Fundamenta Informaticae 85 (2008) 35–50
IOS Press

Methods as Parameters: A Preprocessing Approach to Higher Order in Java

Marco Bellia, M. Eugenia Occhiuto*
Dipartimento di Informatica
Università di Pisa, Pisa, Italy
bellia@di.unipi.it; occhiuto@di.unipi.it

Abstract. The paper investigates the use of preprocessing in adding higher order functionalities to Java, that is in passing methods to other methods. The approach is based on a mechanism which offers a restricted, disciplined, form of abstraction that is suitable to the integration of high order and object oriented programming. We show how this integration can be exploited in programming through the development of an example. Then, we discuss how the expressive power of the language is improved. A new syntax is introduced for formal and actual parameters, hence the paper defines a translation that, at preprocessing time, maps programs of the extended language into programs of ordinary Java.

Keywords: Higher order programming, object oriented programming, preprocessing

1. Introduction

Higher order programming, *HO*, is considered the main programming methodology of functional languages [3]. 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 expressiveness 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 [15] and [25].

*Address for correspondence: Dipartimento di Informatica, Università di Pisa, Pisa, Italy

Though 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 expressiveness. Examples in this direction are Pascal [17], C [18] and C++ [23]. 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 abstraction mechanism is added to the language, in particular, programs are generalized with respect to the programs they invoke in the body.

Object-oriented languages improve code reusability and, in a sense, also add higher order features to imperative languages. In fact, in this case, objects are first class values of the language [1] and [10]. In Java, objects contain values (instance variables) and methods (instance methods) and can be seen as records [1] and interpreted as defining environments, ($Ide \mapsto \{Val \cup Methods\}$), binding selectors to either values or methods, namely *Ide* is the set of selectors that specify object identifiers, *Val* is the set of object values, and *Methods* is the set of object methods. Methods, in this case, are not themselves values, but are contained in objects which are values, hence one method can be *implicitly* passed as a parameter, returned as result and stored in data structures as far as an object containing such a method is passed to, returned from and stored in. In this way, the language provides a kind of higher order, [22]. This viewpoint is not of great help: It leads to programs weighted down from an indirect and tricky use of objects to refer to methods [5], and it leads to complicated codes in the attempt to rephrase, in Java, higher order programs already written, for instance, in a functional language.

In [6] and [7] several approaches to higher order programming in Java are considered. They are characterized by:

- *introspection*, using meta-methods through the package java.lang.reflect for meta programming in Java [5] and [9];
- *emulation* of a calculus of functions through anonymous inner classes of Java [24] or function pointers [20];
- *extension* of the language, introducing special entities for function abstractions as in Pizza [22] or delegates as in J++ [11].
- *integration* of languages, using JNI for interoperation of method and function execution as in Lambada [21].

All the approaches above are valid techniques to support the methodology but none of them seems definitely better than the others. Furthermore, all of them either provide only an indirect way to support the HO methodology or make a neat separation between functions and methods that limits the program expressiveness and makes the use of HO programming a bit tricky. A different solution within the extension approach is discussed in [6] and [7] where the problem of extending the OO paradigm with method abstraction and method extraction is investigated, in the first order ζ-calculus [1], and a way to overcome the conflict originated by combining extraction and subsumption is discussed.

In this paper we discuss an approach to extend Java with mechanisms to pass methods (not arbitrary functions) as parameters and to generalize method invocation. We also discuss how its implementation can be treated at preprocessing time. The main features of the approach are:

- It adds a new kind of parameter: the m_parameter. A m_parameter can be interpreted as defining a function that maps objects (or classes) into instance (respectively, class) methods.

- Since they are functions, m_parameters could be well defined by resorting to function abstraction. Our approach uses a mechanism much less general which offers a restricted, disciplined, form of function abstraction but suitable to the integration of high order and object oriented programming.
- It enhances code reusability by supplying the language with a higher order based, programming expressiveness.
- All the extensions that it introduces can be implemented by a preprocessing technique that maps programs of the extended language into programs of the ordinary language.

In order to show the benefits of the extensions, in Section 2, we compare the code, in Java, of a program that uses higher order generalization with the code in the extended language. In Section 3 we introduce the language extensions. In Section 4 we formally define the translation $\mathcal{E}$ that maps programs of the extended language into programs of standard Java. The last section concludes the paper.

2. Code reusability: generalization vs. inheritance

We illustrate the reasons and the benefits for extending the language with features for higher order generalization and how these features are integrated with those for object oriented programming, mainly code reusability and inheritance.

2.1. Example of the code development for a class of geometric shapes

Let us consider the development, in Java, of the code for the computation of the lists of the areas and of the perimeters of geometric shapes, equipped with methods to compute the area and the perimeter of the shape [16] [4]. In Figure 1 we start giving a very concise definition for generic *lists* of objects: class `FList` provides `Insert`, `Val` and `Tail` operators.

