*Fundamenta Informaticae 93 (2009) 45–64*
*DOI 10.3233/FI-2009-87*
*IOS Press*

# Java$\Omega$: The Structures and the Implementation of a Preprocessor for Java with m_ and mc_parameters

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

**Abstract.** M_parameters extend Java by allowing methods to have methods as parameters. [8] furnishes a semantics of m_parameters and applications in OO programming. In this paper, we present an implementation of the extended language based on program preprocessing. We also discuss the integration of the extended programs with ordinary Java programs, and hence Java API. Furthermore, mc_parameters are defined: they are a variant of m_parameters for which the class hierarchy of the method passed as parameter must be provided in the formal and actual parameter. Semantics for mc_parameters is given but, in this case, an implementation with callbacks [20] is proposed. Eventually, we discuss how mc_parameters deal with overloaded methods.

## 1. Introduction

In [8] Java is extended with mechanisms which allow methods to have other methods as parameters. In that paper we also argued the improvement of the expressive power of languages including such kind of mechanisms, in particular code reusability and as a consequence code correctness. The extended language semantics is defined through a meaning preserving $\mathcal{E}[\![]\!]_\rho$ transformation, which maps extended programs into programs of ordinary Java (version 1.4).

In this paper, section 2, we describe an implementation of $\mathcal{E}[\![]\!]_\rho$ [6]: it is designed as a one-pass preprocessor [1], developed using Lex & Yacc [21] and GNU Bison [11]. The system is here forth called Java$\Omega$. Section 3 shows an example of a program in the extended language, here forth called Java$^\omega$, which uses Java API in order to show how ordinary Java programs can be integrated with Java$^\omega$ programs. This is possible even though standard Java API has not been preprocessed by Java$\Omega$, hence

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

methods there defined cannot be passed as parameters and are not higher order. A comparison of our approach with other similar approaches [23, 25, 10] can be found in [5]. Recently a number of proposals have been presented to add higher order features to Java [4, 12, 2, 19, 15, 13, 3]: all such proposals are concerned with the closure addition. A discussion and comparison of the different proposals is contained in [9] in which we also consider the inclusion of closures in Java$^\omega$ and consequently in Java$\Omega$. Among the others [12] also considers the possibility to have methods that are values of the language, hence can be returned by a method invocation, assigned to variables and passed as parameters, the latter one is the same functionality addressed in this paper. In fact, [12] introduces two different ways to select methods: *method literals* express methods as objects of `java.lang.reflect.Method` at run time, while *method references* express methods as anonymous instances of single method interfaces or abstract classes generated at compile time. Actually, i) method references are type safe and have the same behavior as mc_parameters, while method literals are not type safe, as m_parameters, but their use constraints programs as method references do. ii) Syntax for the invocation of method references is not transparent, since it requires the use of a special method *invoke* according to the implementation of method references. iii) The implementation of method references and mc_parameters is similar but [12] defines a distinct interface for each method reference, while we define only one interface which is used for all mc_parameters. iv) Method literals and method references have a non standard semantics since they have a different meaning but share the same syntactic structure and the distinction between the two forms depends on the context in which they occur. The advantage of our approach, see also [8], is that: i) it offers a restricted, disciplined, form of function abstraction which is suitable to the integration of higher order and object oriented programming. ii) It furnishes a neat, standard, semantics for the new mechanisms of m_ and mc_parameters. iii) The use of m_ and mc_parameters is completely transparent to implementation. iv) The syntax of the invocation of m_ and mc_parameters is uniform with ordinary Java method invocation. In addition, programs in Java$^\omega$ can be defined using all object-oriented mechanisms of Java: that is higher order classes can be defined as extension of classes defined in the API, adding higher order methods and implementing abstract first order methods using higher order methods. In section 4, we extend Java$^\omega$ with mc_parameters already introduced in [8]. Syntax, semantics and $\mathcal{E}[\![]\!]_\rho$ transformation are defined to deal with mc_parameters. The interesting point here is that the $\mathcal{E}[\![]\!]_\rho$ transformation defined, generates Java programs with callbacks [20]. In section 5, a comparison between m_ and mc_parameters is given, showing the example of Section 3 developed using mc_parameters and the transformed program, using callback. Eventually, mc_parameters are extended to deal with overloaded methods.

## 2. Implementation

### 2.1. Language extensions

The main extensions to Java are concerned with formal and actual parameters to include methods as parameters and with the modifier `functional` for classes whose methods may be passed as parameters in the program. If a class is declared `functional` then Java$\Omega$ transforms this class adding the code which allows to pass its methods as parameters to higher order methods. Otherwise, only the higher order methods definitions and invocations, occurring in the class are transformed.

The Java grammar rules directly affected are listed below:

| | | |
|---|---|---|
| *FModifier* | ::= | *Modifier* \| `functional` |
| *FType* | ::= | *Type* |
| | | \| *FunOrStaticFun FTList* $\rightarrow$ *TypeOrVoid* |
| *FTList* | ::= | *[FType (, FType)*]* |
| *AExp* | ::= | *Argument* |
| | | \| `Abs` *Identifier* |
| *FunOrStaticFun* | ::= | `Fun` \| `StaticFun` |
| *TypeOrVoid* | ::= | *Type* \| `void` |

where, `functional`, `Fun`, `StaticFun` and `Abs` are new tokens, and *FModifier*, *FType* and *AExp* are the new syntactic categories for *Modifier*, *Type* and *Argument*, respectively, defined in the Java 1.4 grammar [17].

All these syntactic extensions are captured by the preprocessor Java$\Omega$ as translation directives for a source to source textual manipulation of the Java$^{\omega}$ programs that produces equivalent programs written in ordinary Java 1.4. The nature of this equivalence is discussed in [8] where the meaning preserving transformation $\mathcal{E}[\![]\!]_{\rho}$ is formally defined on the basis of a semantic interpretation that associates to each:

(i) Argument `Abs` $g$, the partial function $A_g(s,t,c)$ below:

$$\lambda s\!:\!S \rightarrow \lambda t\!:\!T \rightarrow \lambda c\!:\!C \rightarrow \mathcal{M}(g,s,t,c)$$

that, given $s$, $t$, $c$, selects the method $\mathcal{M}(g,s,t,c)$, i.e. the method named $g$ in class $c$, with signature $s$ and result type $t$, if any.

