Methods as Parameters: a Preprocessing Approach in Java

Marco Bellia and M. Eugenia Occhiuto
Dipartimento di Informatica,
Università di Pisa e-mail: occhiuto@di.unipi.it

July 14, 2007

Abstract

In this paper we investigate the possibility of adding higher order functionalities to Java, that is passing methods as parameters to other methods. The approach is based on a mechanism which offers a restricted, disciplined, form of function abstraction but suitable to the integration of high order and object oriented programming. We discuss how the expressive power of the language is increased. A new syntax is introduced for formal and actual parameters, hence the paper shows an implementation of the extentions based on a preprocessing technique tha maps programs of the extended language into programs of ordinary Java.

1 Introduction

Higher order programming, *HO*, is considered the main programming methodology of functional languages [Bac78]. 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. These topics are extensively discussed in the literature on functional programming. References, that are a starting point, are [Hud89, Wad92].

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 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 generalized with respect to the programs they invoke in their body.

Object-oriented languages originate from an attempt to add code reusability and, in a sense, also higher order features to imperative languages. In fact, in this case, objects are first class values of the language [AC96, Cas97]. In Java, objects contain values (instance variables) and methods (instance methods) and can be seen as records [AC96]

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 *implicitely* 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, [MW97]. This vision is not of great help: It leads to programs weighted down from an indirect and tricky use of objects to refer to methods [BO04], 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 [BO05, BO06] 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 [BO04, Bri05]; *emulation* of a calculus of functions through anonymous inner classes of Java [Set03]; *extension* of the language, introducing special entities for function abstractions as in Pizza [MW97] or delegates as in J++ [Cor04]. 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 which limits the program expressivity and makes the use of HO programming a bit tricky. A different solution within the extension approach is discussed in [BO05, BO06] where the problem of extending the OO paradigm with method abstraction and method extraction is investigated, in the first order ζ-calculus [AC96], 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 *parameter method*. A parameter method can be interpreted as defining a function that maps objects (or classes) into instance (respectively, class) methods.
- Since they are functions, parameter methods 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 define the language extensions. In Section 4 we show the transformation $\mathcal{E}$ that maps programs of the extended language into programs of standard Java. The last section concludes the paper.

2 Code reusability: generalization vs. inheritance

In this section, we illustrate the reasons and the benefits for which we propose to extend the language with features for high order generalization and how these features are integrated with those for object oriented programming, mainly code reusability and inheritance. This is done discussing the development, in Java, of the code for the computation of the lists of the areas and of the perimeters of a collection of objects for geometric shapes, equipped with methods to compute the area and the perimeter of the shape [Hud00]. 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 forth class `FListShape` is an extension of `FList` equipped with two additional methods `FListArea` and `FListPerimeter`, since we want to compute the list of values that are areas and perimeters of a given list of shapes. The reader can obtain a definition of `FListPerimeter` by replacing `Area` with `Perimeter` in the invocation `(FList)L.Insert(((Shape)it.next()).Area())`. As it is clear, examining the code, in both cases 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 that allows to *generalize* invocation `((Shape).Val()).Area()` into `((Shape).Val()).F()`, where `F` is a functional variable, and that allows to bind `F` to a method, would provide an adequate support to that form of code development and reusability. This development is obtained, for instance, defining a higher order method, `Map`, with a parameter `F`. At each invocation of `Map`, the parameter `F` is bound to a *parameter method* that specifies which object operation, namely `Area` or `Perimeter`, is to be used for `F`. `Map` applies the operation bound to `F` to each object of the list, `Val().F()`, and returns the list of the computed objects. Given a `FList L`, the invocation of `L.Map(Abs Area)` computes like `L.FListArea()` while the invocation of `L.Map(Abs Perimeter)` computes like `L.FListPerimeter()`. Hence, the code for `Map` is reused for `FListArea`, `FListPerimeter()` and for any other method that can be obtained by instantiating the generalization introduced in the definition of `Map`. The solution in the extended language is shown in Figure 2: `FListShape` is still containing the two methods but these are mere invocations of `Map`. The possibility to support code development and reusability of the form used in the example is the main reason for resorting, in object oriented programming, to higher order mechanisms. Moreover, the use of *parameter method* instead of method in the invocation allows a perfect integration of these mechanisms with those of class hierarchy and inheritance. As a matter of fact, in the evaluation of `L.Map(Abs Area)`, for instance, the parameter `Abs Area` stands for the function that, given an object v, returns the *most specific* [GJSB00] method `Area` defined for the object v. Hence, 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

```
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 FList FListArea(){
           FList L= new FList();
           if (Val()!=null) {L=((FListShape)Tail()).FListArea();
                  L=(FList)L.Insert(((Shape) Val()).Area());}
           return L;}
    public FList FListPerimeter(){
           FList L= new FList();
           if (Val()!=null) {L=((FListShape) Tail()).FListPerimeter();
                  L=(FList)L.Insert(((Shape) Val()).Perimeter());}
           return L; }}
```