Then we define the class `Shapes` and several subclasses: one for each specific geometric shape, with two obvious methods `Area` and `Perimeter`. The fourth class `FListShape` is an extension of `FList`, equipped with two additional methods `FListArea` and `FListPerimeter`, to compute the list of areas and perimeters of a given list of shapes.

2.2. Higher order generalization

As it is clear, examining the code, the method `FListArea` and the method `FListPerimeter` are constituted by the same code except for the name of the method in the invocation. In effect, once we have defined one of the two methods, we would like to obtain the other one, using the mechanisms that the language furnishes for code reusability. Unfortunately in Java these mechanisms are based on class hierarchy and inheritance and they are unable to support this kind of code development. Higher order generalization would provide an adequate support to that form of code development and reusability. In fact it allows both to *generalize* invocation `((Shape)Val()).Area()` into `((Shape)Val()).F()`, where `F` is a functional variable, and to bind `F` to the right method. In the example of Figure 1, this would allow to define a higher order method, `Map`, with a parameter `F`. At each invocation of `Map`, the parameter `F` should be bound to an object operation, namely `Area` or `Perimeter`, which is to be used for `F`. The method `Map` would apply the operation bound to `F` to each object `Val().F()` of the list, and return the list

```
public class FList {
    private Object elem;
    private FList next;
    public FList () {elem=null; next=null;}
    public FList Insert (Object x) {
           FList l= new FList(); l.elem=x; l.next=this; return l; }
    public Object Val () { return elem; }
    public FList Tail () { return next; } }
public abstract class Shape {
    public abstract Double Area();
    public abstract Double Perimeter();}
public class Circle extends Shape {
    private double radius;
    public Circle(double r){radius=r;}
    public Double Area() {return new Double(radius*radius*Math.PI);}
    public Double Perimeter() {return new Double(radius*2*Math.PI);}}
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 new Double(base*height);}
    public Double Perimeter() {return new Double(2*(base+height));}}
public class FListShape extends FList {
    public FListShape Insert (Object x) {...}
    public FList FListArea(){
           FList L= new FList();
           if (Val()!=null) {L=((FListShape)Tail()).FListArea();
                  L=L.Insert(((Shape) Val()).Area());}
           return L;}
    public FList FListPerimeter(){
           FList L= new FList();
           if (Val()!=null) {L=((FListShape) Tail()).FListPerimeter();
                  L=L.Insert(((Shape) Val()).Perimeter());}
           return L; }}
```

Figure 1. A class of geometric shapes in Java

of the computed objects. The invocation of `L.Map(Area)` would compute like `L.FListArea()` while the invocation of `L.Map(Perimeter)` would compute like `L.FListPerimeter()`. Hence, the code for `Map` would be reused for `FListArea()`, `FListPerimeter()` and for any other method that is obtained by instantiating the generalization introduced in the definition of `Map`.

```
public class FList {
     ...same as above for instance variables, constructor and methods val, etc.
     public FList Map(Fun → Object F){
            FList L=new FList();
            if (Val()!=null){L=Tail().Map(F);
                             L=L.Insert(Val().F());}
            return L; }}
public class FListShape extends FList{
     public FListShape Insert (Object x){...}
     public FList FListArea() {return Map(Abs Area); }
     public FList FListPerimeter() {return Map(Abs Perimeter); }}
```

Figure 2. Classes FList and FListShape in the extended language

2.3. The *m_parameter*: A restricted form of function abstraction

We extend Java with a mechanism for higher order generalization of the sort described above. However we cannot simply pass methods as parameters because of the late binding semantics of method overriding [14]. Instead, we pass a *m_parameter*. It is denoted by `Abs m` and defines a mapping that, given an object, in an invocation, selects the most specific method of the object having name `m` and the right types for the arguments of that invocation. This yields the solution, in our extended language, in Figure 2: `Map` has one m_parameter `F` whose type is `Fun → Object`. This means that the methods bound to `F` have no arguments and compute a value of type `Object` as results. `FListShape` is still containing two methods, whose bodies are invocations of `Map`: `Map (Abs Area)` for `FListArea` and `Map (Abs Perimeter)` for `FListPerimeter`.

2.4. Integration with ordinary OO mechanisms

Since a m_parameter is a function from objects (or classes) into instance (respectively, class) methods, in the evaluation of `L.Maps(Abs Area)`, for instance, the parameter `Abs Area` stands for the function that, given an object v, returns the *most specific* [13] method `Area` defined for v. Assumed that `L` is the list $(v_1, ..., v_n)$, then `L.Map(Abs Area)` computes the list $(r_1, ..., r_n)$ where, for each $1 \leq i \leq n$, r_i is the result of $v_i.m_i()$ and m_i is the *most specific* method `Area` of object v_i. For $1 \leq i \neq j \leq n$, the methods m_i and m_j selected for object v_i and v_j, respectively, may differ, as it is the case when v_i is an instance of class `Circle` while v_j is instance of class `Rectangle`. Though objects of class `Circle` and objects of class `Rectangle` are objects of the superclass `Shape` and they may inherit from `Shape` the code for many methods, the code of method `Area` is based on subclass specialization and is different in the two classes. In this way we obtain a perfect integration of higher order generalization with the mechanisms of class hierarchy and inheritance allowing to write programs of an OO language using a higher order methodology.