(ii) Parameter $p$ in `Fun` $s \rightarrow t$ $p$, together with the argument `Abs` $g$ supplied for it, the partial function $P_{pg}(c)$ below:

$$\lambda c\!:\!C \rightarrow \mathcal{M}(g,s,t,c)$$

that, given a class $c$, selects method $\mathcal{M}(g,s,t,c)$.

(iii) Occurrence *e.p*(*e*$_1$,...,*e*$_n$) in the body of a method invoked with argument `Abs` $g$ supplied for parameter `Fun` $s \rightarrow t$ $p$, the invocation of method $\mathcal{M}(g,s,t,c)$, if any, where $c$ is the most specific class of the object computed by $e$ (or, the computed class, if $e$ computes a class: in this case $\mathcal{M}(g,s,t,c)$ is a class method), and $s$ is a correct signature for the types of $e_1,...,e_n$. If $\mathcal{M}(g,s,t,c)$ does not exist then the evaluation of *e.p*(*e*$_1$,...,*e*$_n$) abrupts and a `MethodNotFoundException` is thrown.

## 2.2. Preprocessing structures

The preprocessor Java$\Omega$ is a plan implementation of $\mathcal{E}[\![]\!]_{\rho}$ [8]. It provides for a complete traversal of the source program in order to locate 1) the program classes, 2) the declarations of the higher order methods and 3) the invocations of m_parameters.

### 2.2.1. The class modifier `functional`.

From a syntactic point of view, the tag `functional` is a modifier that is reserved for class declaration. It has been introduced, in Java$\Omega$, to explicitly declare the classes containing methods that can be passed, as parameters. This means that only the methods of a functional class are m_parameters and hence, need a double mechanism of method invocation. In effect, they can be invoked through the use of a m_parameter invocation, in addition to the ordinary method invocation. The modifier `functional` avoids the useless overhead of adding code, for such kind of double invocation, to all the methods of the classes in the source program, confining it to the methods of the classes declared `functional`.

When Java$\Omega$ encounters a class declared `functional`:

(1) it extends the source class header with the clause `implements ApplyClass`.

(2) it traverses all the class members, collects, in the list MethodHeader, the header of each (class and object) method of the class, and produces a source to source textual manipulation of the body of the methods as described in 2.2.2.2 and 2.2.3.3.

(3) it adds code to the class in order to allow each method of MethodHeader to be invoked as m_parameter. Following $\mathcal{E}[\![]\!]_\rho$ [8], this code consists of the definition of the methods of the interface `ApplyClass`. They are `StaticApply` and `StaticApplyS` for non void and void class methods, respectively, and `Apply` and `ApplyS` for non void and void object methods. These methods share great part of the syntactic structure of the body, differing for the kind (class void, class non void, object void, object non void) of the methods they deal with. The common part consists of a `switch` statement mapping the method internal names into invocations of the corresponding method with the right list of parameters. The preprocessor produces the code for the definition of the private method `Dispatcher` that maps the strings, naming the methods, into their internal names: the internal name is assigned according to the position of the method within the list MethodHeader. Eventually, the preprocessor splits MethodHeader into four sublists one for each of the four kinds of methods. If one of the sublist is an empty list, the body of the corresponding method of `ApplyClass` is: `throw new MethodNotFoundException()`. Once the common part has been produced, the preprocessor completes the code of:

  - `StaticApply` and `Apply`, by enclosing the invocation previously generated, in each case statement of `switch`, within a return statement;
  - `StaticApplyS` and `ApplyS`, with a break statement at the end of each case statement of `switch`.

### 2.2.2. The m_parameter: `Fun` and `StaticFun` declarations.

From a syntactic point of view, the tags `Fun` and `StaticFun` are keywords that prefix a pair of items, $s$ and $t$, that are separated by an arrow, and an identifier $p$. The preprocessor can find a structure of this kind only when it is traversing the parameter list of a method declaration. The declared method is a *higher order* method: The two items declare the method signature $s$ and the returned type $t$, if any, of the m_parameters that can be passed to the higher order method. The identifier $p$ is the name of the formal m_parameter. Eventually, the m_parameter invocations of $p$, in the body of the higher order method, behave as invocations of object, resp. class, methods according to the tag `Fun`, resp. `StaticFun`, in the parameter list declaration and are preprocessed as described in 2.2.3.3.

When Java$\Omega$ encounters the declaration of a m_parameter *FunOrStaticFun TList* $\rightarrow$ *TypeOrVoid par*:

(1) it adds to the list MParameter a record for the identifier $par$. The record contains one entry for the value of *FunOrStaticFun*, another for $FTList$, and a last one for *TypeOrVoid*. The list implements the enviroment $\rho$ of $\mathcal{E}[\![]\!]_\rho$ [8] and it maintains the scope of m_parameters in the body of higher order methods.

(2) it replaces the declaration *FunOrStaticFun FTList* $\rightarrow$ *TypeOrVoid par* of the m_para-meter with `String` *par*

### 2.2.3. Invocation of m_parameters.

From a syntactic point of view, the invocation of a m_parameter involves two distinct structures: one to express the actual m_parameter which is bound to the formal m_parameter, and another for the invocation of the formal m_parameter. The tag `Abs` prefixes an identifier $metName$ which specifies the name of the method that can be passed to the higher order method. The preprocessor can find a structure of this kind only when it is traversing the list of the actual parameters of a method invocation and the invoked method is a higher order method whose formal parameter list has, in correspondence to `Abs` *metName*, an m_parameter as in 2.2.2.2. The invocation of a formal m_parameter has a syntax which is not different from ordinary method invocation, namely *exp sel args* for an expression *exp*, an identifier *sel* and a list of actuals *args*, but the invocation is in the scope of a formal m_parameter that has the same identifier *sel* that occurs in the invocation.

When Java$\Omega$ encounters an expression defining an actual m_parameter `Abs` *metName*, it simply removes the tag `Abs` and it checks the declaration of the invoked higher order method for the correspondence of the parameter with a formal m_parameter.

