JavaΩ: Higher Order Programming in Java

Marco Bellia  
*Dipartimento di Informatica*  
*Università di Pisa, Italy*

Maria Eugenia Occhiuto  
*Dipartimento di Informatica*  
*Universita di Pisa, Italy*

1 Introduction

The paper considers the problems and the solutions that are concerned with the extension of Java with Higher Order (HO) mechanisms. These mechanisms are typical of functional languages and include higher order abstraction, that makes a Java method parametric with respect to other methods that can be passed as parameters, and code as first class value, that can be assigned to variables, passed as parameter, returned by method invocations, and executed in different points of a Java program. The main motivations for the presence of HO mechanisms in a programming language are code reusability and expressivity. The code reusability degree of a language (Biddle & Tempero, 1996) is concerned with the structures that the language provides to re-adapt the code of a program to the modifications which are consequent to its use, for instance to eliminate program bugs, or to modify its context or interface of use, or to add new program functionalities. Java provides inheritance with this aim, while functional languages provide HO functions: Combining the two forms highly enhances the degree of code reusability of the resulting language. Code expressivity (Felleisen, 1991) is concerned with the (data and control) structures that the language provides to express the expected behaviours of the programs, for instance, to represent the application data, or to control data visibility and access, or to model code iteration or code induction on different types of data. A good expressivity improves the use of the language and decreases programming errors. Java provides object, classes and inner classes with this aim, while functional languages provide functions as first class value. HO methods and closures (as anonymous functions as first class values are improperly called in Java) aims (Gafter, 2007) are (a) simplifying many kinds of programmings that currently rely on anonymous inner classes; (b) enabling more powerful libraries with methods (*Control APIs*) in which the controlled statements are received as closures or as method parameters (mc.parameters); (c) improving concurrent programming and simplifying applications that rely on existing concurrent APIs; (d) supporting a programming style that enhances the use of aggregate operations, i.e. operations that apply an operation to collection members; (e) enabling future APIs design to replace language design for extending the Java platform. In fact, at the base of all the above points is the verbosity and the awkwardness of anonymous inner classes in expressing functions and more in general, parameterized code. In addition, the Java scope rules limit, in inner classes, the use of non local variables (or class members) to those that
are declared **final** in the method code. This increases the difficulty in expressing parameterized code that share variables with methods that are invoked during the computation and are no more active.

Describing the evolution of the JavaΩ project we discuss the problems that arise from the design of higher order constructs that can be added to Java and whose implementation must preserve, at the same time, the features of re-target-ability and portability of the Java compilers and of the Java Virtual Machine, JVM. We give the formal definition of such constructs, including the extensions to the syntax of the Java Language Specification, JLS, and the formalisation of a translation semantics. This semantics provides a formal way to map programs of the extended language into behaviourally equivalent programs of the ordinary language and provides an implementation technique based on a preprocessing (Bellia & Occhiuto, 2009b; Bellia & Occhiuto, 2009a)

Several languages exist which integrate OO and HO features, they are: C# (Wiltamuth & Hejlsberg, 2003), F# (Syme, 2003), Python (Community, 2010), Ocaml (Cousineau & Mauny, 1998), Scala (Odersky, 2010), Ruby (Flanagan & Matsumoto, 2008). All such languages are strongly typed, and provide closures (lambdas). All such languages except Ruby are statically typed checked and allow to pass methods as parameters, but none of such languages defines a translation semantics for its HO constructs, neither uses preprocessing for implementation. Hence, a comparative study of such languages is beyond the scope of the paper, since it does not help to understand how these two latter points can be achieved in Java.

In Section 2 a structured presentation of JavaΩ introduces to the faced problems, the proposed solutions, and the results that are obtained on extending Java with HO mechanisms. In Section 3 the principles and the formalisation of the translation semantics, used in the subsequent sections, are illustrated. The mechanism of closures and that of HO methods with mc.parameters are designed in such a way that they result one another orthogonal and can be discussed separately. Hence, in Section 4 Java 1.5 is extended to allow closures. The syntax and semantics adopted are essentially those in (Bellia & Occhiuto, 2009a) except for minor changes in the translation semantics due to technical reasons concerned with the used formalism. In Section 5 Java 1.5 is extended to allow HO methods and mc.parameters, as defined in (Bellia & Occhiuto, 2008a; Bellia & Occhiuto, 2009b), but the translation semantics is modified in order to consider Java 1.5 (with generics) instead of Java 1.3. The Section 4.6 contains the most meaningful examples published in the above cited papers revisited because of the the integration of the two mechanisms. The Section 6 integrates the two extensions by putting the two translations together and addresses the implementation. The last section contains some final remarks. Eventually, the proofs of the Lemmas and Theorems enunciated in this paper can be found in a Technical Complements of this paper (Bellia & Occhiuto, 2007a).