3. Extension to the language

In this section we discuss the extensions to Java to support higher order generalization, in particular parameter passing and method invocation.

3.1. Method declaration

The syntax for method declaration, [13] §8.4, is modified in the following way:

MethodDeclaration::= *ResultType Identifier* (*[FType Identifier (, FType Identifier)*]*)
Block
FType::= *Type* | `Fun` *FTList* → *Type* | `Fun` *FTList* → *void*
FTList::= *[FType (, FType)*]*

it defines a new syntactic category, *FType*, which replaces *Type* in *MethodDeclaration*. *FType* can be an ordinary *Type* or a newly defined type, identified by the type constructor `Fun` that specifies the types of the arguments and type of the result of the methods that can be bound to the parameter, see §3.2.a.

3.2. Method invocation

The syntax for invocation, [13] §15.12, is modified in the following way:

MethodInvocation::= *Expression.Identifier* (*[AExpression (, AExpression)*]*)
| *Expression.Parameter* (*[AExpression (, AExpression)*]*)

AExpression::= *Expression*
| `Abs` *Identifier*

with the following meanings and constraints:

(a) method invocation is extended by the second rule of *MethodInvocation*. It adds invocations of the form `e.p(l)` where `p` is a formal m_parameter and `e` and `l` are object and list of arguments, respectively, of the invocation. Let `Abs m` be the actual m_parameter bound to `p`. Let `p` be of type `Fun` t_1`, ...`t_n → `t`. Let ⟦`e`⟧ be the value computed by `e`. Then the meaning of `e.p(l)` is the invocation of `e.m(l)`, provided that *i)* `l` is a list of expressions of type $t_1, \ldots t_n$, and *ii)* `m` is the name of a method, of the hierarchy of ⟦`e`⟧, that applies to a list of arguments of type $t_1, \ldots t_n$ and returns a value of type t.

(b) actual parameters are extended by the second rule of *AExpression*. It adds parameters of the form `Abs m`, where `m` is the name of a method. The meaning is a function that given an object `c` of any class hierarchy `C` yields the most specific method of `c` having name `m`.

3.3. Example

In the extended language we can write, for instance, in a class `C`, the declaration:

```
int A (B X, Fun int → int F) {return X.F(0);}
```

This defines a method `A`, which returns an integer and has two parameters: `X` of type `B` and `F` of type `Fun int → int`. Let `Abs M` be the actual parameter bound to `F`. Then, in the invocation `X.F(0)`, in the body of `A`, the most specific method of the class hierarchy of (the object bound to) `X`, having name `M`, one argument of type `int` and returning a value of type `int`, is selected and applied, with argument `0`, to `X`. For instance, let `c` be an object of class `C`, and `b` be an object of class `B`, the invocation `c.A(b, Abs M)`

ClassDeclaration ::= `class` *Identifier* [`extends` *Type*] [`implements` *TypeList*] {*(MemberDecl)** }
MemberDecl::=;
|*ModifiersOpt FieldDeclarator*
|*ModifiersOpt Identifier FParameters* [`throws` *QualifiedIdentifierList*] *Block*
|*ModifiersOpt Type Identifier FParameters* [`throws` *QualifiedIdentifierList*] *Block*
|*ModifiersOpt* `void` *Identifier FParameters* [`throws` *QualifiedIdentifierList*] *Block*
|*ModifiersOpt ClassOrInterfaceDeclaration*
|[`static`] *Block*
FParameters::= (*[FParameter (,FParameter)*]*)
FParameter ::= *[* `final` *] FType VariableDeclaratorId*
FType::= *Type* |`Fun` *FTList* → *Type* |`Fun` *FTList* → `void`
FTList::=*[FType(, FType)*]*
Selector ::= *.Identifier [Arguments]* |*.Par Arguments* |`.this`
|`.super` *SuperSuffix* |`.new` *InnerCreator* |[*Expression*]
Arguments::= (*[AExp (, AExp)*]*)
AExp::= *Expression* | `Abs` *Identifier*

Figure 3. Extended syntax [13]

applies `A` to `c` with arguments `b` and the m_parameter `Abs M`. In the evaluation of the body of `A`, `X.F(0)` yields the invocation of the most specific method of `b` which has name `M` and type `int` → `int`. Because of Java overriding many methods may exist with name `M` and type `int` → `int` for the objects of class `B`. Hence different objects `b` may involve different methods in the evaluation of `X.F(0)`.

