Fundamenta Informaticae 67 (2005) 29–44
IOS Press

Higher order Programming in Java: Introspection, Subsumption and Extraction

Marco Bellia[*]
Dipartimento di Informatica, Università di Pisa,
Largo B. Pontecorvo 3, I-56127 Pisa, Italy
bellia@di.unipi.it

M. Eugenia Occhiuto
Dipartimento di Informatica e Scienze dell'Informazione,
Università di Genova, via Dodecaneso 35, I-16146 Genova, Italy
occhiuto@di.unipi.it

Abstract. Higher order programming is considered a good methodology for program design and specification, furthermore it is fundamental for rapid prototyping. The paper is devoted to higher order programming in Java and, more in general, in the OO programming paradigm. We discuss introspection to write higher order programs and compare this technique with other different, interesting approaches, including function emulation and function integration. Finally, we address the problem of embedding, in the OO paradigm, the mechanisms for method passing and method extraction that are basic to the higher order programming methodology.

1. Introduction

Higher order programming, *HO*, is considered the main programming methodology of functional languages [2]. 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 HO programming are in the expressivity of the code, which becomes more concise, clear and well structured and can be reused more easily.

[*]Address for correspondence: Dipartimento di Informatica, Università di Pisa, Largo B. Pontecorvo 3, I-56127 Pisa, Italy

These topics are extensively discussed in the literature on functional programming. References, that are a starting point, are [9, 8, 18].

The drawback of higher order is the inefficiency of the implementation, for this reason functional languages have never become an effective alternative to the imperative ones, instead limited HO programming features have been added to imperative, first order languages to improve their expressivity. Examples in this direction are Pascal [12], C [13] and C++ [16]. 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 features to imperative languages. In fact, in this case, objects 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 $\mapsto$ $\{\text{Val} \cup \text{Methods}\}$). Methods, in this case, are not themselves values, but are contained in objects 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, [15], which helps in getting some higher order programs but does not cope with general HO programming, i.e. functionals typically used and defined in functional languages. In the first part of the paper we show how it is possible to write general higher order programs in Java. Several approaches are considered starting from those based on *introspection*, which is obtained using the package java.lang.reflect, for Java reflection. Another approach is based on *emulation* of a function calculus through anonymous inner classes of Java [17]. The introduction of function abstractions in object-oriented programming is pursued in Pizza [15]. Finally delegates in J++ [6] are considered. All the analyzed approaches are valid techniques to support the methodology but none of them is definitely better than the others. Furthermore, all of them provide only an indirect way, to support the HO methodology, which limits the program expressivity and makes the use of HO programming a bit tricking. The use of method passing, in HO programming, calls the mechanism of method extraction and its conflicting relation with subsumption subtyping in the OO programming paradigm. Then, we investigate a way, in the first order ζ-calculus [1], to extend the OO paradigm with method passing and method extraction. Subsumption states that any object of a class is also one object of its superclass(es), hence it can be used in any context in which an object of the superclass(es) can be used. In other words, such an object could have the type of the objects of (each of) its superclass too. In particular, a method, once it has been extracted from an object, has the type that makes its invocation from objects, of the superclasses, well typed. However, since such a method was defined in the environment of an object of a subclass, the method body can contain references that have been introduced in the subclass but not in the superclass(es). Hence, at static time, the typing system says that it is correct”O, but, at run time, the application can go into one fatal error”O in trying to select components that are not defined in objects of the superclass(es). This conflict is well exemplified in [1], see pages 106-108. A way to overcome the situation is discussed in section 5 where we limit subsumption when terms are using extraction.

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. Section5 addresses the embedding, in the OO paradigm, of

```
public class FList {/*Aim: define list of elements with the usual methods insert, val and tail*/
    private Object elem; private FList next; private int sz;
    /*Implementation: using next of type FList and sz to represent the length
    of the rest of the list*/
    public FList () {sz=0;}
    /*Constructor: The empty list has all instance variables null except sz which is 0*/
    public void insert (Object x) {/*Effects: modifies this inserting a new element*/
        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;} }
```

Figure 1. Class FList

mechanisms for method passing and method extraction. 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. Then we define the abstract class Shapes and several subclasses for specific geometric shapes, Figure 2 and the extension ShapeList.