Figure 1: Example in Java

```
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=(FList)L.Insert(Val().F()));}
            return L; }}
public class FListShape extends FList{
     public FList FListArea() {return Map(Abs Area); }
     public FList FListPerimeter() {return Map(Abs Perimeter); }}
```

Figure 2: Example in the extended language

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.

3 Extension to the language

In this section we discuss the mechanisms that can be added to Java to support invocation and passing of parameter methods.

3.1 Method invocation

In the extension we propose, the syntax, for invocation, is modified in the following way:

Expression: *Expression.Identifier* (*[AExpression* {*, AExpression*}*]*)
| *Expression.Parameter* (*[AExpression* {*, AExpression*}*]*)

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

To pass methods as parameters and use such parameters in the invocation of a method, we need that:

- expressions which are actual parameters, namely *AExpression*, must be able to specify a parameter method. This is obtained introducing the constructor `Abs` that applies to an identifier and defines a function from (classes) objects into (possibly static) methods which have such an identifier as selector.
- expressions which are method invocations must be able to specify the invocation of a method parameter. This is syntactically expressed by adding the form *Expression.Parameter(...)*.

3.2 Method declaration

The syntax of the extended language for method declaration is:

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

To allow parameter methods in the list of formal parameters, we extend the syntax defining a new category, *FType*, which is used in place of *Type* to specify the type of the parameter. *FType* can be a *Type* or a newly defined type, identified by the type constructor `Fun` that specifies the type (types of the arguments and type of the result) of the methods that can be bound to the parameter.

3.3 Example

According to the new syntax, we can write, for instance:

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

in a class `C`. In this way, a method `A` is defined, which returns an integer and has two parameters `X` of type `B` and `F` which will be bound to a method which has an argument of type `int` and computes a value of type `int`. In the body of `A`, `F` is invoked by `X` and applies to 0. 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)` applies `A` to `b` and to a parameter method `M`. In the evaluation of the body of `A`, `F` is bound to `M`, which is the function mapping each object into the most specific method of the object that has name `M` and type `int` → `int`. The evaluation of `X.F(0)` becomes the invocation of the most specific method of `b` which has name `M` and type `int` → `int`. In fact, because of Java overriding many methods may exist with name `M` and type `int` → `int`. Hence, in a different definition of `A`, `F` could be applied many times, each time invoked by a possibly different object and each time the parameter method is computing a possible different most specific method. This well suits Java late binding semantics [GJSB05], which defers to invocation time the selection of the effective method a selector is referring to.

4 Implementation

4.1 Preprocessing: Structure of a syntactic translation

The implementation proposed in this paper uses a preprocessing technique which maps programs, in the extended syntax, back into programs of Java 1.4. A formal definition of the transformation is given by the definition of $\mathcal{E}$ in Figures 4 and 5. $\mathcal{E}$ applies, separately, to each class of the source program producing a corresponding class of an equivalent program in Java 1.4. The idea is to have two methods, `Apply` and `ApplyS`, to deal with method invocation when the invoked method is passed as a parameter. A string, namely ToS(Ide_i) is associated to each method selector, Ide_i, to allow evaluation and comparison.