4. Implementation

4.1. Preprocessing: Structure of a syntactic translation

The implementation proposed in this paper uses a preprocessing technique which maps programs, of the extended language, back into programs of Java 1.4 [13]. A formal definition of the transformation is given by the mapping $\mathcal{E}$. It applies, separately, to each class of the source program producing a corresponding class of an equivalent program in Java 1.4. $\mathcal{E}$ is a compositional transformation and this allows to express the translation through the collection of rules of Figures 4 and 5 that apply descending on the class syntactic structure of the extended language. Because of space limitations, we restrict the presentation to the class structure of Figure 3. $\mathcal{E}$ is indexed by the additional parameter ρ: It is an environment with the scope information; It contains the binding $\langle name, FType \rangle$ of each formal m_parameter, visible in the code, currently transformed by $\mathcal{E}$. At the basis of $\mathcal{E}$ there is the transformation of the higher order introduced by the functions `Abs m` into structures that can be handled at first order, in Java. All these functions:

- differ one another for the method that must be selected once the object to apply to and the types of the arguments of the invocation are known, while
- share the computation structure *i)* to apply the function to the object and the types, *ii)* to access the class hierarchy of the object, *iii)* to find the most specific method with that name and types of the arguments, *iv)* to apply the selected method to the object with the arguments of the invocation.

Let ClassDef $\equiv$ `public class A` $\{$
ModifiersOpt $Type_0$ Ide_0 *[=*Exp_0*]*;
$\ldots$
ModifiersOpt $Type_h$ Ide_h *[=*Exp_h*]*;
ModifiersOpt A($Type_{C_0}$ Ide_{C_0})$Block_{C_0}$
$\ldots$
ModifiersOpt A($Type_{C_k}$ Ide_{C_k})$Block_{C_k}$
ModifiersOpt $Type_{M_0}$ Ide_{M_0} ($FType_{FP_{M_0}}$ $Ide_{FP_{M_0}}$) $Block_{M_0}$
$\ldots$
ModifiersOpt $Type_{M_k}$ Ide_{M_k} ($FType_{FP_{M_k}}$ $Ide_{FP_{M_k}}$) $Block_{M_k}$
ModifiersOpt `void` $Ide_{M_{k+1}}$ ($FType_{FP_{M_{k+1}}}$ $Ide_{FP_{M_{k+1}}}$) $Block_{M_{k+1}}$
$\ldots$
ModifiersOpt `void` Ide_{M_n} ($FType_{FP_{M_n}}$ $Ide_{FP_{M_n}}$) $Block_{M_n}$

$\mathcal{E}[\![\text{ClassDef}]\!]_\rho =$ `public class A implements ApplyClass` $\{$
ModifiersOpt $Type_0$ Ide_0*[=*Exp_0*]*;
$\ldots$
ModifiersOpt $Type_h$ Ide_h*[=*Exp_h*]*;
ModifiersOpt A ($Type_{C_0}$ Ide_{C_0}*)*$Block_{C_0}$
$\ldots$
ModifiersOpt A ($Type_{C_k}$ Ide_{C_k}*)*$Block_{C_k}$
ModifiersOpt $Type_{M_0}$$Ide_{M_0}$($\mathcal{E}[\![FType_{FP_{M_0}}$ $Ide_{FP_{M_0}}]\!]_\rho$)$\mathcal{E}[\![Block_{M_0}]\!]_{\rho'_0}$
$\ldots$
ModifiersOpt $Type_{M_k}$$Ide_{M_k}$($\mathcal{E}[\![FType_{FP_{M_k}}$ $Ide_{FP_{M_k}}]\!]_\rho$)$\mathcal{E}[\![Block_{M_k}]\!]_{\rho'_k}$
ModifiersOpt `void` $Ide_{M_{k+1}}$($\mathcal{E}[\![FType_{FP_{M_{k+1}}}$ $Ide_{FP_{M_{k+1}}}]\!]_\rho$)$\mathcal{E}[\![Block_{M_{k+1}}]\!]_{\rho'_{k+1}}$
$\ldots$
ModifiersOpt `void` Ide_{M_n}($\mathcal{E}[\![FType_{FP_{M_n}}$ $Ide_{FP_{M_n}}]\!]_\rho$)$\mathcal{E}[\![Block_{M_n}]\!]_{\rho'_n}$

Figure 4. Transformation $\mathcal{E}$ - part 1

```
private static int Dispatcher(String S){int pos=-1;
     if (S.equals(ToS(Ide_{M_0}))) pos=0;
     else if ...
     else if (S.equals(ToS(Ide_{M_k}))) pos=k;
     else if (S.equals(ToS(Ide_{M_{k+1}}))) pos=k+1;
     else if ...
     else if (S.equals(ToS(Ide_{M_n}))) pos=n;
     return pos;}
public Object Apply(String M, Object Par) throws MethodNotFoundException{
     int pos=Dispatcher(M);
     switch (pos){case 0    :  return Ide_{M_0}((E[[FType_{FP_{M_0}}]]_ρ)Par);
                 ...
                 case k    :  return Ide_{M_k}((E[[FType_{FP_{M_k}}]]_ρ)Par);}
                 default   :  throw new MethodNotFoundException();}
public void ApplyS(String M, Object Par) throws MethodNotFoundException {
     int pos=Dispatcher(M);
     switch (pos){case k+1 :  {Ide_{M_{k+1}}((E[[FType_{FP_{M_{k+1}}}]]_ρ)Par);
                               break; }
                 ...
                 case n    :  {Ide_{M_n}((E[[FType_{FP_{M_n}}]]_ρ)Par);}
                               break; }
                 default   :  throw new MethodNotFoundException();}}
```