## 2 The JavaΩ Project

This project originated in 2004 (Bellia & Occhiuto, 2004) where, through a case study, we compare the expressivity of five different programs, all based on the OO programming paradigm but using also higher order mechanisms (mainly in the form of methods with higher order functionalities), all solving the same problem. Three programs are written in Java, one is written in J++ and the last one is written in Pizza. As far as the the higher order mechanisms are concerned, the first two programs in Java use reflection, the third one uses anonymous classes, the J++ program uses delegates and the Pizza program uses function abstractions (closures). The result of the comparison led to identify three possible mechanisms to add to Java to extend it with higher order: higher order methods, method extraction and closures.

In (Bellia & Occhiuto, 2005) the integration of such mechanisms in an OO paradigm is studied, through the extension of the ζ-calculus (Abadi & Cardelli, 1996). The new ζ-calculus provides, for such
mechanisms, the reduction semantics and the typing rules and prove calculus confluence, subtyping, sub-
typing contravariance, subject reductions hence type soundness. If the problem of the integration of HO
mechanisms in an OO language can be considered positively solved from a theoretical point of view, the
problem of how this can be accomplished to match the specific characteristics of Java, which include for
instance, interfaces (and their fundamental role in API design), methods (with their overloading resolution
at compile time, and their overriding resolution at run time), and its implementation (which is based on a
mixture of compilation and interpretation, JVM) is still open.

In (Bellia & Occhiuto, 2007b) we began to study the design of HO mechanisms especially tailored
for Java and its peculiar implementation, defining the structure of \textit{m\_parameters} as a form of higher order
methods for Java. (Bellia & Occhiuto, 2007b; Bellia & Occhiuto, 2008c) provide the syntactic extensions
to Java grammar to pass methods as parameters to other methods. A formal semantics is also defined and
examples that show the use of the mechanism are shown. The expressivity is discussed by comparing
programs that use such a mechanism with programs that compute in an almost equivalent way but are
written in ordinary Java.

One of the most remarkable aspects of our study is the formal semantics which is defined in the
form of a translation semantics. This kind of semantics maps programs of the extended language into
programs of ordinary Java as discussed in Section 3. This approach in the design of HO extensions for
Java is used in (Bellia & Occhiuto, 2008a) where the translation semantics of Java 1.4, extended with
\textit{m\_parameters}, is defined and used to form the kernel (Bellia & Occhiuto, 2008b) of a pre-processor for
Java compilers (Bellia & Occhiuto, 2007b). The drawbacks of \textit{m\_parameters} are the impossibility to \textit{i})
statically check type correctness and \textit{ii}) pass overloaded methods. Such drawbacks are both overcome in
(Bellia & Occhiuto, 2008a; Bellia & Occhiuto, 2009b) where \textit{mc\_parameters} are defined. Mc\_parameters
have a type which specifies also the class hierarchy in which the passed method must be defined, allowing
a static type checking and overloading static resolution. The translation semantics of \textit{mc\_parameters} is
defined using anonymous inner classes, involved in the Java methodology known as callback (Horstmann,
2007).

In 2006, the Java community started a debate about the opportunity to introduce closures in Java.
The proposals resulting from such a debate are contained in (Bracha et al., 2006; Colebourne & Shulz,
2006; Lee et al., 2006) and more recently (Bracha et al., 2008; Reinhold, 2009). In such proposals Java is
extended with different closure forms and other mechanisms with the aim to allow a better use of closures
and integration in the language (closure conversion, shared variables, loop abstraction). As a matter of
fact, our paper (Bellia & Occhiuto, 2005) about $\zeta_x$-calculus already contained the definition of a form of
closures. In (Bellia & Occhiuto, 2009a) we apply the translation semantics approach to the extension of
Java with a form of closures which adopts many of the features of the above cited papers. As for HO
methods the approach guarantees the goals of portability and re-target-ability but is not well suited to study
different and alternative definitions of closures and related mechanisms. For this reason we resorted to a
minimal core calculus, more meaningful and suited to Java features, to extend it with closures. Such a
calculus (Bellia & Occhiuto, 2010) is obtained extending Featherweight Java (Igarashi et al., 2001).