When Java$\Omega$ encounters an invocation *exp sel args*, it checks the current list MParameter for the occurrence of a m_parameter of name *sel*, and in the affirmative case it extracts from the list the record $[FunOrStaticFun, FTList, TypeOrVoid]$, then:

(1) it produces a text $Selector$=*A(sel,arg)* for the method invocation $A$, where:

- $A$ is `ApplyS`, resp. `Apply`, if $TypeOrVoid$ is `void`, resp. a type, and *FunOrStaticFun* is `Fun`, otherwise $A$ is `StaticApplyS`, resp. `StaticApply`, if $TypeOrVoid$ is `void`, resp. a type, and *FunOrStaticFun* is `StaticFun`;
- $sel$ is the name of the invoked m_parameter
- $arg$ is the text for the array of arguments described in (3) below.

(2) it preprocesses the expressions in the list *args* of the invocation arguments by obtaining the list $args'$=$\{(t_1)exp_1, ..., (t_n)exp_n\}$ for $n \geq 0$ ($args'$=$\{\}$ for $n = 0$), where each expression in the list $args'$ is cast to the corresponding type of the list $FTList$ of the types expected for the arguments of the m_parameter invocation;

(3) it produces a text $arg$=`new Object[]`$\{(t_1)exp_1, ..., (t_n)exp_n\}$, where the members of $args'$ become elements of an array of arguments.

(4) it preprocesses expression $exp$ by obtaining expression $exp'$: if $exp$ is the empty text then $exp'$=`this`. Then it produces the text $Primary$="((`ApplyClass`)$exp'$)." if either *FunOrStaticFun* is `Fun` or $exp$ is not a class name, $Primary$="$exp'$." otherwise.

(5) eventually, it replaces the invocation *exp sel args* with the invocation which results from the concatenation of $Primary$ with $Selector$.

The preprocessor Java$\Omega$ is completely implemented in Lex&Yacc [21] and in GNU Bison [11]. It consists of one file Lex for generating a lexer of Java$^{\omega}$, of one file Bison for generating a parser of Java$^{\omega}$ and of auxiliary files for the language C procedures that implement the source to source translation described in the previous three sections, and for the computation process documentation. In defining the syntax of Java$^{\omega}$, for documentation sake, we choose to start from the official grammar of Java 1.4, distributed in [17]. The grammar has been suitably extended with the grammar rules of Section 2.1 and it results a LR(2) grammar [1]. Java$\Omega$ has been designed as an attribute grammar for one-pass preprocessor. Hence it produces the object code as it parses the source code, i.e. it evaluates the semantic rules during the parsing. These rules furnish also an integrated pretty printer that shows the object code in a structured and quite readable form.

## 3. Example with Java APIs

The example presented in this section, shows how higher order methods can be integrated with Java APIs, defining higher order classes (that are classes with higher order methods), which extend classes of Java APIs, in the same way as ordinary first order classes. All object-oriented mechanisms are available including overloading and overriding.

The example defines an extension of `LinkedList` namely `FList` and an implementation of `Comparable`, namely `HighComparable`. Such classes define higher order methods, in particular `HighComparable`, Fig.1, is an abstract class which implements interface `Comparable` adding a higher order method `compareTo`: It compares objects invoking the method, passed as parameter, on both the objects, to evaluate a value on which the objects are effectively compared. In the example shown, geometric shapes can be compared considering either their areas or their perimeters. This is obtained passing method `area` or method `perimeter` as parameter to the higher order method `compareTo`. Higher order `compareTo`, overloads first order `compareTo`, which is still undefined in `HighComparable`. It is implemented as an invocation of method parameter `m` on both `this` and `x`, the values returned are subtracted in order to evaluate 0 if such values are equal, a negative value if the value evaluated for `this` is minor than the value evaluated for `x`, positive otherwise. `FList` is defined in Fig.2 as an extension of `LinkedList`, in which the method `addOrd` is defined using higher order. Method `addOrd`, adds elements to the list supposing they are ordered and preserves such ordering. It uses the iterator (`listIterator`) of `LinkedList`. The arguments passed are: `x`, the element to be inserted and `m`, a method which is to be used to compare the elements in the list. The method invoked to compare the elements is the higher order method `compareTo`, defined in the abstract class `HighComparable`. The method parameter `m` passed to `addOrd` is passed to higher order `compareTo`. The structure of the method `addOrd` is quite usual, it considers the case in which the list is empty, otherwise the iteration starts, a `try-catch` clause is necessary to trap `NoSuchElementException`. At each step an element in the list is compared (by means of `compareTo`) to `x` (the element to insert) when a greater element is found `x` is inserted before the current element. To complete the example the abstract class `Shape`, `Circle` and `Rectangle` subclasses of `Shape`, and the `main` method must be defined. Such definitions are shown in Fig. 3, with the exception of main where only the list construction cycle is shown.The abstract class `Shape` type of the elements to be inserted in the list, is defined as extending `HighComparable`, inherits higher order `compareTo` and defines first order `compareTo` (in the example it calls higher order `compareTo` passing `area` as method parameter).

## 4. Mc_parameters

In this section we extend Java$^{\omega}$ with mc_parameters with the aim to avoid generating a runtime exception in case the method passed does not exist, and because it deals with overloaded methods (see Sections 5.1)

### 4.1. Syntax and semantics of mc_parameters

Differently from m_parameters, mc_parameters require the specification of the root of the class hierarchy of all the classes to which the mc_parameter can apply. The syntax for *FType* is extended with the following productions, where *Identifier* is the root class name:

$$FType ::= \texttt{Fun}\ Identifier{:}\ \ FTList \longrightarrow TypeOrVoid$$

As a matter of fact, the root class name is specified also in the actual mc_parameter. This is done extending *AExp* with the following production, where the second identifier is the root class name:

$$AExp ::= \texttt{Abs}\ Identifier\ \texttt{from}\ Identifier$$

All these syntactic extensions yield the extended Java grammar of Fig. 10. The mc_parameter semantics is different from m_parameter semantics. In this case the semantics interpretation associates to:

(i) Argument `Abs` $g$ `from` $c_a$, the partial function $A_{gc_a}(c_f, s, t, c)$ below:

$$\lambda c_f{:}C \rightarrow \lambda s{:}S \rightarrow \lambda t{:}T \rightarrow \lambda c{:}C \rightarrow \mathcal{M}(g,s,t,c)\ \textbf{if}\ c \preceq c_f \preceq c_a \wedge \mathcal{M}(g,s,t,c_a) \neq \bot$$

that, given $c_f$, $s$, $t$, $c$, selects the method $\mathcal{M}(g,s,t,c)$, i.e. the method named $g$ in class $c$, with signature $s$ and result type $t$: if $\mathcal{M}(g,s,t,c) \neq \mathcal{M}(g,s,t,c_a)$ then method $\mathcal{M}(g,s,t,c)$ is an overriding of method $\mathcal{M}(g,s,t,c_a)$ of class $c_a$. If $\mathcal{M}(g,s,t,c_a)$ does not exist then the program is not legal.