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 HO 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 a legal Java expression. F in fact is the formal parameter of map hence a value (of type Method) and not a method identifier. In section 5, it is shown how this problem can be overcome in a calculus of objects.

```
public abstract class Shape {
        /*Aim: define geometric shapes with methods area , val and perimeter*/
        public abstract double area();
        public abstract double perimeter();}
public class Rectangle extends Shape {
        /*Aim: define rectangles as extension of 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 {
        /*Aim: define circles as extension of Shape defining the methods area and perimeter*/
        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 {
        /*Aim: define lists of shapes as extension of FList with methods areaList and
        perimeterList*/
        public FList areaList(){
        /*Effects: construction of the list of areas of the shapes elements of this*/
                FList L= new FList();
                if (size()!=0) {L=((ShapeList)tail()).areaList();
                              L.insert(((Shape) val()).area());}
                return L; }
        public FList perimeterList(){/*Effects: analogous to areaList*/
                FList L= new FList();
                if (size()!=0) {L=((ShapeList) tail()).perimeterList();
                              L.insert(((Shape) val()).perimeter());}
                return L; }}
```

Figure 2. The first order solution in Java

```
public class FList { ... same as above for instance variables, methods etc.
    public FList map(Method F){
    /*Effects: construction of the list of values obtained evaluating F
    on each element of this*/
        FList L= new FList();
        if (size()!=0) {L=tail().map(F);
                L.insert(val().F());}
                return L; }}
public class ShapeList extends FList {
/*Aim: define lists of shapes as extension of FList*/
    public FList areaList() {return map(area);}
    public FList perimeterList() {return map(perimeter);}}
```

Figure 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

```
public class FList { . . . same as above for instance variables, methods etc.
      public FList map(String F){
      /*Effects: construction of the list of values obtained evaluating the
      method of name F on each element of this*/
            Object [] Arg={};
            FList L= new FList();
            try {if (size()!=0){      L=tail().map(F);
                                      Method M=val().getClass().getMethod(F,Arg);
                                      /*M is evaluated through getClass and getMethod */
                              L.insert(M.invoke(val(),Arg));}
            catch (Exception e){}
            return L; }
            }

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

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

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 HO 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 OO 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 5.

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. In fact, 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. In fact 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 5. Reader can compare this solution with the one in 5.2

Another version of this first solution is shown in [3], Such 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.

A different solution which perhaps more strictly resembles map in functional languages is shown in Figure 5. This solution is quite similar to the solution in Figure 4, it does not resort to Enumerator, but