The syntax of *ClassDef* to which $\mathcal{E}$ applies is given in Fig. 3. $\mathcal{E}$ is compositional, descending on the syntactic structure of the class, and indexed by the additional parameter ρ. ρ is an environment that contains the bindings name of the parameter method and `FType` of the parameter. This is necessary for type casting and to limit syntactic

transformations to the cases in which the method is effectively passed as a parameter method. Three methods are added to each class definition, they are `Apply`, `ApplyS` and `Dispatcher`:

- Method `Apply` and `ApplyS` are used to transform method invocation when the invoked method has been passed as a parameter method. In Java style, an interface `ApplyClass` is defined in the following way:

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

  with `Apply` and `ApplyS` as two abstract methods. They both have two parameters:

  1. `M` (of type `String`) is the name of the method to invoke,
  2. `Par` (of type `Object`) is the parameter of the method to invoke. `Par` is cast to the type of the argument of the method taken from ρ

  `Apply` and `ApplyS`, by means of a private method `Dispatcher` simply invoke the method whose name is equal to `M` 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, i.e. methods whose invocation is a statement. In this case `ApplyS` simply invokes the method. No type cast is required since both methods, `ApplyS` and the invoked one, do not return any value.

- A method `Dispatcher` is added which computes, given a string, the position in the class definition of the method whose name is equal to the given string.

In this solution type errors due to invocation of a method, passed as parameter, which does not exist are detected at run time, in this case `Apply` throws a `MethodNotFoundException`. A different solution would restrain considerably the mechanism usability, in particular it is necessary to require that the formal method parameters specify the top of the class hierarchy which the method must belong to, besides specifying the method descriptor. In such case also overloaded methods could be passed as parameters while now passing overloaded method is forbidden.

To restrain the presentation and make it more readable and comprehensible, $\mathcal{E}$ definition considers only methods with one parameter, it does not consider hierarchy of classes and static methods.

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`. In the example there are methods which have one parameter and methods which have no parameters. The simplified version of $\mathcal{E}$ does not allow to express

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` *Ide*

Where: *FParameters* stands for *FormalParameters*.

Figure 3: Extended syntax [GJSB00]

the transformed program in this latter case, since only methods with one parameter are considered. So the general transformation must be adopted. It passes to `Apply` (or `ApplyS`) an array of `Objects` in which each element is cast to the relative type taken from ρ. If the method has no argument the empty array is passed. All environments ρ produced are empty except for class `FList` as specified.

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` with such element (of the class `Circle`) as invoking object, and `F` whose value is the string `"Area"`, causes the invocation of the method `Area` of the class `Circle`. On the other hand, in the second recursive invocation a rectangle is obtained from `Val` invocation. Hence applying `Apply` with such element (of the class `Rectangle`) as invoking object and `F` whose value is still `"Area"` causes 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 value for the parameter `F`. This would not be possible if `F` value were a method, but more important, this is the behaviour required to fully exploit higher order programming in a context of a language with a type system based on type hierarchy and subtyping.

Let ClassDef ≡ `public class A {`
ModifiersOpt $Type_0$ Ide_0 *[=*Exp_0*]*;
...
ModifiersOpt $Type_h$ Ide_h *[=*Exp_h*]*;
ModifiersOpt A $(FParameter_{C_0})Block_{C_0}$
...
ModifiersOpt A $(FParameter_{C_k})Block_{C_k}$
ModifiersOpt $Type_{M_0}$ Ide_{M_0} $(FParameter_{M0})$ $Block_{M0}$
...
ModifiersOpt $Type_{M_k}$ Ide_{M_k} $(FParameter_{M_k})$ $Block_{M_k}$
ModifiersOpt `void` $Ide_{M_{k+1}}$ $(FParameter_{M_{k+1}})$ $Block_{M_{k+1}}$
...
ModifiersOpt `void` Ide_{M_n} $(FParameter_{M_n})$ $Block_{M_n}$