(ii) Parameter `Fun` $c_f$: $s \rightarrow t\ p$, together with the argument `Abs` $g$ `from` $c_a$ supplied for it, the partial function $P_{pg}(c)$ below:

$$\lambda c{:}C \rightarrow \mathcal{M}(g,s,t,c)\ \textbf{if}\ c \preceq c_f \preceq c_a \wedge \mathcal{M}(g,s,t,c_a) \neq \bot$$

that, given a class $c$, subclass of $c_f$, selects method $\mathcal{M}(g,s,t,c)$, provided that the class $c_f$ is a subclass of the class $c_a$.

(iii) Occurrence *e.p*($e_1$,...,$e_n$) in the body of a method invoked with argument `Abs` *g* `from` $c_a$ supplied for parameter `Fun` $c_f$: *s* $\rightarrow$ *t p*, the invocation of method $\mathcal{M}(g, s, t, c)$ where $c$ is the most specific class of the object computed by $e$ (or, the computed class, if $e$ computes a class: in this case $\mathcal{M}(g, s, t, c)$ is a class method), and $s$ is a correct signature for the types of $e_1, ..., e_n$.

The semantics states that, analogously to m_parameters, the invoked method belongs to class *c* of the object on which the method is invoked. The additional conditions specified for mc_parameters, force: *i)* the class *c* to be a subclass of $c_f$ which must be a subclass of $c_a$; *ii)* class $c_a$ to contain a method with name *g*, signature *s* and result type *t*. These two conditions can be checked at compile time. Such a check guarantees that the evaluation of *e.p*($e_1$,...,$e_n$) can never abrupt, at run time, because of a failure due to the fact that the selected method $\mathcal{M}(g, s, t, c)$ does not exist.

The transformation defined in this paper for mc_parameters is based on computational structures which are different from those in [8], since mc_parameters are implemented using callback [16].

Callback is a way to pass executable code to procedures, hence to define higher order programs. In OO languages callback is implemented through function objects [22, 20]. The idea comes from the observation that everything can be modeled using objects. Hence classes are defined to use objects as they were functions. In Section 4.2 we show a methodology which allows to write, in Java, higher order methods using callback. The transformation defined in Section 4.3 is based on such a methodology.

## 4.2. Callback methodology

The methodology is based on the construction of *function objects*, i.e. objects to be invoked as if they where functions. The callback methodology is here described when the functions are actually methods. It can be summarized in four points:

1. An interface representing (non void object) methods is here called `ApplyClass`. It has only one method `Apply` whose first parameter is an object, namely the function object wrapping the method to be invoked, while the second parameter is the array containing the method parameters, i.e. the parameters on which the method, to be invoked, applies.

```
public interface ApplyClass { //for non void object methods
  public abstract Object Apply(Object o, Object [] Pars) ;}
```

2. For each method which is to be passed as parameter a class of function objects, which implements `ApplyClass` and consequently the method `Apply`, must be defined. `Apply` invokes the wrapped method on the object passed to it as first parameter with the arguments passed as second parameter. Let *C* be the *definition class* of methods named *m*$_1$,...,*m*$_k$, i.e. the class in which such methods are declared. Suppose those methods have to be passed as parameters to other methods in the program. Then $k$ classes must be defined: For each method named *m*$_i$, a class *m*$_i$ is defined as implementation of `ApplyClass`, as follows:

```
static class mi implements ApplyClass{
  public Object Apply(Object o, Object [] Pars){
    return ((C) o).mi(Pars[0],Pars[1],...Pars[hi]);}}
```

3. Every higher order method *m*$^h$ is defined having at least one `ApplyClass` parameter that is the object function that contains the method to invoke by means of `Apply`.

   ```
   public T m^h (...ApplyClass o...)
         {...  o.Apply(obj, new Object[]{a_0, a_1, .., a_{h_i}});...}
   ```

   where *obj* is the object on which *m*$_i$ must be invoked and $a_j$, for $j \in [0..h_i]$, are the $h_i + 1$ arguments of *m*$_i$.

4. The invocation of *m*$^h$ requires the construction of the function object containing the method

   ```
   ...  E.m^h(...new C.m_i()...)
   ```

   assuming that the class $m_i$ is defined as an inner class of its definition class *C*.

The methodology, described insofar, considers only non void, object, methods. For class methods and/or void object methods, we have to consider, in point (1), one different interface for each of such kinds of methods, for instance:

```
public interface ApplyClassS { //for void object methods
  public abstract void Apply(Object o, Object [] Pars) ;}

public interface ApplyClassStatic { //for non void class methods
  public abstract Object Apply(Object [] Pars) ;}

public interface ApplyClassStaticS { //for void class methods
  public abstract void Apply(Object [] Pars) ;}
```

All these interfaces differ one another for the signature and/or the returned value of the method `Apply`.

Eventually, different solutions can be considered for the definition, in point (2), of the classes of function objects. In the presentation given above we use inner classes but each of the following solutions could be right:

**Inner classes** of class *C*. This is the solution we have shown above, in exemplifying points (1)-(4). In this case, for each method, the definition of the class (of function objects) is inserted in class *C*. It has the advantage of maintaining the definition of the classes (of function objects) local to the definition class. It allows, among others, to look only at the class definition of the method in order to check if it can be used as a parameter anywhere else in the program.

**Anonymous inner classes** defined in the invocation of higher order methods. In this case, point (2) is suppressed while invocation in point (4) becomes:

```
...  E.m^h(...new ApplyClass(){
       public Object Apply (Object o, Object [] Pars)
            {return ((C)o).m_i(Pars[0],Pars[1],...Pars[h]);}}...)
```

It is a compact solution but the resulting program is verbose and awkward (reading and writing). Moreover, it leads to code duplication, if a given method is passed twice.