where: $\rho'_i = \mathcal{R}[\![FType_{M_k} Ide_{M_k}]\!]_\rho$

$\mathcal{R}[\![FType\ Ide]\!]_\rho(x)$ = *FType* if *Ide* = *x*

$\mathcal{R}[\![FType\ Ide]\!]_\rho(x) = \rho(x)$ if *Ide* $\neq$ *x*

$\mathcal{E}[\![Block]\!]_\rho = \mathcal{E}[\![St]\!]_\rho ; \mathcal{E}[\![StList]\!]_\rho$ with *Block = St; StList*

$\mathcal{E}[\![Arguments]\!]_\rho = [\mathcal{E}[\![\ AExp\ ]\!]_\rho (, \mathcal{E}[\![\ AExp\ ]\!]_\rho)^*$

$\mathcal{E}[\![AExp]\!]_\rho = \mathcal{E}[\![\ Expression\ ]\!]_\rho \mid \text{ToS}(Ide)$

$$\mathcal{E}[\![FType]\!]_\rho = \begin{cases} Type & \text{with} \quad FType = Type \\ \texttt{String} & \text{with} \quad FType = \texttt{Fun}\ FType \rightarrow Type \\ \texttt{String} & \text{with} \quad FType = \ \texttt{Fun}\ FType \rightarrow \ \texttt{void} \end{cases}$$

$$\mathcal{E}[\![St]\!]_\rho = \begin{cases} ((\texttt{ApplyClass})\mathcal{E}[\![Exp_1]\!]_\rho).\texttt{ApplyS(}Par, & \\ \quad \texttt{new Object[]}\{(FType)\mathcal{E}[\![Exp_2]\!]_\rho)\}, & \text{with } St = Exp_1.Par(Exp_2) \wedge \\ & \rho(Par) = \texttt{Fun } FType \rightarrow \texttt{void} \\ \mathcal{E}[\![Exp_1]\!]_\rho.Ide(\mathcal{E}[\![Exp_2]\!]_\rho), & \text{with } St = Exp_1.Ide(Exp_2) \wedge \\ & \rho(Ide) = \bot \\ \texttt{if}(\mathcal{E}[\![Exp]\!]_\rho)\mathcal{E}[\![St_1]\!]_\rho \texttt{ else } \mathcal{E}[\![St_2]\!]_\rho; & \text{with } St = \texttt{if } Exp\ St_1 \textbf{ else } St_2 \\ \texttt{while}(\mathcal{E}[\![Exp]\!]_\rho)\mathcal{E}[\![St]\!]_\rho & \text{with } St = \texttt{while } Exp\ St \\ \text{etc.} & \end{cases}$$

$$\mathcal{E}[\![Exp]\!]_\rho = \begin{cases} (Type)((\texttt{ApplyClass})\mathcal{E}[\![Exp]\!]_\rho).\texttt{Apply(}Par, & \\ \quad \texttt{new Object[]}\{(FType)\mathcal{E}[\![Exp]\!]_\rho)\}, & \text{with } Exp = Exp.Par(Exp) \wedge \\ & \rho(Par) = \texttt{Fun } FType \rightarrow Type \\ \mathcal{E}[\![Exp]\!]_\rho.Ide(\mathcal{E}[\![Exp]\!]_\rho), & \text{with } Exp = Exp.Ide(Exp) \wedge \\ & \rho(Ide) = \bot \\ \mathcal{E}[\![Exp]\!]_\rho\ Op\ \mathcal{E}[\![Exp]\!]_\rho & \text{with } Exp = Exp\ Op\ Exp \\ etc. & \end{cases}$$

Where:

- *Exp*, *St*, *StList*, *FParameters*, *Ide* stand for *Expression*, *Statement*, *StatementList*, *FormalParameters*, *Identifier* respectively;
- ToS(*m*) computes the string univocally associated to the method of name *m*

Figure 5. Transformation $\mathcal{E}$ - part 2

The idea is to have the computation structure of those functions as a sort of run time support that is included in the classes of the transformed programs, and used through suitable methods. This leads to methods, `Apply`, `ApplyS` that deal with all phases *i)-iv)* and are specific to each source class having methods that are passed as m_parameters in the source program. Invocations of `Apply` and `ApplyS` replace, in the transformed program, invocations in which the invoked method is a m_parameter. An interface `ApplyClass` defines them as two abstract methods:

```
public interface ApplyClass {
     public abstract Object Apply(String M, Object [] Pars) throws
                                        MethodNotFoundException;
     public abstract void ApplyS(String M, Object [] Pars) throws
                                        MethodNotFoundException;}
```

They have two parameters:

1. `M` is the string univocally (see *ToS* below) associated to the name of the method to invoke,
2. `Pars` is the array of the actual parameters of the method to invoke. Each parameter of the array is cast to the corresponding type of the argument type list bound to the m_parameter in the environment ρ.