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

```
    private static int Dispatcher(String S){int pos=-1;
         if (S.equals(ToS(Ide_M0))) pos=0;
         else if ...
         else if (S.equals(ToS(Ide_Mk))) pos=k;
         else if (S.equals(ToS(Ide_Mk+1))) pos=k+1;
         else if ...
         else if (S.equals(ToS(Ide_Mn))) pos=n;
         return pos;}
    public Object Apply(String M, Object Par) throws MethodNotFoundException{
         int pos=Dispatcher(M);
         switch (pos){case 0   :  return Ide_M0(Par);
                     ...
                     case k   :  return Ide_Mk(Par);}
         throw new MethodNotFoundException();}
    public void ApplyS(String M, Object Par) throws MethodNotFoundException {
         int pos=Dispatcher(M);
         switch (pos){case k+1 :  Ide_Mk+1(Par);
                     ...
                     case n   :  Ide_Mn(Par);}
         throw new MethodNotFoundException();}}
```

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

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

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

$\mathcal{R}[\![FP]\!]_\rho(\text{ide}) = \langle \texttt{Fun}\ \mathit{FType}\ \mathit{Type}\rangle$ if $FP = \texttt{Fun}\ \mathit{FType} \rightarrow \mathit{Type\ Ide}$
$\mathcal{R}[\![FP]\!]_\rho(\text{ide}) = \langle \texttt{Fun}\ \mathit{FType}\ \mathit{Type}\rangle$ if $FP = \texttt{Fun}\ \mathit{FType} \rightarrow \texttt{void}\ \mathit{Ide}$
$\mathcal{R}[\![FP]\!]_\rho(x) = \rho(x)$ if $FP = \mathit{Type\ Ide}$

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

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

$$\mathcal{E}[\![St]\!]_\rho = \left| \begin{array}{ll} \mathcal{E}[\![Exp_1]\!]_\rho.\texttt{ApplyS(}\mathit{Par},(\mathit{FType})\mathcal{E}[\![Exp_2]\!]\texttt{))}, & \text{with } \mathit{St} = \mathit{Exp}_1.\mathit{Par(Exp}_2) \wedge \\ & \rho(\mathit{Par}) = \langle \texttt{Fun}\ \mathit{FType} \rightarrow \texttt{void}\rangle \\ \mathcal{E}[\![Exp_1]\!]_\rho.\mathit{Ide}(\mathcal{E}[\![Exp_2]\!]), & \text{with } \mathit{St} = \mathit{Exp}_1.\mathit{Ide(Exp}_2) \wedge \\ & \rho(\mathit{Ide}) = \bot \\ \texttt{if}(\mathcal{E}[\![Exp]\!]_\rho)\mathcal{E}[\![St_1]\!]_\rho\ \texttt{else}\ \mathcal{E}[\![St_2]\!]_\rho; & \text{with } \mathit{St} = \texttt{if}\ \mathit{Exp}\ \mathit{St}_1\ \textbf{else}\ \mathit{St}_2 \\ \texttt{while}(\mathcal{E}[\![Exp]\!]_\rho)\mathcal{E}[\![St]\!]_\rho & \text{with } \mathit{St} = \texttt{while}\ \mathit{Exp}\ \mathit{St} \\ \text{etc.} & \end{array} \right.$$

