JavaΩ: A Preprocessor for Java with m_parameters

Marco Bellia and M. Eugenia Occhiuto

Dipartimento di Informatica, Università di Pisa, Italy {*bellia,occhiuto*}*@di.unipi.it*

Abstract. In [BO08] an extension of Java is described which allows methods to have other methods as parameters and a meaning preserving transformation is defined, which maps programs of the extended language into ordinary Java programs. In this paper we present an implementation for such a language, based on the transformation defined. We also discuss the integration of the programs of the extended language with ordinary first order programs, and hence Java API . Eventually the language is extended with mc_parameters for which an implementation with callbacks [Hor07] is shown.

1 Introduction

In [BO08] 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 $\mathcal{E}$ transformation, which maps extended programs into programs of ordinary Java (version 1.4).

In this paper, section 2, we describe an implementation of $\mathcal{E}$ [BO07]: it is designed as a one-pass preprocessor [ALSU07], developed using Lex & Yacc [LMB95] and GNU Bison [CS06]. The system is here forth called JavaΩ. 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 JavaΩ, hence methods there defined cannot be passed as parameters and are not higher order. The advantage of our approach, also with respect to other similar approaches [ORW98,Set03,Bri08,SS07,BGGA08], is that 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 [BO08]. Syntax, semantics and $\mathcal{E}$ transformation are defined to deal with mc_parameters. The interesting point here is that the $\mathcal{E}$ transformation defined, generates Java programs with callbacks [Hor07], which is not possible for m_parameters. Eventually, in section 5, the example shown in section 3 is developed using mc_parameters and the transformed program, using callback, is shown.

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. If a class

is declared `functional` then JavaΩ transforms it 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*
		| (`Fun` | `StaticFun`) *FTList* $\rightarrow$ *Type*
		| (`Fun` | `StaticFun`) *FTList* $\rightarrow$ *void*
FTList	::=	*[FType (, FType)*]*
AExp	::=	*Argument*
		| `Abs` *Identifier*

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 [GJSB00].

All these syntactic extensions are captured by the preprocessor JavaΩ 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 [BO08] where the meaning preserving transformation $\mathcal{E}$ is formally defined on the basis of a semantic interpretation that associates to each:

(i) identifier f in a construct `Abs` f, a (partial) function

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

where S is the domain of Java signatures, T is the domain of Java types, C is the domain of Java classes. and $\mathcal{M}(f, s, t, c)$ is a method of c, having name f, signature s and result type t.

(ii) formal parameter p in a construct `Fun` $s \rightarrow t$ p, a function

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

(iii) invocation `e.p(`$\mathtt{e}_1$`,...,`$\mathtt{e}_n$`)` that involves a formal `p`, as in (ii), for which $\mathtt{e}_1$,...,$\mathtt{e}_n$ have the types of the signature s, an invocation of the method $\mathcal{M}(f, s, t, c)$ that the function bound to `p` computes once the evaluation of `e` results in a class c (or in a object v with most specific class c).

2.2 Preprocessing structures

The preprocessor JavaΩ is a plan implementation of $\mathcal{E}$ [BO08]. 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 JavaΩ, to explicitly declare which class contains 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.

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 two items declare the 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.

2.2.3 Invocation of m_parameters. From a syntactic point of view, the invocation of a m_parameter involves two distinct structures: one for expressing 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 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.

The preprocessor JavaΩ is implemented as an attribute grammar [ALSU07] in Lex & Yacc [LMB95] and in GNU Bison [CS06]. 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 main procedures that implement the source to source translation described in the previous three sections (see [**?**] for details). It has also an integrated pretty printer that shows the object code in a structured and quite readable form.

3 Example with Java API

The example presented in this section, shows how higher order methods can be integrated with Java API, defining higher order classes (that is classes with higher order methods), which extend classes of Java API, in the same way as ordinary first order classes. All object-oriented mechanisms are available including overloading and overriding, with the exception that overloaded methods cannot be passed as parameters.

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`.

`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

```
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;}}
```

Fig. 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);}}
```

Fig. 2. `FList` definition

```
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);
             ...  }
```

Fig. 3. `Shape`, `Circle` and `Rectangle` 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;}}
```