`Apply` and `ApplyS` find, through `Dispatcher`, the method whose name is equal to `M` and invoke it with appropriate parameters taken from `Pars`. They differ because:

- `Apply` is used for methods returning a value, i.e. methods whose invocation is an expression. In this case `Apply` returns an object: the one computed by the invoked method. A type cast in $\mathcal{E}$ is imposed on `Apply` result.
- `ApplyS` is used for methods which do not return any value. In this case `ApplyS` simply invokes the method. No type cast is required.

Auxiliary method `Dispatcher` is introduced to help in structuring phase *iii)*. It traverses the class hierarchy to find the method of a given name. Found the class containing the right definition, it computes the position in the class of the method. A string, namely `ToS`(Ide_i), is univocally associated to each method name, Ide_i, for this purpose.

4.2. Example: The transformed program

The program resulting from the preprocessing of the program in Figure 2 is presented in Figures 6,7 and 8. The code for the class `Rectangle` is omitted since it is equal to the one for `Circle`. All environments ρ involved are empty except for class `FList`.

Let us consider now the execution of the method `FListArea` on a list of `Shapes` constituted of two elements a circle c and a rectangle r. Suppose the circle is the first element in the list, obtained applying `Val` in the first `Map` invocation. Hence, applying `Apply` to such element (of the class `Circle`) with argument the string `"Area"`, yields the invocation of the method `Area` of the class `Circle`. In the second recursive invocation, a rectangle is obtained by `Val` invocation. Hence applying `Apply` to such element (of the class `Rectangle`) with still `"Area"` yields the invocation of the method `Area` of the class `Rectangle`. So two different methods `Area` of the class `Circle` and `Area` of the class `Rectangle` are invoked, within a `Map` invocation with the same m_parameter `F`. This would not be possible if `F` value were a method, but more important, this is the behavior required in a language with class hierarchy and inheritance mechanisms.

4.3. Syntactic simplifications

We confined the presentation of $\mathcal{E}$ in several ways. However, $\mathcal{E}$ can be completed for full Java 1.4 extended with the mechanisms for m_parameters. In particular, source programs may contain: *i)* m_parameters that yield methods with more that one argument or that are class methods, *ii)* exceptions, *iii)* class hierarchies involving abstract classes and interfaces, *iv)* inner, embedded and local classes. Moreover, $\mathcal{E}$ makes a different treatment between the classes of the source programs that can contain methods that are passed as m_parameters and those that cannot. Only the transformation of the former requires, to be

```
public class FList implements ApplyClass {
       ...same as Figure 1 for instance variables etc.
       public FList Map(String F){
            FList L=new FList();
            if (Val()!=null) { L=Tail().Map(F);
                               L=L.Insert((Object)((ApplyClass)Val().Apply(F,
                                                    new Object[]{})); }
            return L; }
       private static int Dispatcher(String S){
            int pos;
            if (S.equals("Insert")) pos=0;
            else if (S.equals("Val")) pos=1;
            else if (S.equals("Tail")) pos=2;
            else if (S.equals("Map")) pos=3;
            else pos=-1;
            return pos;}
       public Object Apply(String M,
                           Object [] Pars) throws MethodNotFoundException{
            int pos=Dispatcher(M);
            switch (pos){   case 0:return Insert((Object)Pars[0]);
                            case 1:return Val();
                            case 2:return Tail();
                            case 3:return Map((String)Pars[0]);}
                            default :  throw new MethodNotFoundException();}}
       public void ApplyS ...}
```

where: ρ is such that $\rho(F) = \texttt{Fun} \rightarrow \textit{Object}$;

Figure 6. Class FList

```
public class Circle extends Shape implements ApplyClass {
       ...same as Figure 1 for instance variables etc.
       private static int Dispatcher(String S){int pos;
            if (S.equals("Area")) pos=0;
            else if (S.equals("Perimeter")) pos=1;
            else pos=-1;
            return pos;}
       public Object Apply(String M,
                           Object [] Pars) throws MethodNotFoundException{
            int pos=Dispatcher(M);
            switch (pos){  case 0:return Area();
                           case 1:  return Perimeter();}
                           default:  throw new MethodNotFoundException();}}
       public void ApplyS ...}
```