```
public class FList { ...same as above for instance variables, methods etc.
    public FList map(Method F){
    /*Effects: returns the list of the values obtained evaluating the method F
    on each element in this. It doesn't resort to getClass and getMethod*/
        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")); }
        /* Since parameters of map are static methods, getStaticMethhod
        and forName are necessary*/
```

Figure 5. A solution which uses static methods

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 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.

4. Comparison with other approaches

In this section we analyze three other proposals which provide solutions for HO 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 [17], such an approach is used to model a calculus of λ-terms with a by-value evaluation. In [3] it is shown how anonymous inner classes can be used to provide a solution of the problem defined in Section 2.

In our opinion this approach has two main drawbacks:

- It applies only to static methods. In fact, instance variable names should be passed as parameters 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 HO programming benefits.

4.2. Delegates

Visual J++ is a development environment [6] 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 delegate creation, and late bound at invocation time. Two solutions of the problem defined in 2 are defined in [3], one using early bound and static methods, the other one using late bound and instance methods. Such solutions are very similar to those using 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 [11] 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 parameter for a corresponding formal parameter defined as an abstract function. Methods can be defined having functions as parameters.

A nice solution in Pizza of the problem defined in section 2 is shown in [3]. It uses the abstract class Enumerator defined in [14].

In our opinion this approach has the drawback of adding a new mechanism to define functions which already can be defined through methods. Furthermore, using function abstraction types, to define arguments types in method definitions, has the benefit to allow static type checking. On the other hand, in this way, the actual parameters are constrained to match the formal parameter types. For this reason, in the example the static method myArea had to be defined.

5. A calculus of objects with method passing and method extraction

The use of methods as parameters, in the methodology, requires two distinct mechanisms: method passing and method extraction. In this section, we address the problem of embedding such mechanisms in object oriented languages, extending the *ς-calculus* [1] with constructions for the two. In the sequel, in order to simplify the reading, we use the definitions and the notations of [1] with the following typographic changes: $\rightarrow$ for the *weak reduction* $\rightsquigarrow$, $\Rightarrow$ for *judgment* $\vdash$, $b\{v_1 \ldots v_n\}$ for *substitution*

$b\{\{v_1 \dots v_n\}\}$ of v_1 for x_1 … v_n for x_n in term $b[x_1 \dots x_n]$, namely the term b with, possibly free occurrences of letters $x_1 \dots x_n$.

Method passing is the ability to refer, within a method body, to one (or more) abstraction that is instantiated, i.e. bound, to a, possibly different, method each time the body is evaluated. From a syntactic point of view, method passing comes in the ζ*-calculus* by allowing methods to have methods, of a suitable kind, as parameters. We extend the ζ*-calculus* in the calculus ζ_x*-calculus*, as in figure 6, by providing:

- one additional (clause 3) type structure, *M-types*, in order to assign more complex types to methods;
- parameters in the methods (clause 5): in the ζ*-calculus*, methods have no parameter in addition to the parameter *self*;
- parameter passing in method invocation (clause 6): in the ζ*-calculus*, the parameter passing of *self* is implicit so we add a construct for general parameter passing;