3 Translation Semantics and Source to Source Translations: Principles

The translation semantics describes the meaning of the (data and control) structures of a language (called,
the source language) in terms of the structures of another language (called the object or target language).
Hence it associates to a program $P_s$ in the source language a program $P_t$ in the target language as its meaning:
We call \( P_t \) the semantics of \( P_s \). In the Java\( \Omega \) project, the source language is a Java extension while the target language is ordinary Java. Hence, It has several benefits including:

- It defines the meanings of the new constructs in terms of the well known constructs of Java, without the introduction of the semantic structures otherwise needed.
- It helps in evaluating the expressivity power of the new constructs allowing an immediate comparison between a program that uses the new constructs and its translation.
- It allows an implementation of the execution support of the extended language completely re-targetable on all the compilers developed for Java.

The translation semantics is basic in the compiler design and implementation, where it is used in combination with other kinds of semantics (in order to guarantee the correctness of the compiler back-end) and is almost all expressed through attribute grammars. Other uses of translation semantics are in compiler bootstrapping, source-to-source translation (Albrecht et al., 1980; Koser et al., 2003), reverse engineering (Ward, 1994).

A translation semantics can be formally defined by means of a rule system containing one or more rules for each of the productions of an unambiguous grammar \( G \) of the source language. Let \( p \) be the production \( C_0 \rightarrow C_1 \ldots C_n \) of the grammar, with, \( C_0,\ldots,C_n, n+1 \) (possibly non different) grammatical categories. A grammatical category can be either a syntactic, e.g. Expression, or a terminal (lexical) category, e.g. InfixOp (cfr. the Java grammar of JLS (Gosling et al., 2005)). Let \( G_S \) and \( G_L \) be the set of all the syntactic and terminal respectively, categories of a grammar \( G \). Moreover, a grammatical category can define either an infinite set of syntactic structures, e.g. all the expressions that can be written in the language, or a finite set, e.g. all the infix operators that can occur in the expressions of the language. Terminal categories never occur in the left part of a production and may consist of singletons called tokens. Let \( \mathcal{E} \) be a source to source language translation and \( \Sigma^G_\mathcal{E} \) be the rule system that defines \( \mathcal{E} \) using grammar \( G \). Then, in correspondence to production \( p, \Sigma^G_\mathcal{E} \) contains \( k \geq 1 \) rules of the form:

\[
\mathcal{E}(A_0) = r_i(A_1, \mathcal{E}(A_1), \ldots, A_n, \mathcal{E}(A_n)) \quad \text{with} \quad b_i(A_0, A_1, \ldots, A_n),
\]

Symbols \( A_0,\ldots,A_n \) are variables and are ranging on the structures of the set defined by the corresponding categories \( C_0,\ldots,C_n \). \( b_i \) is a predication and expresses a condition that constrains the structures that can be bound to each \( A_j \) in order to apply the \( i^{th} \) rule of production \( p \). Eventually, \( r_i \) is a syntax constructor and expresses the translation produced by the rule on the structure bound to \( A_0 \). Hence, the rule can be read: if the structure that we are translating is a structure of category \( C_0 \) and is exactly composed of a structure \( A_1 \) of category \( C_1 \) followed by...followed by a structure \( A_n \) of category \( C_n \) and moreover, on the structures \( A_0, A_1, \ldots, A_n \) condition \( b_i \) holds, then the structure \( A_0 \) is translated in the structure obtained by applying \( r_i \) to the structure \( A_1 \), and possibly, to the translated structure of \( A_1, \ldots \) to the structure \( A_n \), and possibly, to the translated structure of \( A_n \). The one described above, is the general structure of a rule system for source to source translation. In fact, 1) predication \( b_i \) rarely requires more that one or two of the \( n+1 \) arguments indicated, as well as the constructor \( r_i \) rarely requires the translation of all the component structures \( \mathcal{E}(A_1),\ldots,\mathcal{E}(A_n) \) and the use of both \( A_i \) and of translation \( \mathcal{E}(A_i) \); 2) More often, the rule system contains only one rule for production. In this case condition \( b_i \) is obviously absent. 3) Eventually, only a small number of rules have a different \( r_i \). These rules are the one really significant for characterising the translation defined by rules system. The most part of rules have the following form:

\[
\mathcal{E}(X_0) = f(X_1, \mathcal{E}(X_1)),\ldots,f(X_n, \mathcal{E}(X_n)) \quad \text{with} \quad b_i(X_0, X_1, \ldots, X_n),
\]
In the rule, the constructor \( r_i \) is the identity constructor that leaves unchanged the structure \( X_0 \) but replaces each substructure \( X_j \) (\( j > 0 \)) with the corresponding translation if any: Hence \( f(X_j, \mathcal{E}(X_j)) \) stands for \( X_j \) if \( X_j \in G_L \), for \( \mathcal{E}(X_j) \) if \( X_j \in G_S \). A Rule as the above one is really, a metarule when each symbol \( X_j \) is a metavariable ranging on variables of many different categories. A metarule stands for all the rules that can be obtained, starting from the productions of the grammar, by instantiating symbols \( X_j \) with variables of the right categories. Looking at the two rule systems given in next sections, we note that they have 12 and 8 respectively, significant rules and only one metarule for a Java grammar that has many dozen of productions. The metarule is for both the systems,

\[
\mathcal{E}(X_0) = f(X_1, \mathcal{E}(X_1)), \ldots, f(X_n, \mathcal{E}(X_n)) \text{ otherwise },
\]

It is called otherwise metarule, and applies to the each production of the grammar for which none of the other rules of the system apply.

Eventually we note that, the use of language grammar guarantees completeness and termination of source to source translation, namely each program has its translation and it is computed in as many steps as the productions used to parse the program. Moreover, the non ambiguity of grammar guarantees soundness, namely the program translation is unique.

### 4 Closures: Lambda Expressions (Anonymous Functions) in Java

Closures enclose a piece of code, namely a statement list and/or an expression, possibly parameterized, and make such code available for possibly, many and different invocations in the program. More important, closures are values (function objects (Gafter, 2008)) that can be computed by and passed to methods: They allow the definition of methods whose code, hence whose behaviour, is parametric with respect to the closures with which are invoked. A closure may contain free variables that are bound in the lexical scope of the blocks enclosing the closure definition.

#### 4.1 Closure Basic Structure

A basic common structure for Java closures emerges from (Bracha et al., 2008; Colebourne et al., 2008; Reinhold, 2009). This structure can be described by the following characteristics: Closures

1. encapsulate an arbitrary piece of Java code and generalise it through parameters (as anonymous functions (Landin, 1966), written in Java)

2. have a type which depends on the argument types, the type of the possibly computed value, and the exception types. In Java closure types are generic.

3. are values hence can be assigned, passed as parameters, returned as value result (of the correct type)

4. can be invoked in a uniform way. In particular if a closure is bound to a formal parameter \( x \) of a method, or another closure, the invocation of \( x \) does not depend on (selectors defined in the) the closure type.

5. can contain non-local variables, in this case the non local variable:

   - is the one bound in the lexical scope of the nearest block enclosing the closure, according to Java static scope rule.
• such variable remains accessible even though the frame related to the execution of the block has been deallocated.
• it can be accessed and modified as local variables.

Syntax apart, such a structure leads to express the same behaviours (i.e. intended semantics) and programming uses (i.e. methodologies) for closures (Gafter, 2008; Colebourne & Shulz, 2006), and includes those in (Lee et al., 2006). As a matter of fact, such closures basic structure is adopted in (Bellia & Occhiuto, 2010) and also here.

4.2 Syntax of Closures.

A closure is syntactically an expression. The following example shows a closure definition: it has a formal parameter of name y and type int and returns a value of type int, the body contains a non local variable x.

```
{(int y) :int => x += y; return ++x;}
```

The declaration of the return type follows the syntactic structure of method definition in Java and does not force the implementation to infer the return type as in (Bracha et al., 2008). However, we defer to type checking, of ordinary Java compilers, the burden of checking the correctness of the types declared in the translated programs. As a matter of fact, our translation maps closures with incorrect types into programs with incorrect types too. A closure has a type and it may be assigned to a variable, passed as a parameter of the right type, returned as a value, invoked as a method using the reserved selector invoke.

All the extensions needed to Java grammar are reported in the Appendix. The main extensions for closures are:

```
Primary::=...| Closure ...
Closure ::= { FParameters : (Type | void) ThrowsOpt => Block }
Type ::= ParameterizedType | BasicType | ClosType
ClosType ::= { ( ExtendedTypeList ) : (Type | void) ThrowOpt }
ExtendedTypeList ::= ExtendedType[(,ExtendedType)∗]
NonInvocationSelector ::= ... | .invoke [Arguments] ...
```