**Stand alone classes** outside class *C*. In this case, classes in point (2) are somewhere in the program and invocation in point (4) becomes:

```
...  E.m^h(...new m̃_i()...)
```

where $\tilde{m}_i$ is the reserved name for the class (of function objects) defined in point (2), for the method named $m_i$. In fact, because of the unicity property of class names, a namespaces management [1] would be required in this case, defining reserved identifiers for such classes to eliminate class name clashes in the program. A strategy similar to the one used in Pizza [24], [23], could be used. The drawback of this solution is that locality property is lost [8, 14] and we have to check the entire program in order to detect which methods are wrapped in the `Apply` method of a class (of function objects) and as a consequence, can be passed as a parameter.

## 4.3. $\mathcal{E}[\![\,]\!]_\rho$ transformation with callback

The transformation defined, Figg.11-12, can be considered an extension of the one defined in [8], but it is here presented in a self contained way. However the partial functions that the semantics associates to mc_parameters, and the computational structures change. Actually the functions:

- differ one another for *i)* the method that must be selected once the object to apply to and the types of the arguments of the invocation are known, and *ii)* the computational structure to apply the function to the object and to the argument values, while

- they share the computational structure *i)* to find the most specific method with that name and types of the arguments, *ii)* to apply the selected method to the object with the arguments of the invocation, that is the invocation of method `Apply` (of interface `ApplyClass`).

The computational structures of those functions are a sort of run-time support which is included in the classes of the transformed program and is used through suitable methods. In Figg.11-12 the rules of the transformation are given defining the classes of function objects as inner classes. Moreover, we discuss how the rules implement the callback methodology to transform programs in Java$^\omega$ into ordinary Java code:

- Argument `Abs` $g$ `from` $c_a$ is transformed (second rule of *AExp*) replacing it with `new` $c_a.g$`()`. In this way, the partial function $A_{gc_a}(c_g, s, t, c)$, see $(i)$ in Section 4.1, is implemented constructing a function object (of the class $g$ which is defined as an inner class of $c_a$) that contains a method `Apply` that wraps method $\mathcal{M}(g, s, t, c)$, i.e. method `Apply`, given an object $o$, with most specific class $c$, and an array of arguments $\{a_1, ..., a_n\}$, invokes method $\mathcal{M}(g, s, t, c)$ on $o$ with arguments $a_1, ..., a_n$. The cast, introduced in the rule of *ClassDef* (lines 10 and 14 - definitions of the classes of function objects in Fig. 11) guarantees that the existence of method $\mathcal{M}(g, s, t, c_a)$ in class $c_a$ is checked at compile time on the program transformed by $\mathcal{E}[\![\,]\!]_\rho$.

- Parameter `Fun` $c_f$*:s→t p* is transformed (second rule for *FType*) replacing it with `ApplyClass` *p*. In this way, the partial function $P_{pf}(c)$, see $(ii)$ in Section 4.1, is implemented by binding $p$ to the function object resulting from the transformation of an argument `Abs` *g* `from` $c_a$, as described above.

- The occurrence *e.p*($a_1$,...,$a_n$) in the body of a method with a parameter `Fun` $c_f$: *s* → *t p*, is transformed (first rule for *Exp*) replacing it with `(`*t*`)(`*p*`.Apply((`$c_f$`)`*e*`, new Object[] {`$a_1$,...,$a_n$`}))`. In this way, the invocation of method $\mathcal{M}(g,s,t,c)$, see $(iii)$ in Section 4.1, is implemented invoking method `Apply` of the function object bound to $p$ which, actually, results into the invocation of the wrapped method $\mathcal{M}(g,s,t,c)$ on $e$, with most specific class $c$, and arguments $a_1,...,a_n$. The cast, introduced in the first rule of the transformation of *Exp*, guarantees that the object resulting from the evaluation of $e$ is of a class $c$ which is subclass of $c_f$, specified in the formal parameter. This is concerned with the subclass relationship $c \preceq c_f \preceq c_a$ required in (i)-(ii) of the semantics of mc_parameters which guarantees that every time *e.p*($a_1$,...,$a_n$) is executed method $\mathcal{M}(g,s,t,c)$ exists.

The transformation $\mathcal{E}[\![]\!]_\rho$ for the case in which the classes of function objects are defined as anonymous inner classes is given in Fig. 13. In this case, it is not necessary to declare the classes as `functional`, see Section 2.2.1.1, to use their methods as parameters in the program. As a matter of fact, the class definition (of function objects) is given at the invocation of the higher order method. Hence the benefit of the implementation using anonymous inner classes, for which also methods already defined and not preprocessed, for instance API methods, can be passed as parameters.

## 5. Mc_parameters vs m_parameters: example

From a syntactic point of view m_parameters differ from mc_parameters since the latter specify, in addition to signature and return type, the class hierarchy which the passed method belongs to. This fact has as consequence that programs that invoke a method with an actual argument `Abs` *g* `from` $c_a$, must have that method declared with an mc_parameter `Fun` $c_f$: *s* → *t p* where $c_f \preceq c_a$. In this section we show the example described in section 3 using mc_parameters. `FList` is still defined as an extension of `LinkedList` with the higher order method `addOrd`. The two classes `HighComparable` and `FList` are defined in Fig. 4 and 5. Class `Shape` is defined in Fig. 6, together with classes `Circle` and `Rectangle` and the sketch of method `main`. Comparing it with listings in Figg. 1-2-3, which use m_parameters, we note that both methods `addOrd` and `compareTo` have an mc_parameter `Fun Shape:` → `Double m`. In particular, since invoked with argument `Abs area from Shape` (fourth line of listing in Fig. 6), method `compareTo` can be invoked only with methods named `area` and defined in subclasses of `Shape`. As a consequence, programming with this definition of `compareTo` is much more constraining than it would be with the one in Fig. 1 which uses m_parameter. On the other hand, with such definition it cannot happen that invocations of `compareTo` turn out to throw exceptions trying to invoke methods (bound to the mc_parameter) that do not exist. In Fig. 7, 8 and 9 the previous defined example is transformed by the $\mathcal{E}[\![]\!]_\rho$ transformation with callback defined in Section 4.3. `Circle` and `Rectangle` class definitions are the same as those in Fig. 3.

```
public functional abstract class HighComparable implements Comparable{
  public int compareTo(Fun -> Double m, Object s){
        Double a=(this.m()-(s.m())) ;
        if (a>0) return 1;
        else if (a==0) return 0; else return -1;}}
```

