters to be

passed to the method. Invoke may generate exception (IllegalAccessException, IllegalArgumentException, InvocationTargetException).

The classes contained in the package java.lang.reflect can be used to define all kinds of higher level methods, that is either methods with methods as parameters and also methods returning methods as result and even data structures containing methods. Nevertheless, the language is not higher order and the only expression that can return a value of type Method is an invocation to getMethod. The invocation to getMethod must be issued on the class of the object, providing the method name and the type of its parameters. This is concerned with the Java overloading and overriding mechanisms which allow to define several static or non-static methods with the same name in different levels of the class hierarchy. Another problem is concerned with static type checking of types of the methods used as formal parameters. In fact, suppose m is declared to be a method with an argument x of type Method. In no way domain and range type of x can be declared, hence a complete type checking cannot be performed in the body of m. As a consequence run time exception may occur if a method of the specified name and type does not exist or the type of the actual parameter is not compatible with its use.

A critical aspect in object-oriented languages, related with the problem of higher order programming, is the fact that methods can be either static or instance methods. Higher order functions like map behave differently whether their parameters are static or instance methods. In this paper we privilege solutions based on instance methods since the o-o paradigm is based on objects with instance methods. In some case, any way, resorting to static methods provides interesting different solutions as the one in Figure 7.

A first solution, which uses only instance methods, is shown in Figure 4. In this case map argument is a String which is passed to each recursive invocation, hence a class and a method are computed for each element in the list. This is necessary because of hierarchies and overriding mechanisms. Infact, in this case, the list contains elements of type Shape which can be Circle or Rectangle and the methods to compute the area of circles and rectangles are different and are defined in each subclass. The benefits of such a solution are simplicity and clearness of the code of the extended class, e.g. ShapeList. Infact all the ugly code needed to invoke and access the method is hidden in map definition, once and forall. This is due to the choice to define map with a String argument instead of a Method argument, as for instance in the solution presented in Figure 7.

Another version of this first solution is shown in Figure 5, it also has method map with a String argument and shares the same benefits of simplicity and clearness. Such solution uses the abstract class Enumerator whose complete definition is available at [Occ], in Figure 6 a partial definition of Enumerator is shown, with the definitions of the two methods used in the example. Enumerator and its subclasses, firstly defined for Pizza at [ORW02], provide a number of higher order functions usually defined in functional languages for lists. The abstract class Enumerator, which implements the interface Enumeration, is defined to provide an enumeration on which the higher order methods can be used. The methods

```
public class FList { ...same as above for instance variables, methods etc.
      public FList map(String F){
            Object [] Arg={};
            FList L= new FList();
            try {
                  if (size()!=0) { L=tail().map(F);
                        Method M=val().getClass().getMethod(F,Arg);
                        L.insert(M.invoke(val(),Arg));}
            catch (Exception e){}
            return L; }
            }

public class ShapeList extends FList {
      public FList areaList() {
      return map("area"); }}
```

Fig. 4. A first Java definition of map and its application

defined are: filter, dropWhile, takeWhile (two particular filters), map, forall, reduceLeft and reduceRight. All filters and map are implemented similarly using a corresponding class, which implements methods nextElement and hasMoreElements; Figure 6 shows map definition. Method forall iterates the application of a method, which is its second argument f, on each element in the enumeration. This solution has the benefit to provide a library of higher order methods which can be used on any kind of data type which generates an enumeration of homogeneous values. This can be obtained, using a well known methodology, as shown in Figure 6. where the definition of class FList is refined with the enumList method, which provides an Enumerator for the elements of the list. Such a method enumList is defined through the class myEnumerator which is an extension of the abstract class Enumerator for which methods hasMoreElements and nextElement are defined. Notice that map (of Enumerator) does not construct the list of the areas but returns an enumeration of such area values, while it is forall the method which constructs the list, by means of insert. Note also that map (of Enumerator) has a String argument (for the name of the method) hence class MapEnumerator has an instance variable of type String. As a consequence, for each invocation of nextElement the class and the method to invoke are computed (through getClass and getMethod).