Figure 7. Class Circle

```
public class FListShape extends FList implements ApplyClass {
       public FList FListArea(){ return Map("Area");}
       public FList FListPerimeter(){return Map("Perimeter");}
       private static int Dispatcher(String S){
            int pos;
            if (S.equals("FListArea")) pos=0;
            else if (S.equals("FListPerimeter")) pos=1;
            else pos=-1;
            return pos;}
       public Object Apply(String M,
                           Object [] Pars) throws MethodNotFoundException{
            int pos=Dispatcher(M);
            switch (pos){ case 0:  return FListArea();
                          case 1:  return FListPerimeter();
                          default:  throw new MethodNotFoundException();}}
       public void ApplyS ...}
```

Figure 8. FListShape

completed, the code to implement the interface `ApplyClass`. The currently available implementation applies to programs of full Java 1.4 extended with m_parameters and an additional modifier for classes that explicitly declare whether the class is an implementation of `ApplyClass`. The first prototype was obtained from scratch and was developed using Lex-Yacc tools. This prototype uses the syntactic categories of Java 1.4. reported in chapter 18 of [13]. It is one-pass preprocessor and can be freely downloaded from [8]. A new, and more elaborate, implementation is in progress. The new implementation automatically detects the classes that are implementation of `ApplyClass`. Detection is obtained through the program analysis designed for the static checks discussed later on in 4.7: A class C can contain methods passed as m_parameters if both *(i)* and *(ii)* hold for one of its methods: *i)* it has same signature and same return type of a m_parameter occurring in the declaration of a method, let us say p; *ii)* it has same name of (the name in) the actual parameter, bound to the m_parameter, in one of the invocation of p in the program (having the class in its scope). Also this implementation, is one-pass preprocessor but it resorts to backpatching techniques [2] in order to complete the code of the classes that can contain methods passed as m_parameters once detected.

4.4. Overloaded methods

Currently, $\mathcal{E}$ assumes that overloaded methods are not yielded by m_parameters. In fact, selecting, in each invocation, the most specific signature among those matching the signature the invocation expects for the method, solves method overloading, in Java. Let $e.p(l)$ be an invocation of a method of name p which is overloaded in the class e (belongs to), the expected signature depends only from the type of l and is known at compile time. Dealing with m_parameters, the expected signature, in an invocation of p, now, passed as parameter, is the signature of the formal parameter, p is bound to, which is also known at compile time. Instead, in our approach, it is class e which is unknown at compile time. Since m_parameters are mappings from classes (resp. objects) to class (resp. object) methods, the knowledge

of the class where to look for the most specific signature should be deferred to run time. So the actual implementation forbids to pass overloaded methods, including constructors, as m_parameters. Similar arguments forbid also the use of overloaded methods with m_parameters. Nevertheless, along the lines sketched in 4.7, progress has been done to support overloaded methods as parameters. The idea is to specify the class (hierarchy) the passed method belongs to. This would be done through a different kind of m_parameters , namely mc_parameters that allow to specify the class (hierarchy) in addition to the signature.

4.5. Program locality

Our approach maintains textual locality [12]. In particular: *i)* the class structure of the extended language does not differ from that of ordinary Java nor the programs written in the extended language have a different structure than Java programs. Again, programming in the extended language does not require any additional ability and it leads to programs that show a structure of the classes defined in the program that is close to the one of programs obtained with ordinary Java: the latter ones can contain additional classes and methods in order to specialize, with the expected objects and methods, those invocations that, in the program in the extended language, are abstracted using m_parameters. *ii)* The transformation $\mathcal{E}$ does not modify the class structure of the source program. Noting that the same cannot be obtained when *function pointers* [20] are used since they require the introduction, in the program, of suitable additional classes to contain their definitions.

4.6. Basic types and Java library

Our approach is strongly based on the language class hierarchy. In Java 1.4, *basic types*, including `int`, `boolean` etc. are not classes. In particular `Object` is not a superclass of the basic types. This fact forbids the use, as m_parameters, of the common operators on these types , i.e. `+`,`-`, `==` etc. and also of each method having basic types in the signature. This impure treatment of basic types is modified in the most recent releases of Java, namely 1.5 and 1.6. They introduce adequate new classes, e.g. `Integer`, `Boolean` etc., for each basic type and suitable, automatic, conversion mechanisms, namely *boxing* and *unboxing* to pass from primitive values to the corresponding objects (and backward). In Java 1.4, we must explicitly define, in the program, the class, $\mathtt{C_t}$ in which boxing-unboxing the basic type `t` that is involved in a m_parameter. In particular each occurrence of `t` in the signature of the m_parameter must be replaced by $\mathtt{C_t}$ and if an operator is passed as a m_parameter, then a suitable method of $\mathtt{C_t}$ must be passed instead.

Special care is required when the m_parameter involves methods that are defined in Java API. In this case, it is necessary that the class, containing such a method, is equipped with the methods `Apply` and `ApplyS`, hence its definition must be preprocessed and then recompiled.