$$\mathcal{E}[\![Exp]\!]_\rho = \left| \begin{array}{ll} (\mathit{Type})\mathcal{E}[\![Exp]\!]_\rho.\texttt{Apply(}\mathit{Par}, (\mathit{FType})\mathcal{E}[\![Exp]\!]\texttt{)}, & \text{with } \mathit{Exp} = \mathit{Exp.Par(Exp)} \wedge \\ & \rho(\mathit{Par}) = \langle \texttt{Fun}\ \mathit{FType} \rightarrow \mathit{Type}\rangle \\ \mathcal{E}[\![Exp]\!]_\rho.\mathit{Ide}(\mathcal{E}[\![Exp]\!]), & \text{with } \mathit{Exp} = \mathit{Exp.Ide(Exp)} \wedge \\ & \rho(\mathit{Ide}) = \bot \\ \mathcal{E}[\![Exp]\!]_\rho\ \mathit{Op}\ \mathcal{E}[\![Exp]\!]_\rho & \text{with } \mathit{Exp} = \mathit{Exp\ Op\ Exp} \\ \mathit{etc.} & \end{array} \right.$$

Where:

- *Exp* stands for *Expression*, *St* stands for *Statement*, *StList* stands for *StatementList*, *FParameters* stands for *FormalParameters* and *Ide* stands for *Identifier*
- ToS(*Ide*) is a function which computes the string corresponding to the identifier Ide it is applied to.

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

```
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=(FList)L.Insert(Val().Apply(F,{})); }
            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){
            int pos=Dispatcher(M);
            switch (pos){   case 0:return Insert(Pars[0]);
                            case 1:return Val();
                            case 2:return Tail();
                            case 3:return Map(Pars[0]);}
            return new MethodNotFoundException();}
```

the environment ρ produced in which `Map` is transformed is $\rho = \{\langle$ F, `Fun` *Object* $\rangle\}$

Figure 6: Class FList

```
public class Circle 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){
            int pos=Dispatcher(M);
            switch (pos){  case 0:return Area();
                           case 1:  return Perimeter();
            return new MethodNotFoundException();}
```

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){
            int pos=Dispatcher(M);
            switch (pos){ case 0:  return FListArea());
                          case 1:  return FListPerimeter());
            return new MethodNotFoundException();}
```

Figure 8: FListShape

5 Conclusions

In the paper we have sketched an extension to Java that allows passing methods as parameters to other methods. The approach is based on the notion of *parameter method*, `Fun` *FType* $\rightarrow$ *Type Ide*, which introduces functions that map each object (class) of the program into the most specific method of the object (resp. class) having name Ide and type *FType* $\rightarrow$ *Type*. This notion offers a restricted, disciplined, form of function abstraction but suitable to the integration of high order and object oriented programming. Through examples we have shown the usefulness of such an extension. We have discussed an implementation of the proposed extension. The main advantages of this solution are that it is simple and well suited to the language type system, which is based on type hierarchy and subtyping. It is reasonably efficient, since it is done through a preprocessing step at compile time. Eventually it does not require any modification to JVM.

References

[AC96] M. Abadi and L. Cardelli. *A theory of objects*. Springer-Verlag, 1996.

[Bac78] J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communication ACM*, 21:613–641, 1978.

[BG01] B.Liskov and J. Guttag. *Program Development in Java. Abstraction, Specification and Object-Oriented Design*. Addison-Wesley, 2001.

[BO04] M. Bellia and M.E. Occhiuto. Higher order programming through java reflection. In *Concurrency, Specification and Programming CS&P'2004*, volume 3, pages 447–459, 2004.

[BO05] M. Bellia and M.E. Occhiuto. Higher order programming in java: Introspection, subsumption and extraction. *Fundamenta Informaticae*, 67(1):29–44, 2005.

[BO06] M. Bellia and M.E. Occhiuto. *From Object Calculus to Java with Passing and Extraction of Methods*. University of Pisa, Dipartimento Informatica, 2006.

[Bri05] B. Bringert. Hoj - higher-order java, 2005. http://www.cs.chalmers.se/ bringert/hoj/.

[Cas97] G. Castagna. *Object-Oriented Programming: A Unified Foundation*. Birkhauser Verlag AG, 1997.

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

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification - Second Edition*. ADDISON-WESLEY, 2000.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification - Third Edition*. ADDISON-WESLEY, 2005.

[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.

[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.