- actual parameters (clauses 8): for brevity sake, we limit the structure of actual parameters to methods only ;
- method extraction (clause 10): in the ζ*-calculus*, operations on methods are *selection* and *abstraction* only, hence *extraction* becomes an additional operation;
- formal parameters: clauses 12 and 7 distinguish the parameter *self*, which is always the first of the tuple, and is of object type, *O-types*, from the remaining parameters which must be of method type, *M-types*.

T, T_i ∈ **Types** :=A | B
A, ^{a}A, A_i ∈ **O-types** :=
(1) k∈ Constants -*ground types*
(2) | [l_i :B_i $^{i\in 1..n}$] -*object type (l_i distinct)*
B, ${}^{\rho}$B, B_i∈ **M-types** :=
(3) A_0 B_1… B_n -> A -*method type*
a, b, o, b_i ∈ **Terms** :=
(4) x,y,y_i∈ Var -*variables*
(5) | [l_i = ς(x_i:A par) b_i $^{i\in 1..n}$] -*object former*
(6) | a.ρ act -*method invocation*
(7) | a.l <= ς(x:A par) b -*method update*

act ∈ **Actuals** :=
(8) ε | ⟨ρ act⟩
ρ, ρ_i ∈ **Method operations** :=
(9) l -*selection*
(10) a^l -*extraction*
(11) ς(x:A par) b -*abstraction*
par ∈ **Parameters** :=
(12) ε | y:B par
l ∈ method selectors

Figure 6. Syntax of ζ_x*-calculus*

Method Extraction is the ability to select a fragment of code, in a program, and to abstract it into a new method that has the same (observable) behavior. To support the HO methodology, we can limit the extent of the selectable code to methods that are correctly defined within objects. From a syntactic point of view, the construct comes in the calculus by clause 12 of figure 6. Though simplified in this way, method extraction does not turn out simple at all. In effect, a method has meaning in environments which conform to the one of the object it belongs to. The notion of ”conform” must guarantee enough bindings, of the right type, for the identifiers that occur free in the method body (and will implicitly be bound by the method self parameter). So the treatment of method extraction depends on the one that the calculus provides to *subtyping*, *inheritance* and *subsumption*. Following the principle that inheritance-is-subtyping, as it is in case of Java, a method that has been extracted from an object of type T can occur in a call from objects of type $T^{'}$, provided that $T^{'}$ is subtype of T. Hence, the main problem is the conflict”O between subsumption and method extraction in a *sound type system* as it emerged in [1], pag. 106-108. We will overcome such a problem by limiting subsumption to objects and to object components that are not methods.

5.1. Operational semantics

We extend the reduction relation, $\rightarrow$, of ζ-*calculus* to cope with method passing and method extraction. What”cs new? Method passing is pervasive to the new calculus, in particular method invocation in *Red-Select* is extended: in addition to *self* parameter u, the substitution $b\{u\rho_1 \ldots \rho_k\}$, in the premises, is considering methods ρ_i. *Red-Abstract* is a new rule that allows method introduction, in the invocation. It behaves exactly as Red-Select but the method ρ is bound only to the invocation instead of to the object. *Red-Extract* allows to take the method of selection ${}^{w}l_j$ of an object w and puts it into another object u as one additional method: in doing so, *Red-Extract* extends u into a new object v.

Noting that the new rules preserve the main properties of $\rightarrow$, in particular the rules system is Church-Rosser confluent and the relation $\rightarrow$ can be extended into a conguence on the terms.

Theorem 5.1. (Confluence)
Let $\rightarrow^*$ be the reflexive, transitive, closure of $\rightarrow$, and a, a_1, a_2, b terms:

$$\text{if} \quad a \rightarrow^* a_1 \wedge a \rightarrow^* a_2 \quad \text{then} \quad a_1 \rightarrow^* b \wedge a_2 \rightarrow^* b$$

Red-Abstract, as well as *Red-Extract*, requires that the objects, that are involved in the ultimate invocation, i.e. u and v, respectively, extend the protocol, i.e. the binding environments, of the respective objects a. In order to guarantee it, we need to extend the *type system* as it will be discussed in 5.3.