4.7. Types and static checks

The type of a formal m_parameter is `Fun FTList` $\rightarrow$ `Type`: It states number and types of the arguments and type of the result of the methods that can be used in any invocation in which the m_parameter applies. It has the advantage that such methods belong to different, possibly unrelated, class hierarchies. The effective hierarchy, as well as the effective definition, for overridden methods, depends on the object (resp.

class) the m_parameter applies to, in the invocation. This enhances abstraction and code reusability of the classes that are using m_parameters but forbids the ability to lead some static checks on the source program. For instance, we are unable to statically check the class hierarchies that will be involved, at run time, in method passing for the effective presence of a method with the right signature [13]. The situation is still more complicated if skew inheritance is considered [19]. The execution of one of the methods `Apply` or `ApplyS` of the transformed program can throw a `MethodNotFoundException()`. This guarantees program safety against any wrong use of m_parameters: in no way it can lead to untrapped errors. However, to perform deeper checks on the source program, as those required to deal with method overloading, we are currently considering the introduction of mc_parameters as a specialization of the m_parameters. An mc_parameter is simply a m_parameter with a class which can be interpreted as the root of the class hierarchy of all the classes to which the m_parameter can apply. Thus, the type of a mc_parameter has form `Fun Class: FTList` $\rightarrow$ `Type` that in addition specifies the class hierarchy `Class`. The analysis of the methods in the hierarchy rooted in `Class` can be used: *i)* to find all the classes that are implementation of `ApplyClass`; *ii)* to look for the presence of a method named `m` and matching the signature `FTList`, for each invocation that binds the mc_parameter to the actual `Abs m`; *iii)* to have a technique to statically solve overloading when method named `m`, in the actual `Abs m`, is overloaded and has most specific signature `FTList'`. The replacement of `FTList` with `FTList'` in the type of mc_parameter is promising to solve overloading problem. The implementation under development supports such a statically analysis technique.

5. Conclusions

The paper discussed a preprocessing implementation that supports Java 1.4 extended with higher order functionalities. The approach is based on m_parameters, which introduce functions that map each object (class) of the program into the most specific method of the object (resp. class) having a given name and type. This notion offers a restricted, disciplined, form of function abstraction but suitable to the integration of higher order and object oriented programming. The main advantages of this solution are that it is simple and expressive, it copes with code reusability and it is well suited to class hierarchy and inheritance mechanisms of Java. It is reasonably efficient, since it uses one-pass preprocessing. It does not require any modification to JVM. The approach maintains textual locality. In particular the class structure of the extended language does not differ from the one of ordinary Java and the transformation does not modify the class structure of the source program. The approach guarantees program safety against any wrong use of m_parameters. Although, the presentation of $\mathcal{E}$, in the paper, has been given for a limited program structure, an implementation of it for full Java 1.4 is currently available at [8]. Currently, $\mathcal{E}$ assumes that overloaded methods are not yielded by m_parameters. However, the use of a specialized form of m_parameters provides a promising solution to extend method passing to overloaded methods and is currently under development in a new and more elaborate one-pass, backpatched, preprocessing.

References

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

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Tecniques, and Tools.* Addison-Wesley, 1988.

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

[4] J. Bloch. *Programming Language Guide*. Prentice Hall PTR, 2001.

[5] M. Bellia and M.E. Occhiuto. Higher order programming through Java reflection. In *CS&P'2004*, volume 3, pages 447–459, 2004.

[6] M. Bellia and M.E. Occhiuto. Higher order programming in Java: Introspection, Subsumption and Extraction. *Fundamenta Informaticae*, 67(1):29–44, 2005.

[7] M. Bellia and M.E. Occhiuto. *From Object Calculus to Java with Passing and Extraction of Methods*. University of Pisa, Dipartimento Informatica, 2006.

[8] M. Bellia and M.E. Occhiuto. Jh-preprocessing, 2007. //http://www.di.unipi.it/ occhiuto/JH/.

[9] B. Bringert. HOJ - higher-order Java, 2005. //cs.chalmers.se/bringert/hoj/.

[10] G. Castagna. *Object-Oriented Programming: A Unified Foundation*. Birkhauser Verlag AG, 1997.

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

[12] R. Dyer, H. Narayanappa, and H. Rajan. Nu: Preserving design modularity in object code. *ACM SIGSOFT Software Engeneering Notes*, 31, 2006.

[13] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification - Second Edition*. Addison-Wesley, 2000.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification - Third Edition*. Addison-Wesley, 2005.

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

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

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

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

[19] H. Langmaack, A. Salwicki, and M. Warpechowski. A deterministic algorithm elaborating direct superclasses in java-like languages. In *CS&P'2007*, pages 388–399, 2007.

[20] G. McCluskey. Using method pointers and abstract classes vs. interfaces. *Electronic Notes TCS*, 2001. //java.sun.com/developer/JDCTechTips/2001/tt1106.html.

[21] E. Meijer. Lambada, Haskell as a better Java. *Electronic Notes TCS*, 41(1), 2001.

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

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

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

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