A different solution which perhaps more strictly resembles map in functional languages is shown in Figure 7. This solution is quite similar to the solution in Figure 4, it does not resort to Enumerator, but defines method map having an argument of type Method. The problem to invoke the correct method to compute the area of the geometric shapes is solved using a static method myArea which simply invokes the instance method area on the argument s thus obtaining the invocation of the correct method, through dispatching. Note also the use

```
public abstract class Enumerator {
public abstract boolean hasMoreElements();
public abstract Object nextElement() throws NoSuchElementException;
public Enumerator map (String f){return new MapEnumerator(f,this);}
public void forall (Object o,Method f) {
    while (hasMoreElements()) {
        Object [] Parm={nextElement();}
        try { f.invoke(o,Parm);} catch (Exception e){}}}}
public class MapEnumerator extends Enumerator {
    private String s; private Enumerator dom;
    public MapEnumerator (String f, Enumerator d){s=f;dom=d;}
    public boolean hasMoreElements(){return dom.hasMoreElements();}
    public Object nextElement() {
        try     {Object o=dom.nextElement(); Object [] Parm={};
                 Method m=o.getClass().getMethod(s,Parm);
                 return m.invoke(o,Parm);} }
        catch (Exception e){ }
    throw new NoSuchElementException();}}
...
```

Fig. 5. Abstract class Enumerator and higher order methods definition in Java

```
public class FList {...same as in Figure 1
public Enumerator enumList(){ return new myEnumerator(this);}
private static class myEnumerator extends Enumerator{
    private FList L;
    public myEnumerator (FList LL){L=LL;}
    public boolean hasMoreElements(){ return L.size()!=0;}
    public Object nextElement(){
        try {Object v= L.val(); L=L.tail(); return v;}
        catch(EmptyException e) { throw new NoSuchElementException();}}}
public class ShapeList extends FList {
public ShapeList() { super();}
public FList map(String s) {
    FList l= new FList(); Class [] ParamI={new Object().getClass()};
    try {   this.enumList().map(s).forall(l,l.getClass().getMethod("insert",ParamI));}
    catch (Exception e){}
    return l; }
...
ShapeList a=...
...
FList al=a.map("area");
...
```

Fig. 6. A map definition based on Enumerator

of getStaticMethod and forName to cope with static methods. This solution is slightly more complex than the one in Figure 4, mainly for the additional static method definition.

```
public class FList { ...same as above for instance variables, methods etc.
    public FList map(Method F){
        FList L= new FList();
        try {
            if (size()!=0) { L=tail().map(F);
                Object [] Arg={this.val()};
                L.insert(F.invoke(null,Arg));} }
        catch (Exception e){}
        return L; }}

public class ShapeList extends FList {
    public static double myArea(Shape s) { return s.area();}
    public FList areaList() throws NoSuchMethodException, EmptyException{
        Class [] Arg={};
        return map(getStaticMethod(Class.forName("ShapeList"),"myArea")); }
```

Fig. 7. A solution which uses static methods

4 Comparison with other approaches

In this section we analyze three other proposals which provide solutions for higher order programming of Java-like languages. They are: anonymous inner classes, delegates in Visual J++ and the Pizza language.

4.1 Anonymous inner classes

The idea underlying this approach is to emulate functions using objects of a particular kind of class(es). Each function is represented by one class and all the classes representing functions have a unique method apply which must be invoked to apply the function to its arguments. In this approach anonymous inner classes are a convenient way to express function values, which are passed to or returned by other functions. In [Set03], such an approach is used to model a calculus of λ-terms with a by-value evaluation. In this section we show how anonymous inner classes can be used to provide a solution of the problem defined in Section 2. In Figures 8, we define an interface Compute and the method map in the class FList, and classes Shape, Rectangle, and also Circle which is omitted but can be easily derived. In Figure 8 map invocation is also shown.