5.2. Example

To illustrate the calculus, we consider objects of the following type ${}^{A}List$.

$$\begin{aligned} {}^{A}List \doteq [\quad & \textit{val} : {}^{A}List \mapsto A, \\ & \textit{tail} : {}^{A}List \mapsto {}^{A}List, \\ & \textit{insert} : {}^{A}List \;\; A \mapsto {}^{A}List, \\ & \textit{map} : {}^{A}List \;\; B \mapsto {}^{B}List]. \end{aligned}$$

$$\textbf{Red-Object}\quad \frac{}{\Rightarrow v \rightarrow v}\ \textit{(where } v\equiv[l_i=\rho_i{}^{i\in 1..n}])$$

$$\textbf{Red-Update}\quad \frac{\Rightarrow a \rightarrow u \qquad j=1..n}{\Rightarrow a.l^u_j \Leftarrow \rho \rightarrow v}\ \textit{(where } u\equiv[l^u_i=\rho^u_i{}^{i\in 1..n}],\ v\equiv[l^u_i=\rho^u_i{}^{i\in(1..n)-j},\ l^u_j=\rho])$$

$$\textbf{Red\_Select}\quad \frac{\Rightarrow a \rightarrow u \quad \Rightarrow b_j\ \{u,\rho_1,\ldots,\rho_k\} \rightarrow v \qquad j=1..n}{\Rightarrow a.l_j\langle\rho_1,\ldots,\rho_k\rangle \rightarrow v}\ \textit{(where } u\equiv[l_i=\zeta(x_i,y^i_1,\ldots,y^i_{ni})b_i{}^{i\in 1..n}])$$

$$\textbf{Red-Abstract}\quad \frac{\Rightarrow a \rightarrow u \quad \Rightarrow b\ \{u,\rho_1,\ldots,\rho_k\} \rightarrow v}{\Rightarrow a.\rho\langle\rho_1,\ldots,\rho_k\rangle \rightarrow v}\ \textit{(where } \rho\equiv\zeta(x,y_1,\ldots,y_k)b)$$

$$\textit{(where } u\equiv[{}^ul_i={}^u\rho_i{}^{i\in 1..k}],\ w\equiv[{}^wl_i={}^w\rho_i{}^{i\in 1..n}], v\equiv[{}^ul_i={}^u\rho_i{}^{i\in 1..k}, l_{k+1}={}^w\rho_j],\ r\equiv\rho_1,\ldots,\rho_m)$$

$$\textbf{Red-Extract}\quad \frac{\Rightarrow a \rightarrow u \qquad \Rightarrow o \rightarrow w \qquad j=1..n}{\Rightarrow a.o{}^\wedge{}^w l_j\langle r\rangle \rightarrow v.l_{k+1}\langle r\rangle}$$

Figure 7. ζ_x-*calculus*: Operational Semantics

The objects have methods *insert*, *val*, *tail*, of the expected behaviours for ordinary computation with lists of objects of type A, and method *map* defined as below:

$$map \doteq \zeta(x :^A List, f : A \mapsto B)x.tail.map\langle f\rangle.insert(x.val.f)$$ [1]

Suppose that type A is type *Shape*, for objects that are representing, in ζ-*calculus*, geometric shapes of Section 2, and that o is an object of type *Shape*, *area* is a method of o, and that *list-a* is a list of objects of type *Shape*. Then,

list-a$.map\langle o^\wedge area\rangle$

results into a list, *list-b*, of objects: *i-th* object of *list-b* is equal to the object that $v_i.area$ computes, where v_i is the *i-th* object of *list-a*. Note that each object of *list-a* is invoking one identical method, that of object o. A different situation happens when we consider:

list-a$.map\langle\zeta(x : A)\ x.area\rangle$

In this case, each object of *list-a* is invoking the, possibly different, method that is bound to the

[1]Method *insert*, as well as methods *val*, *tail*, could be entirely defined in standard ζ-*calculus*. In this case,$insert(a)$, noting round parentheses, is invocation with *method passing through emulation of functions with objects* (see [1] pag 66-69).

selection *area* in the object. We show it. Suppose *list-a* is the object for the list $(a_1 a_2)$ that contains the two objects of type *Shape*: $a_1 \equiv [\dots, area = \rho_1, \dots]$ and $a_2 \equiv [\dots, area = \rho_2, \dots]$. Then, we obtain the sequence of figure 8.

List-a.map⟨ζ(x:A) x.area⟩
  by Red-Select*:*
→ x.tail.map ⟨f⟩.insert(x.val.f) {list-a, ζ(x:A) x.area}
  by repeated Red-Select - *invocation of* tail *and introduction of letter* list-a_2 *for the object that results from:*
→* list-a_2.map⟨ζ(x:A) x.area⟩.insert(a_1.ζ(x:A) x.area)
  by Red-Select *and by letter* empty *for the empty list:*
→ empty.map⟨ζ(x:A) x.area⟩.insert(a_2.ζ(x:A) x.area).insert(a_1.ζ(x:A) x.area)
  by Red-Select - *invocation of* map:
→* empty.insert(a_2.ζ(x:A) x.area).insert(a_1.ζ(x:A) x.area)
  by Red-Abstract:
→ empty.insert(x.area {a_2}).insert(a_1.ζ(x:A) x.area)
  by Red-Select:
→ empty.insert(a_2.ρ_2).insert(a_1.ζ(x:A) x.area)
  by repeated reductions:
→ empty.insert(a_2.ρ_2).insert(a_1. ρ_1)

Figure 8. The sequence of reductions of an invocation of the method *map*

5.3. The type System: Inheritance, Subsumption and Subtyping

We now equip the ζ_x-*calculus* with a first order type system that contains axioms for subtyping with inheritance and subsumption. In order to constraint *subsumption*, when dealing with method extraction, the system uses two distinct *inferences*: $\supset_s$ and $\supset$. Inference $\supset$ is defined by all the axioms of inference $\supset_s$, table A in figures 9-10, but rule *Subsume*, in figure 9, which concerns subsumption subtyping. In particular, in order to obtain the complete set of rules of the inference $\supset$, it suffices to replace $\supset_s$ with $\supset$, everywhere in table A.

In the table A-a, we have collected the basic rules for subtyping introduction, *Sub-Int*, the transitivity of subtyping, *Sub-Tran*, and the rule for inheritance, *Inherit*, and for subsumption, *Subsume*. In particular, *Inherit* states that a subtype is a type which extends another one, so going down in the type hierarchy, while the rule of subsumption goes up in the hierarchy. In this way, subsumption allows to assign *supertypes* to objects. This works well in the object oriented paradigm quite everywhere. In particular, *type unicity* becomes the unicity of the *minimum* among the assignable types. We use the minimum to allow that the program works well with objects, of whatever type, in the entire hierarchy of the supertypes

$$\textbf{Sub-Int}\ \frac{E \supset_S T}{E \supset_S T <: T} \qquad \textbf{Sub-tran}\ \frac{E \supset_S T <: T_1 \quad E \supset_S T_1 <: T_2}{E \supset_S T <: T_2}$$

$$\textbf{Inherit}\ \frac{E \supset_S B_i \quad (\forall i \in 1..n+m)}{E \supset_S [l_i{:}B_i\ ^{i\in 1..n+m}] <: [l_i{:}B_i\ ^{i\in 1..n}]}$$

$$\textbf{Subsume}\ \frac{E \supset_S a{:}\ {}^aA \qquad E \supset_S {}^aA <: {}^bA}{E \supset_S a{:}\ {}^bA}$$

Figure 9. Table A-a: Subtyping, inheritance and subsumption in ζ_x*-calculus*

of that minimum. This is what happens in the terms, of the calculus, by using the rules *Object*, *Update* and *Select*.

For the other two rules, we have to be more careful. We must arrange two distinct requirements. The first one demands that the type ${}^oA$ of the object o, from which the method is imported (in the case of Extract, or from which the method can be invoked, in the case of Abstract), must be a supertype of the type ${}^aA$, of the object a to which such method is applied. This restricts the classes of objects, i.e. of a's, that can invoke the method to only those classes that, by extending the type of o, contain all the *selections* that the object o can invoke, namely all the ones that the body of a method which is invoked from o can contain. This is expressed by imposing ${}^aA <: {}^oA$ in the premises of the rules. The second one demands that the type ${}^oA$, considered in the previous comparison, is the effective type of the object o and not one of its supertypes. This is expressed, in rule *Extract*, by imposing that the type ${}^oA$ has been inferred without resorting to the axiom *Subsum*, i.e. using inference $\supset$. In rule *Abstrat*, the requirement that ${}^oA$ is the effective type is guaranteed by the type annotations of the method, in particular from the fact that $x : {}^oA, y_1 : B^1, \ldots, y_k : B^k$ is a correct annotation, in E, for inferring a type for the body b. Before to prove that the system of types of table A assigns correct types to terms of the ζ_x*-calculus*, we consider some basic lemmas.

Lemma 5.1. (Subsumption)
For term a and type T:

$$\text{if} \supset a : T \text{ then } \exists T_s : \supset_s a : T_s \wedge \supset_s T <: T_s$$

Lemma 5.2. (Substitution)
For environment E, term $a[x]$, possibly having x free, term b and types T_x, T_a:

$$\text{if } E, x : T_x \supset_s a[x] : T_a \wedge E \supset_s b : T_x \text{ then } E \supset_s a\{b\} : T_a$$

Lemma 5.3. (Contravariance)