Fig. 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);}}
```

Fig. 5. `FList` definition for mc_parameter example

`HighComparable`. The method parameter `m` passed to `addOrd` is passed to higher order `compareTo`.

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, already introduced in [BO08] with the aim to avoid generating a runtime exception in case the method passed does not exist, and because it seems a promising approach to allow passing overloaded methods, in addition to ordinary methods.

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 the *Identifier* is the root class name:

FType::= Fun *Identifier*: *FTList* → *Type*
FType::= Fun *Identifier*: *FTList* → void

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

AExp::= Abs *Identifier* from *Identifier*

The mc_parameter semantics is different from the semantics of m_parameter. In this case the semantics interpretation associates to:

(i) identifier f in a construct `Abs` f `from` c_a, a (partial) function

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

where S is the domain of Java signatures, T is the domain of Java types, C is the domain of Java classes and $\mathcal{M}(f, s, t, c)$ is a method of c, having name f, signature s and result type t.

(ii) formal parameter p in a construct `Fun` c_f: $s \to t$ p, a function

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

(iii) invocation `e.p(`$\mathtt{e}_1$`,...,`$\mathtt{e}_n$`)` that involves a formal `p`, as in (ii), for which $\mathtt{e}_1$,...,$\mathtt{e}_n$ have the types of the signature s, an invocation of the method $\mathcal{M}(f, s, t, c)$ that the function bound to `p` computes once the evaluation of `e` results in a class c (or in a object v with most specific class c).

The semantics states that, analogously to the m_parameters, the invoked method belongs to class c of the object on which the method is invoked.The additional conditions specified for the mc_parameters, force: *i)* the class c to which the method belong to be a subclass of c_a which must be a subclass of c_f; *ii)* class c_a to contain a method with name f, signature s and result type t.

The transformation defined in this paper for mc_parameters is based on computational structures which are different from those in [BO08], since mc_parameters are implemented using the callback pattern [GHJV05]. Callbacks are widely considered a way to define function pointers in Java [Mey04,Hor07], that is a way to define higher order methods. The idea comes from the observation that everything can be modeled using objects. Hence a class is defined to refer to methods through objects. In subsection 4.2 we show a methodology which allows to write higher order methods using callbacks. The transformation defined in subsection 4.3 is based on such methodology.

4.2 Callback methodology

The methodology can be summarized in four points:

1. An interface representing methods is here called `ApplyClass`. It has only one method `Apply` whose first parameter is an object, the object containing the method to be invoked, while the second parameter is the array containg the method parameters.

```
public interface ApplyClass {
  public abstract Object Apply(Object m, Object [] Pars) ;}
```

2. For each method which is to be passed as parameter a class, which implements `ApplyClass` and consequently the method `Apply`, must be defined. `Apply` invokes the method on the object passed as first parameter with the arguments passed as second parameter. Suppose we have defined a class `C` with methods named $\mathtt{m}_1$,...,$\mathtt{m}_k$, which have possibly to be passed as parameters then k classes must be defined. For each method named $\mathtt{m}_i$, a class $\mathtt{m}_i$ is defined as implementation of `ApplyClass`, as follows:

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

3. Every higher order method `hm` is defined having at least an `ApplyClass` parameter that is the object containing the method to invoke by means of `Apply`.

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

   `obj` is the object on which $\mathtt{m}_i$ must be invoked and $\mathtt{a}_j, j \in 0, h_i$ are the $\mathtt{h}_i + 1$ arguments of $\mathtt{m}_i$.
4. The `hm` invocation requires the construction of the object containing the method

   ```
   ... E.hm(...new C.m_i()...)
   ```

Different solutions can be considered for the definition of the classes that must be defined for the methods passed as parameters as:

Inner classes of class `C`. In this case, for each method, the previous class definition is inserted in class `C`.

Anonymous inner classes defined during invocation, in the following way:

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

Stand alone classes outside class `C`. However, in this case, because of the unicity property of class names a namespaces management [ALSU07] would be required, defining reserved identifiers to eliminate ambiguities. A strategy similar to the one used in Pizza [OW97], [ORW98], could be used. The drawback of this solution is that locality property is lost [BO08,DNR06].

This methodology is sketched considering only non void object methods. For class methods and/or void object methods, we have to define a different interface for each of such kind of methods.

4.3 $\mathcal{E}$ transformation with callbacks

The transformation here defined, Fig. 11 and 12, can be considered an extension of the one defined in [BO08], but it is presented in a self contained way. However the functions that the semantics associates to mc_parameters, and the computational structures change. Actually the functions (see 4.1, (i)-(iii)):

- 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 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`).

Also in this case the computational structures of those functions are a sort of run-time support that is included in the classes of the transformed program and used through suitable methods. Differently from the other case we must:

- transform higher order method definitions substituting the mc_parameter type `Fun` *Ide: FType* $\rightarrow$ *Type* with `ApplyClass`.
- transform the actual mc_parameter substituting `Abs` *Ide* `from` *C* with `New` *C.Ide*`()`.
- define for each method, of a given class *C*, that is to be passed as parameter, a class implementing `ApplyClass` (or `ApplyClassStatic` etc. as described in section 4.2). Actually the transformation is given, in Figg. 10-12, considering only non void object.

We define $\mathcal{E}$, in Figg. 10-12, for the two solutions, where classes for methods are defined as inner classes of *C*, and as anonymous inner classes. In this second case, it is not necessary to declare the classes as functional to pass their methods as parameters, since the class definition 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. The drawback, on the other hand, is the code duplication (if a given method is passed twice) and the lack of readability of the generated code.

In the case of inner and stand alone classes, the root class *C* is used to construct the object containing the method *Ide*. For inner classes it is simply `new` *C.m*`()`. For stand alone classes the situation is more complicated since depends on the namespaces management. In the case of anonymous inner classes the root class name *C* is used to cast the object which the method is to be applied to.

5 Example with mc_parameter

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`. The difference with the example of m_parameters is that now a class must be declared in each higher order method invocation, while in its definition a superclass, of all the classes declared in all possible invocations, must be specified.

In Fig. 7, 8 and 9 the previous defined example is transformed by the $\mathcal{E}$ transformation with callback defined in section 4.3. Class `Circle` and `Rectangle` definitions are the same as those in Fig. 3.

6 Conclusions

In this paper we continue the work presented in [BO08], in particular, we present the implementation of the extended language Java$^\omega$ based on the $\mathcal{E}$ transformation there defined. We discuss the integration of programs written in Java$^\omega$, through an example, with programs written in ordinary Java, and in particular using Java API. Furthermore we extend the language with mc_parameters, and extend the $\mathcal{E}$ transformation generating programs using the callback pattern. Callbacks are widely considered a way to define function pointers in Java, that is a way to define higher order methods. We show the example with mc_parameters, and the transformed program generated according to the callback transformation.