Figure 1. HighComparable definition

```
public class FList extends LinkedList{
  public void addOrd(Fun -> Double m, HighComparable x){
    if (!this.isEmpty()) {ListIterator i= this.listIterator(0);
      int j=0;
      try {
        while(i.hasNext()&&((((HighComparable)i.next()).compareTo(m,x))<= 0))j++;
        add(j,x); }
      catch (NoSuchElementException e){System.out.println(...);} }
    else add(0,x);}}
```

Figure 2. FList definition

## 5.1. Method overloading

Introducing mc_parameters, in Section 4, we mentioned the problem of passing overloaded methods. In fact, to deal with overloaded methods, in addition to name and to the belonging class, a signature must be specified in order to determine, univocally, a method (among those sharing such a name in the class). Moreover, on the basis of the signatures that the different overloaded methods have in the class, Java resolves overloading at compile time, selecting, in each invocation, the method to be invoked and leaves to method dispatch (see Section 15.12.4.4 in [18]) to resolve, at run time, possibly overridden implementations of the selected method. This way of dealing with methods allows overloaded methods to be passed as parameters on the following conditions: i) actual parameters must specify the class and the signature of the passed method; ii) $\mathcal{E}[\![]\!]_\rho$ must transform actual parameters in such a way that the compiler can univocally select the overloaded method that must be invoked. Hence in order to pass overloaded methods, mc_parameters must be extended to specify the method signature in the actual parameter, as follows:

$AExp$::= Abs $Identifier(FTList)$ from $Identifier$

As far as ii) is concerned a solution for $\mathcal{E}[\![]\!]_\rho$ is shown in Fig. 14, where the invocation of the wrapped method $Ide_m$ constraints the type of the argument casting it to the type specified in the signature. The solution given for mc_parameters is based on anonymous inner classes. The other ways to define classes for function objects, Section 4.2, require a specific name space management to assign to each class of object functions, created for the methods to be passed as parameters, a unique name which depends on the name, signature and belonging class of the method.

```
public functional abstract class Shape extends HighComparable {
  public abstract Double area();
  public abstract Double perimeter();
  public int compareTo(Object s){return compareTo(Abs area ,s);} }
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 static void main(String[] args){
      ...for (i=0; i<n;i++){...
                  Shape sh=readShape(in,x);
                  L.addOrd(Abs area,sh);
             ...  }}
```

Figure 3. `Shape`, `Circle`, `Rectangle` and `main` definition

```
public functional abstract class HighComparable implements Comparable{
  public int compareTo(Fun Shape:  -> Double m, Object s){
      Double a=(this.m()-(s.m())) ;
      if (a>0) return 1;
      else if (a==0) return 0; else return -1;}}
```

Figure 4. `HighComparable` definition for mc_parameter example

```
public class FList extends LinkedList{
  public void addOrd(Fun Shape:  -> Double m, HighComparable x){
   if (!this.isEmpty()) {ListIterator i= this.listIterator(0);
     int j=0;
     try {
      while(i.hasNext()&&((((HighComparable)i.next()).compareTo(m,x))<= 0))j++;
      add(j,x); }
     catch (NoSuchElementException e){System.out.println(...);} }
   else add(0,x);}}
```

Figure 5. `FList` definition for mc_parameter example

```
public functional abstract class Shape extends HighComparable {
  public abstract Double area();
  public abstract Double perimeter();
  public int compareTo(Object s){return compareTo(Abs area from Shape ,s);} }
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 static void main(String[] args){
      ...for (i=0; i<n;i++){...
                 Shape sh=readShape(in,x);
                 L.addOrd(Abs area from Shape,sh);
             ...  }}
```

Figure 6. Shape, Circle, Rectangle and main definition for mc_parameter example

```
public abstract class HighComparable implements Comparable{
  public int compareTo(ApplyClass m, Object s){
    Double a=(Double)(m.Apply((Shape)this, new Object[]{}))-
             (Double)m.Apply((Shape)s, new Object[]{}));
    if (a>0) return 1;
    else if (a==0) return 0; else return -1 ;} }
```

Figure 7. Transformed program for HighComparable definition

```
public class FList extends LinkedList {
  public void addOrd(ApplyClass m, HighComparable x){
  if (!(this.isEmpty())) {ListIterator i= this.listIterator(0);
    int j=0;
    try {while ((i.hasNext()) && (((HighComparable)i.next()).compareTo(m,x)<=0)) j++;
         add(j,x); }
    catch (NoSuchElementException e){System.out.println(...);}}
  else add(0,x) ;}}
```

Figure 8. Transformed program for FList definition

```
public abstract class Shape extends HighComparable {
  public abstract Double area();
  public abstract Double perimeter();
  public abstract String toString();
  public int compareTo(Object s){return compareTo(new area(),s);}
  static class area implements ApplyClass{
    public Object Apply(Object o, Object [] Pars){ return ((Shape)o).area();}}
  static class perimeter implements ApplyClass{
    public Object Apply(Object o, Object [] Pars){ return ((Shape)o).perimeter();}}}

public static void main(String[] args){
      ...for (i=0; i<n;i++){...
                Shape sh=readShape(in,x);
                L.addOrd(new Shape.area(),sh);...} ...}
```

Figure 9. Transformed program for `Shape` and `main` definition

*ClassDeclaration* ::= `public 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* `from` *Identifier*