For environment E, term $a[x]$, possibly having x free, and types T_x, T_s, T_a:

$$\text{if } E, x : T_s \supset_s a[x] : T_a \wedge E \supset_s T_x <: T_s \text{ then } E, x : T_x \supset_s a[x] : T_a$$

$$\textit{(where } A\equiv[l_i\!: A\, B^1_i\ldots B^{ki}_i \text{ -> } A_i\ ^{i\in 1..n}])$$
$$\textbf{Object}\ \frac{E, x_i : A, y^1_i : B^1_i, \ldots, y^{ki}_i : B^{ki}_i \supset_s b_i : A_i \quad (\forall i\in 1..n)}{E \supset_s [l_i = \zeta(x_i : A, y^1_i : B^1_i, \ldots, y^{ki}_i : B^{ki}_i)\ b_i\ ^{i\in 1..n}]) : A}$$

$$\textit{(where } A\equiv[l_i\!: A\, B^1_i\ldots B^{ki}_i \text{ -> } A_i\ ^{i\in 1..n}])$$
$$\textbf{Update}\ \frac{E \supset_s a : A \qquad E, x_j : A, y^1_j : B^1_j, \ldots, y^{kj}_j : B^{kj}_j \supset_s b_j : A_j \qquad j=1..n}{E \supset_s a.l_j \Leftarrow \zeta(x_j : A, y^1_j : B^1_j, \ldots, y^{kj}_j : B^{kj}_j)\ b_j : A}$$

$$\textit{(where } A\equiv[l_i\!: A\ B^1_i\ldots B^{ki}_i \text{ -> } A_i\ ^{i\in 1..n}])$$
$$\textbf{Select}\ \frac{E \supset_s a : A \qquad E \supset_s \rho_i : B^i_j\, (\forall i\in 1..k) \qquad k=k_j \qquad j=1..n}{E \supset_s a.l_j\langle \rho_1, \ldots, \rho_k\rangle : A_j}$$

$$\textit{(where } {}^{o}A\equiv[{}^{o}l_i\!: {}^{o}A\ {}^{o}B^1_i\ldots {}^{o}B^{ki}_i \text{ -> } {}^{o}A_i\ ^{i\in 1..n}])$$
$$\textbf{Extract}\ \frac{E \supset_s a : {}^{a}A \qquad E \supset o : {}^{o}A \qquad E \supset_s {}^{a}A <: {}^{o}A \qquad E \supset_s \rho_i : {}^{o}B^i_j\, (\forall i\in 1..k) \qquad k=k_j \qquad j=1..n}{E \supset_s a.o\text{^}{}^{o}l_j\langle \rho_1, \ldots, \rho_k\rangle : {}^{o}A_j}$$

$$\textit{(where } \rho\equiv\zeta(x\!:{}^{o}A, y_1\!:B^1, \ldots, y_k\!:B^k)\ b])$$
$$\textbf{Abstract}\ \frac{E \supset_s a : {}^{a}A \qquad E \supset_s {}^{a}A <: {}^{o}A \qquad E \supset_s \rho_i : B^i\, (\forall i\in 1..k) \qquad E, x : {}^{o}A, y_1 : B^1, \ldots, y_k : B^k \supset_s b : A}{E \supset_s a.\rho\langle \rho_1, \ldots, \rho_k\rangle : A}$$

Figure 10. Table A-b: The typing rules of ζ_x-*calculus* terms

Theorem 5.2. (Subject reduction)
For closed terms a, v, and type T:

$$\text{if} => a \to v \wedge \supset_s a : T \text{ then } \supset_s v : T$$

Proof.

The proof is by induction on the derivation steps of the sentence $=> a \to v$, and it follows that of Abadi and Cardelli (see[1] pag. 87) for axioms *Red-Object, Red-Select* and *Red-Update* of ζ-*calculus*. In particular, that proof can be easily rephrased for the three corresponding axioms of ζ_x-*calculus*. So, here, we limit only to consider the case *Red-Abstract* and *Red-Extract*.

Case Red-Abstract. We suppose that $=> a.\zeta(x, y_1, \ldots, y_k)b\langle\rho_1, \ldots, \rho_k\rangle \to b\{u, \rho_1, \ldots, \rho_k\}$ because (*) $=> a \to u$ for some object u of type ${}^{u}A$. By premises of *Abstract*, a has type ${}^{a}A$, x has type A for ${}^{a}A <: {}^{o}A$, for each $1 \leq i \leq n$, y_i has the type B^i of method ρ_i and, both term b and term $a.\zeta(x, y_1, \ldots, y_k)b\langle\rho_1, \ldots, \rho_k\rangle$ have type A. But, by inductive hypothesis, ${}^{u}A'{}^{a}A$, since (*), hence, by lemma on substitution, term $b\{u, \rho_1, \ldots, \rho_k\}$ has type A.

Case Red-Extract. We suppose that $=> a.o^{\wedge o}l_j\langle m\rangle \to [\ {}^u l_i\ = {}^u\rho_i\ {}^{i\in 1..k}\ ,\ l_{k+1}\ = {}^w\rho_j].l_{k+1}\langle m\rangle$ because (**) $=> a \to u \equiv [{}^u l_i\ ={}^u\rho_i\ {}^{i\in 1..k}]$ and $=> o \to w \equiv [{}^w l_i\ ={}^w\rho_i\ {}^{i\in 1..n}]$ for $1 \leq j \leq n$. By premises of *Extract*, a has type ${}^aA$, o has type ${}^oA \equiv [{}^o l_i :{}^oA\ {}^oB_i^1 \ldots {}^oB_i^{k_i} \mapsto {}^oA_i\ {}^{i\in 1..n}]$, ${}^aA <: {}^oA$, and $a.o^{\wedge o}l_j\langle m\rangle$ has type ${}^oA_i$. By inductive hypothesis, ${}^aA \equiv {}^uA$ and ${}^oA \equiv {}^wA$, since (**), and, by *Inherit*, $[\ {}^u l_i\ = {}^u\rho_i\ {}^{i\in 1..k}\ ,\ l_{k+1}\ ={}^w\rho_j]$ has type ${}^vA$ such that ${}^vA <: {}^uA$. Hence, by lemma on *contravariance* and by *Object*, l_{k+1} has type ${}^vA\ {}^oB_j^1 \ldots {}^oB_j^{k_i} \mapsto {}^oA_j$ as its associated type in ${}^vA$, and $[\ {}^u l_i\ = {}^u\rho_i\ {}^{i\in 1..k}$ $,\ l_{k+1} = {}^w\rho_j].l_{k+1}\langle m\rangle$ has type ${}^oA_j$.