In our opinion this approach has two main drawbacks:

– It applies only to static methods. In fact, instance variable name should be passed as parameter and access to the value of the instance variable could be obtained only through reflection. In this case computeArea is defined as a static method of class Rectangle.
– Arguments to higher order methods are not methods, but special values, in this case objects of class Compute. Hence only those functions, defined of type Compute can be passed as arguments to higher order methods. The program designer has to decide in advance which functions will be passed as arguments to higher order methods. This is a severe limitation, which vanishes higher order programming benefits.

```
public interface Compute {
        Object app(Object x);
    public FList map(Compute m){
        FList L= new FList();
        try {
            if (size()!=0) { L=tail().map(m);
                L.insert(m.app(this.val()));}}
        catch (Exception e){}
        return L; }}
public class Shape {
    public static Compute computeArea(){
        return new Compute(){
        public Object app(Object x) { return null;};};}}
public class Rectangle extends Shape {
    private double base;
    private double height;
    public static Compute computeArea( final Rettangle r){
    return new Compute(){ public Object app(Object x){
        return new Double(r.base*r.altezza);};};}
    ...
    LL=L.map(Shape.computearea());
```

Fig. 8. A solution using anonimous inner classes

4.2 Delegates

Visual J++ is a development environment [Cor04] for Java which provides delegates as a new feature. Delegates are objects which encapsulate methods. Two kinds of bounds are possible: early bound, at the time of the delegate creation, and late bound at invocation time. The first example shown in Figure 9 uses early-bound instantiation. Such a solution resorts to static methods definition

since passing a String name of the method to map would require invocation of getMethod to designate the method in the delegate instantiation.

```
delegates Object Fun(Object o);

public FList map(Fun m){
      FList L= new FList();
      try {
            if (size()!=0) { L=tail().map(m);
                  L.insert(m.invoke(this.val()));}}
      catch (Exception e){}
      return L; }}
...
public class Shape {
      public static Object myArea(Shape s);
            return null;};}
public class Rectangle extends Shape {
      private double base;
      private double height;
      public static Object myArea( final Rectangl r){
      return new Double(r.base*r.altezza);};};}

            ...
      LL= L.map(new Fun(Shape.myArea));
```

Fig. 9. A solution using delegates with early-bound instantiation

A solution using late-bound delegate instantiation is shown in Figure 9, where only map definition is shown since Rectangle, Shape and map invocation are those shown in Figure 4. As the reader will note, the two solutions are quite similar, they only differ because one uses reflection and the other one uses delegates. Hence delegates are a good economic alternative to the expensive and unrestricted use of reflection.

4.3 Pizza

Pizza is an extension of Java providing, beyond other features, function abstractions and parametric polymorphism which has been included in Java 1.5 [SM04] Since parametric polymorphism is rather standard, we will not further describe such a feature. On the contrary, functions abstractions require some comments. The language is extended either to define function types through ((TypeList) $\rightarrow$ Type) and also function values can be constructed through (fun TypeFun Body). Furthermore every method is a function hence can be passed as an actual pa-

```
delegates Object Fun();

public FList map(String s){
      FList L= new FList();
      try {
            if (size()!=0) { L=tail().map(s);
                  Fun m=new(this.val(),s);
                  L.insert(m.invoke());}}
      catch (Exception e){}
      return L; }}
```

Fig. 10. A solution using delegates with late-bound instantiation

rameter for a corresponding formal parameter defined as an abstract function. Methods can be defined having functions as parameters.

A Pizza solution of the previously presented problem is reported in Figure 11. Class Shape, Circle and Rectangle definitions are omitted since they are retained in Pizza as they are defined in Figure 2. The definition of FList takes advantage of the possibility to use generic types,(see the additional argument of FList), which guarantees the homogeneity of the elements inserted in a FList. More complicate to understand is the solution to obtain map, since it makes use of the abstract class Enumerator whose definition is available at [ORW02]. The class Enumerator is similar to the one defined for Java in Section 3, whose definition was inspired by the Enumerator defined in Pizza. Enumeretor is equipped with all the same methods, but their parameters are different because in Pizza function abstractions types are used to declare parameters types. For instance map has a function of type (A)→ B where A is the generic type of the enumeration elements, returns a new enumeration of values of type B computed applying the function to each element of the original enumeration. In Figure 11 a class FListEnumerator is defined to allow the enumerations of elements in a list. The definition of class ListShape with a method map is shown in Figure 11. The method map of ShapeList has an argument m whose type is a function type (Shape)→ B and it invokes methods map and forall of the Enumerator class. In particular forall is used to insert in a list the elements of the enumeration computed by map (of the Enumerator). Unfortunately, map (of Enumerator) argument cannot be the instance method Area of Shape since its type is ()→ B. Hence a static method myArea must be defined to be passed as actual parameter in map invocation. Also forall requires a parameter function of type (A)→ B but in this case l.insert is exactly the needed method. In Figure 11 the computation of the list of the areas is obtained by means of an invocation of map (of ShapeList) with myArea as parameter.