Figure 10. Extended syntax [17]

```
Let ClassDef $\equiv$ public class $Ide_A$ {
                    $ModifiersOpt\ Type_0\ Ide_0\ [=Exp_0];\ \ldots ModifiersOpt\ Type_h\ Ide_h\ [=Exp_h];$
                    $ModifiersOpt\ A(Type_{C_0}\ Ide_{C_0})Block_{C_0}\ \ldots ModifiersOpt\ A(Type_{C_k} Ide_{C_k})Block_{C_k}$
                    $ModifiersOpt\ Type_{M_0}\ Ide_{M_0}\ (FType_{FP_{M_0}} Ide_{FP_{M_0}})\ Block_{M_0}$
                    $\ldots ModifiersOpt\ Type_{M_k}\ Ide_{M_k}\ (FType_{FP_{M_k}} Ide_{FP_{M_k}})\ Block_{M_k}$
                    $ModifiersOpt$ 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 $Ide_A$ {
    $ModifiersOpt\ Type_0\ Ide_0[=\mathcal{E}[\![Exp_0]\!]_\rho];\ \ldots ModifiersOpt\ Type_h\ Ide_h[=\mathcal{E}[\![Exp_h]\!]_\rho];$
    $ModifiersOpt\ A\ (Type_{C_0}\ \text{Ide}_{C_0})\mathcal{E}[\![Block_{C_0}]\!]_\rho\ \ldots ModifiersOpt\ A\ (Type_{C_k}\ \text{Ide}_{C_k})\mathcal{E}[\![Block_{C_k}]\!]_\rho$
    $ModifiersOpt\ Type_{M_0} Ide_{M_0}(\mathcal{E}[\![FType_{FP_{M_0}} Ide_{FP_{M_0}}]\!]_\rho)\mathcal{E}[\![Block_{M_0}]\!]_{\rho'_0}$
    $\ldots ModifiersOpt\ Type_{M_k} Ide_{M_k}(\mathcal{E}[\![FType_{FP_{M_k}}\ Ide_{FP_{M_k}}]\!]_\rho)\mathcal{E}[\![Block_{M_k}]\!]_{\rho'_k}$
    $ModifiersOpt$ 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 $Ide_{M_n}(\mathcal{E}[\![FType_{FP_{M_n}} Ide_{FP_{M_n}}]\!]_\rho)\mathcal{E}[\![Block_{M_n}]\!]_{\rho'_n}$ }
// inner classes of function objects wrapping the methods defined in the class
    static class $Ide_{M_0}$ implements ApplyClass{
        public Object Apply(Object $o$, Object [] $Pars$){
            return (($Ide_A$) $o$).$Ide_{M_0}$($Pars$[0])}}
            ...
    static class $Ide_{M_k}$ implements ApplyClass{
        public Object Apply(Object o, Object [] $Pars$){
            return (($Ide_A$) o).$Ide_{M_k}$($Pars$[0])}}
    }
```

Figure 11. Transformation $\mathcal{E}[\![]\!]_\rho$ - part 1

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

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

$$\mathcal{E}[\![\mathit{AExp}]\!]_\rho = \begin{cases} \mathcal{E}[\![\mathit{Expression}]\!]_\rho & \text{with } \mathit{AExp}= \mathit{Expression} \\ \texttt{new Ide}_A.\mathit{Ide}() & \text{with } \mathit{AExp}=\texttt{Abs } \mathit{Ide} \texttt{ from } \mathit{Ide}_A \end{cases}$$

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

$$\mathcal{E}[\![\mathit{St}]\!]_\rho = \begin{cases} \mathit{Par}.\texttt{ApplyS((}\mathit{Ide}_A)\mathcal{E}[\![\mathit{Exp}_1]\!]_\rho,\texttt{new Object} & \texttt{[]}\{(\mathit{FType})\mathcal{E}[\![\mathit{Exp}_2]\!]_\rho\}), \\ & \text{with } \mathit{St} = \mathit{Exp}_1.\mathit{Par}(\mathit{Exp}_2) \wedge \\ & \rho(\mathit{Par}) = \texttt{Fun } \mathit{Ide}_A\text{: } \mathit{FType} \rightarrow \texttt{void} \\ \mathcal{E}[\![\mathit{Exp}_1]\!]_\rho.\mathit{Ide}(\mathcal{E}[\![\mathit{Exp}_2]\!]_\rho), & \text{with } \mathit{St} = \mathit{Exp}_1.\mathit{Ide}(\mathit{Exp}_2) \wedge \\ & \rho(\mathit{Ide}) = \bot \\ \texttt{if}(\mathcal{E}[\![\mathit{Exp}]\!]_\rho)\mathcal{E}[\![\mathit{St}_1]\!]_\rho \texttt{ else } \mathcal{E}[\![\mathit{St}_2]\!]_\rho; & \text{with } \mathit{St} = \texttt{if } \mathit{Exp}\ \mathit{St}_1 \textbf{ else } \mathit{St}_2 \\ \texttt{while}(\mathcal{E}[\![\mathit{Exp}]\!]_\rho)\mathcal{E}[\![\mathit{St}]\!]_\rho & \text{with } \mathit{St} = \texttt{while } \mathit{Exp}\ \mathit{St} \\ \text{etc.} & \end{cases}$$

$$\mathcal{E}[\![\mathit{Exp}]\!]_\rho = \begin{cases} ((\mathit{Type})(\mathit{Par}.\texttt{Apply((}\mathit{Ide}_A)\mathcal{E}[\![\mathit{Exp}_1]\!]_\rho,\texttt{new} & \text{Object[]}\{(\mathit{FType})\mathcal{E}[\![\mathit{Exp}_2]\!]_\rho\}), \\ & \text{with } \mathit{Exp} = \mathit{Exp}_1.\mathit{Par}(\mathit{Exp}_2) \wedge \\ & \rho(\mathit{Par}) = \texttt{Fun } \mathit{Ide}_A\text{: } \mathit{FType} \rightarrow \mathit{Type} \\ \mathcal{E}[\![\mathit{Exp}_1]\!]_\rho.\mathit{Ide}(\mathcal{E}_2[\![\mathit{Exp}]\!]_\rho), & \text{with } \mathit{Exp} = \mathit{Exp}_1.\mathit{Ide}(\mathit{Exp})_2 \wedge \\ & \rho(\mathit{Ide}) = \bot \\ \mathcal{E}[\![\mathit{Exp}_1]\!]_\rho\ \mathit{Op}\ \mathcal{E}[\![\mathit{Exp}_1]\!]_\rho & \text{with } \mathit{Exp} = \mathit{Exp}_1\ \mathit{Op}\ \mathit{Exp}_1 \\ \mathit{etc.} & \end{cases}$$

where: $\rho'_i = \mathcal{R}[\![\mathit{FType}_{M_k} \mathit{Ide}_{M_k}]\!]_\rho$
$\mathcal{R}[\![\mathit{FType}\ \mathit{Ide}]\!]_\rho(x) = \mathit{FType}$ if *Ide* = *x*
$\mathcal{R}[\![\mathit{FType}\ \mathit{Ide}]\!]_\rho(x) = \rho(x)$ if $\mathit{Ide} \neq x$

and *Exp*, *St*, *StList*, *FParameters*, *Ide* stand for *Expression*, *Statement*, *StatementList*, *FormalParameters*, *Identifier* respectively;