We conclude considerations on ζ-*calculus* by remarking that a calculus of object can be extended in order to allow, in a reasonable way, the use of *method passing* and *method extraction*. Although, ζ_x-*calculus* still lacks some important characteristics of Java, we think that this result can be adapted to the other calculi [7, 10, 5, 4] that are nearest to the language and, next, to Java itself.

6. Conclusions

We have motivated and exemplified resorting, in Java and in OO programming, to the HO programming methodology.

For all the analyzed approaches, we succeeded in providing higher order solutions resorting to static methods, but only the approaches which extend the expressivity of the language, through *reflection* or *delegates*, succeed in providing real object oriented, higher order solutions, using instance methods.

Hence, we have investigated, in the first order calculus ζ-*calculus*, a way to extends the OO programming language paradigm with the basic mechanisms that support HO programming. This lead to embedding *method passing* and *method extraction* in a calculus equipped with *subsumption* as one subtyping axiom. To overcome the conflict arising from the combination of extraction and subsumption, we reformulated the calculus in a way that it limits subsumption when the term is using method extraction.

Future work could be devoted to consider the expressivity that the various techniques, we have considered, offer in writing OO programs using HO programming methodology. Again, it is interesting to understand how our solution could be extended to other calculi and to Java. Again, the combination of method extraction and inheritance could exploit new applications for the combination of OO and HO programming. As a matter of fact, consider the term *list-a*$.map\langle o^{\wedge}area\rangle$ of the example in 5.2 and suppose that *o* is an object of a superclass of the classes of the objects occurring in *list-a*, suppose also that all such classes have a new, possibly, all different definition of the method *area*. Then, the evaluation of the term behaves as a brute force invocation of the method of the superclass instead of those the classes have redefined.

References

[1] Abadi., M., Cardelli, L.: *A theory of objects*, Springer-Verlag, 1996.

[2] Backus, J.: Can programming be liberated from the von Neumann style? A functional Style and its algebra of programs, *Communication ACM*, **21**, 1978, 613–641.

[3] Bellia, M., Occhiuto, M.: Higher Order Programming through Java Reflection, *CS&P*, 2004, 447–459.

[4] Bierman, G., Parkinson, M.: Effects and effect inference for a core Java Calculus, *Electronic Notes TCS*, **82**(8), 2003, 1–29.

[5] Clark, D.: An Object Calculus with Ownership and Containment, *8th. FOOL*, 2001.

[6] Microsoft Corporation, M.: Delegates in Visual J++, 2004, Msdn.microsoft.com/vjsharp/productinfo/visualj-/visualj6/technical% -/articles/general/delegates/default.aspx.

[7] Drossopoulou, S., Eisenbach, S.: Is the Java Type System Sound?, *Journal of Theory and Practice of Object Systems*, **5**, 1999, 3–24.

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

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

[10] Igarishi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ., *ACM TOPLAS*, **23**, 2001, 396–450.

[11] Sun Inc., S. M.: A Sun Developer Network Site, 2004, Http://java.sun.com/j2se /1.5.0/download.jsp.

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

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

[14] Odersky, M., Runne, E., Wadler, P.: 2002, Pizzacompiler.sourceforge.net /examples/enumerator.html.

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

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

[17] Setzer, A.: Java as a Functional Programming Language, *TYPES 2002,LNCS 2646.*, 2003, 279–298.

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