In our opinion this approach has the drawback of adding a new mechanism to define functions which already can be defined through methods. Furthermore,

```
public class FList ⟨C⟩{
    private C elem;private FList ⟨C⟩ next; private int sz;
    public FList() { sz=0; }
    public void insert(C x) {
        if (sz==0) {elem = x; sz=1; next=new FList();}
        else {sz++; next.insert(x);}}
    public C val(){return elem;}
    public FList ⟨C⟩tail() {return next;}
    public int size(){return sz;}
    public Enumerator ⟨C⟩ enumFList(){ return new FListEnumerator(this);}}
private static class FListEnumerator ⟨C⟩ extends Enumerator ⟨C⟩{
    private FList ⟨C⟩ L;
    public FListEnumerator (FList ⟨C⟩ LL){L=LL;}
    public boolean hasMoreElements(){return L.size()!=0;}
    public C nextElement(){C v= L.val(); L=L.tail(); return v;}}
public static Double myArea (Shape x){ return x.area();}
public static class ShapeList ⟨B⟩ extends FList ⟨Circle⟩ {
    public ShapeList() {super();}
    public FList⟨B⟩ map((Shape) → B m) {
        FList ⟨B⟩ l = new FList();
        this.enumFList().map(m).forall(l.insert);
        return l;}}
    ...
    ShapeList a=...
    ...
    FList⟨Double⟩} al=a.map(myArea);
    ...
```

Fig. 11. Pizza solution

using function abstraction types to define arguments types in method definitions, has the benefit to allow static type checking, but, on the other hand, constraining the actual parameters, may require the definition of new methods to match the formal type parameters (in the example the static method myArea).

5 Conclusions

In this paper we address the problem of writing higher order programs in Java. Different approaches have been considered and for all the solution of the same problem has been presented. Only those using reflection and delegates with late-bound provide satisfactory solutions, which do not constrain to use static methods or to decide too early about design choices.

A drawback of reflection is inefficiency since it is generally implemented through interpretation. However, reflection allows to extend the language with

new features, which can successively be directly and more efficiently be implemented, if they come out to be interesting. Eventually,techniques to improve efficiency of programs using reflection have already been investigated with success [N00].

References

[Cor04] Microsoft Corporation. Delegates in visual j++, 2004. msdn.microsoft.com/vjsharp/productinfo/visualj/visualj6/technical%-/articles/general/delegates/default.aspx.

[Fla04] D. Flanagan. *Java examples in a nutshell.* O'Reilly, 2004.

[Hud89] P. Hudak. Conception, evolution , and application of functional programming languages. *ACM Computing Surveys*, 21:359–411, 1989.

[Hud00] P. Hudak. *The Haskell school of Expression.* Cambridge University Press, 2000.

[JW75] K. Jensen and N. Wirth. *Pascal User Manual and Report.* Springer, second edition, 1975.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C programming Language.* Prentice-Hal, 1988.

[MW97] Odersky M. and P. Wadler. Pizza into Java: translating theory into practice. In *Proc. 24th Symposium on Principles of Programming Languages*, pages 146–159, 1997.

[N00] M. Braux . and J. Noy. Towards partially evaluating reflection in java. In *PEPM'00*, 2000.

[Occ] M.E. Occhiuto. www.di.unipi.it/˜ occhiuto/enumerator.java.

[ORW02] M. Odersky, E. Runne, and P. Wadler, 2002. pizzacompiler.sourceforge.net/examples/enumerator.html.

[Sch95] H. Schildt. *C++ The Complete Reference.* McGraw Hill, Inc, 1995.

[Set03] A. Setzer. Java as a functional programming language. In *TYPES 2002,LNCS 2646.*, pages 279–298, 2003.

[SM04] Inc Sun Microsystems. A sun developer network site, 2004. http://java.sun.com/j2se/1.5.0/download.jsp.

[Wad92] P. Wadler. The essence of functional programming. In *Proc. 19th Symposium on Principles of Programming Languages*, pages 1–14, 1992.

This article was processed using the LaTeX macro package with LLNCS style