```
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 {...}
  public class Rectangle extends Shape {...} }

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

Fig. 6. `Shape`, `Circle` and `Rectangle` definition for mc_parameter example

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

Fig. 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) ;}}
```

Fig. 8. Transformed program for `FList` definition

```
  public abstract class Shape extend HighComparable {
    public abstract Double area();
    public abstract Double perimeter();
    public abstract String toString();
    public int compareTo(Object s){return compareTo(new area(),s);}
    public class Circle extends Shape {...}
    public class Rectangle extends Shape {...} }

    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);....}
```

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

References

[ALSU07] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[BGGvdA08] G. Bracha, N. Gafter, J. Gosling, and P. von der Ahe. Closures for the java programming language (aka bgga), 2008. www.javac.info.

[BO07] M. Bellia and M.E. Occhiuto. JH-preprocessing, 2007. www.di.unipi.it /∼occhiuto/JH.

[BO08a] M. Bellia and M.E. Occhiuto. *JavaΩ: The Structures and the Implementation of a Preprocessor for Java with m_parameters*. University of Pisa, Dipartimento Informatica, 2008.

[BO08b] M. Bellia and M.E. Occhiuto. Methods as parameters: A preprocessing approach to higher order in java. *Fundamenta Informaticae*, 85(1):35–50, 2008.

[Bri08] B. Bringert. HOJ - higher-order Java, 2008. cs.chalmers.se/bringert/hoj.

[CS06] C.Donnely and R. Stallman. Bison: The yacc-compatible parser generator, 2006. www.gnu.org/software/bison/manual.

[DNR06] R. Dyer, H. Narayanappa, and H. Rajan. Nu: Preserving design modularity in object code. *ACM SIGSOFT Software Engeneering Notes*, 31, 2006.

[GHJV05] E. Gamma, R. Helm, R. Johnson, and J.M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2005.

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

[Hor07] C. Horstmann. *Big Java ,3rd ed.* Wiley Computing, 2007.

[LMB95] J.R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. OŔelly, 1995.

[Mey04] 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.

[ORW98] 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.

[OW97] 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.

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

[SS07] N. Sridranop and R. Stansifer. Higher-order functional programming and wildcards in java. In *ACMSE 2007, ACM*, pages 42–46, 2007.

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*

Fig. 10. Extended syntax [GJSB00]

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

Fig. 11. Transformation $\mathcal{E}$ - part 1

```
static class Ide_{M_0} implements ApplyClass{
     public Object Apply(Object o, Object [] Pars){
           return ((A) o).Ide_{M_0}(Pars[0])}}
           ...
static class Ide_{M_k} implements ApplyClass{
     public Object Apply(Object o, Object [] Pars){
           return ((A) o).Ide_{M_k}(Pars[0])}}
}
```

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 $Block = St;\ StList$

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

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

Implementation using anonymous inner classes. In this case classes Ide_{M_i} are not defined in class A

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

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

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

$$\mathcal{E}[\![Exp]\!]_\rho = \begin{cases} ((Type)(Par.\texttt{Apply(}[\![Exp_1]\!]_\rho, \texttt{new Object[]\{(}FType\texttt{)}\mathcal{E}\ [\![Exp_2]\!]_\rho\texttt{)\}})), & \\ & \text{with } Exp = Exp.Par(Exp) \wedge \\ & \rho(Par) = \texttt{Fun } FType \rightarrow Type \\ \mathcal{E}[\![Exp]\!]_\rho.Ide(\mathcal{E}[\![Exp]\!]_\rho), & \text{with } Exp = Exp.Ide(Exp) \wedge \\ & \rho(Ide) = \perp \\ \mathcal{E}[\![Exp]\!]_\rho\ Op\ \mathcal{E}[\![Exp]\!]_\rho & \text{with } Exp = Exp\ Op\ Exp \\ etc. & \end{cases}$$

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

Fig. 12. Transformation $\mathcal{E}$ - part 2