Figure 12. Transformation $\mathcal{E}[\![]\!]_\rho$ - part 2

$$\mathcal{E}[\![\mathit{AExp}]\!]_\rho = \begin{cases} \mathcal{E}[\![\mathit{Expression}]\!]_\rho & \text{with}\mathit{AExp}{=}\ \mathit{Expression} \\ \texttt{new ApplyClass}\{ & \text{with}\mathit{AExp}{=}\texttt{Abs}\ \mathit{Ide}_m\ \texttt{from}\ \mathit{Ide}_A \\ \quad\texttt{public Object Apply(Object}\ \mathit{o}\texttt{, Object} & \texttt{[]}\mathit{Pars)}\{ \\ \quad\texttt{return ((}\mathit{Ide}_A\texttt{)}\mathit{o}\texttt{).}\mathit{Ide}_m\texttt{(}\mathit{Pars}\texttt{[0])}\}\} & \end{cases}$$

Figure 13. $\mathcal{E}[\![]\!]_\rho$ using anonymous inner classes

$$\mathcal{E}[\![\mathit{AExp}]\!]_\rho = \begin{cases} \mathcal{E}[\![\mathit{Expression}]\!]_\rho & \text{with}\mathit{AExp}{=}\ \mathit{Expression} \\ \texttt{new ApplyClass}\{ & \text{with}\mathit{AExp}{=}\texttt{Abs}\ \mathit{Ide}_m(\mathit{Type})\ \texttt{from}\ \mathit{Ide}_A \\ \quad\texttt{public Object Apply(Object o,} & \texttt{Object []}\mathit{Pars)}\{ \\ \quad\texttt{return ((}\mathit{Ide}_A\texttt{)}\mathit{o}\texttt{).}\mathit{Ide}_m\texttt{((}\mathit{Type}\texttt{)} & \mathit{Pars}[0])\}\} \end{cases}$$

Figure 14. $\mathcal{E}[\![]\!]_\rho$ for overloaded methods

## 6. Conclusions

In this paper we presented the implementation of the extended language Java$^\omega$ firstly defined in [8] through the rules system $\mathcal{E}[\![]\!]_\rho$. The system is a set of source to source translation rules that state the meaning of the new constructs in terms of compositions of well known ordinary Java structures and provide a one pass translation process of the programs of the extended language back into ordinary Java programs. Hence, the implementation is obtained through a source to source, one pass, preprocessor [6, 7], easy to write using standard development tools [1, 21, 11]. Then, we discussed the integration of programs written in Java$^\omega$ with programs written in ordinary Java. We showed that higher order classes can be defined as extension of classes defined in the APIs, adding higher order methods and implementing abstract first order methods using higher order methods. Then, we applied the same approch used in [8] to further extend the language with mc_parameters, as a variant of m_parameters, which specify the belonging class and the name of the passed method. We defined syntax, semantics of mc_parameters. Then, we showed a four steps callback methodology for wrapping methods into function objects and we extended transformation $\mathcal{E}[\![]\!]_\rho$ with rules translating programs with mc_parameters into ordinary Java programs using callback. It is worth noting that a transformation using callback cannot be defined for m_parameters, since callback requires the knowledge of the class the wrapped method belongs to. As a matter of fact, the contrary would be possible that is mc_parameters could be implemented using the same technique of the transformation given in [8] for m_parameters. Eventually, we compared programming with m_ and mc_parameters and we showed that mc_parameters are much more constraining than m_parameters but they allow (i) a static checking on existence of the passed method and (ii), extended with method signature, the use of overloaded methods as parameters.

## References

[1] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Tecniques, and Tools*. Addison-Wesley, 2007.

[2] G. Bracha, N. Gafter, J. Gosling, and P. von der Ahe. Closures for java, 2006. //blogs.sun.com/ahe/resource/closures.pdf.

[3] G. Bracha, N. Gafter, J. Gosling, and P. von der Ahe. Closures for the java programming language (aka bgga), 2008. www.javac.info.

[4] D. Lea B. Lee and J. Bloch. Concise instance creation expressions: Closure without complexity, 2006. crazybob.org/2006/10/java-closure-spectrum.html.

[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. JH-preprocessor, 2007. www.di.unipi.it/∼occhiuto.

[7] M. Bellia and M.E. Occhiuto. *JavaΩ: The Structures and the Implementation of a Preprocessor for Java with m_parameters*. Technical Report TR-08-22, Dipartimento Informatica, University of Pisa, 2008.

[8] M. Bellia and M.E. Occhiuto. Methods as parameters: A preprocessing approach to higher order in java. *Fundamenta Informaticae*, 85(1):35–50, 2008.

[9] M. Bellia and M.E. Occhiuto. *JavaΩ: Preprocessing Closures in Java*. Technical Report TR-09-03, Dipartimento Informatica, University of Pisa, 2009.

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

[11] C.Donnely and R. Stallman. Bison: The yacc-compatible parser generator, 2006. www.gnu.org/software/bison/manual.

[12] S. Colebourne and S. Shulz. First_class methods: Java style closures, 2006. docs.google.com/view?docid=ddhp95vd_6hg3qhc.

[13] S. Colebourne, S. Shulz, and R. Clarkson. Fcm+jca, 2008. docs.google.com/ View?docid=ddhp95vd_0f7mcns.

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

[15] N.M. Gafter. Jsr proposal: Closures for java, 2007. JavaCommunity Process, www.javac.info/consensus-closure-jsr.html.

[16] E. Gamma, R. Helm, R. Johnson, and J.M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2005.

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

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

[19] B. Goetz. The closures debate: Should closures be added to the java language, and if so, how?, 2007. Java Theory and Practice, IBM Technical Library, www.ibm.com/developerworks/java/library/j-jtp04247.html.

[20] C. Horstmann. *Big Java ,3$^{rd}$ ed.* Wiley Computing, 2007.

[21] J.R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. OŔelly, 1995.

[22] B. Meyer. The power of abstraction, reuse and simplicity: An object-oriented library for event-driven design. In *Essay in Memory of Ole-Johan Dahl 2004*, volume 2635 of *LNCS*, pages 236–271. Springer, 2004.

[23] M. Odersky, E. Runne, and P. Wadler. Two ways to bake your pizza - translating parameterised types into Java. In *Generic Programming 1998, Proceedings of a Dagstuhl Seminar,LNCS 1766.*, pages 114–132, 1998.

[24] 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.

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