Closures are anonymous functions but unlike functional abstractions can (access and) modify (free, i.e. non local) variables bound in the lexical scope of the block in which the closure is defined. In particular a closure can use variables that are bound in a scope that is no more active (Lindholm & Yellin, 1996) at the time the closure is invoked, see (Tronicek, 2007) and the use of the hashtable in the example in Figure 2. Since local variables are allocated in a Java frame (Lindholm & Yellin, 1996) when the scope is active and deallocated when it is not active, then the handling of non local variables in closures needs a specific treatment. A solution of this problem leads to a new, additional, notion for those local variables that can be accessed and modified as non local in a closure. Such variables must be allocated in the heap. In (Gafter, 2008) this kind of variables are annotated @shared to mark the difference with ordinary local variables. In the structure adopted in this paper, shared is an additional modifier for local variables to mark that the difference between this new kind of variables and the variables of ordinary Java is mainly semantics and it involves memory allocation (heap vs. frame (Lindholm & Yellin, 1996)), access and modification of the variable. Hence the syntax of both the local variable declarations and the formal parameters (Gosling et al., 2000) is extended adding:

```
LocalVariableDeclarationStatement ::= [ final ] shared ] Type VariableDeclarators
FormalParameter ::= [ final | shared ] [Annotations] ExtendedType VariableDeclaratorId
```
4.3 The Intended Meaning of Closures

The intended meaning introduces the (abstract) domains on which the meaning of the various constructs is given by the semantics. The intended meaning of closures associates to a:

(i) closure type \{(ET_1,...,ET_n) : ET \ [throws QI_1,...,QI_k]\}, through a one-to-one correspondence \(\mathcal{J}\), a reference type \(\mathcal{J}(ES_1,...,ES_n,\{QI_1,...,QI_k\},\mathcal{I}\ES)\). Where, for each \(i, ES_i\) (resp. \(ES\)) is the type \(ET_i\) (resp. \(ET\)), if \(ET_i\) (resp. \(ET\)) is not a closure type, otherwise it is the reference type associated to \(ET_i\) (resp. \(ET\)).

(ii) closure literal \{(FS_1 At_1 ET_1 V_1,...,FS_n At_n ET_n V_n) : ET \ [throws QI_1,...,QI_k] \Rightarrow BB\}, a (function) object, of type \(\mathcal{J}(ES_1,...,ES_n,\{QI_1,...,QI_k\},\mathcal{I}\ES)\), that wraps a function \(\mathcal{H}_{BB}\). Type \(\mathcal{J}(ES_1,...,ES_n,\{QI_1,...,QI_k\},\mathcal{I}\ES)\) is obtained according to (i) above from type \{(ET_1,...,ET_n) : ET \ [throws QI_1,...,QI_k]\} that is extracted from the closure literal in the obvious way and is the type of the closure literal. Function \(\mathcal{H}_{BB}\), applied to a \(n\)-tuple of values \(V_1,...,V_n\) of type \(ES_1,...,ES_n\) respectively, either computes a value \(V\) of type \(ES\) or fails depending on the execution of \(BB_r\) in a frame where \(V_1\) is bound to value \(V_1\) and..., and \(V_n\) is bound to value \(V_n\). Where \(BB_r\) is the ordinary Java code resulting from the sequence of statements (and/or declarations) of closure body \(BB\) in which each free (i.e. non-local) variable, if any, is bound to the corresponding shared (or final) variable of the lexical scope in which the closure occurs. If the computation fails then an exception (of a type possibly in \(QI_1,...,QI_k\)) is thrown.

(iii) closure invocation \(E._\text{invoke}(E_1,...,E_n)\), where \(E\) is an expression computing a (function) object \(c\) of type \(\mathcal{J}(ES_1,...,ES_n,\{QI_1,...,QI_k\},\mathcal{I}\ES)\), the application of the function, that \(c\) is wrapping, to the \(n\)-tuple of values \(V_1,...,V_n\) that results from the evaluation of the argument list \(E_1,...,E_n\). Arguments \(E_1,...,E_n\) must be of the right types \(ES_1,...,ES_n\).

(iv) shared variable (resp. parameter) \(V\) of type \(ET\), possibly initialised with expression \(E\), a variable of a class of objects wrapping variables of type \(ET\) that are allocated in the heap. The wrapped variable is possibly initialised to the value of \(E\). The shared variable \(V\) is then, accessed and modified through such a reference.

4.4 The Translation Semantics \(\mathcal{F}[]\)\(_\tau\).

The translation semantics is defined by translation \(\mathcal{F}[]\)\(_\tau\) in Figure 1, it is based on the structures of interfaces, anonymous classes, and classes of variables, in the sense that \(\mathcal{F}[]\)\(_\tau\) translates closures into a composition of such structures. The one-to-one correspondence \(\mathcal{J}\) of the closure intended meaning is implemented as a bijective function \(\text{ClosType} \rightarrow \text{Interface}\), between the closure type and the interface for objects wrapping functions of that type. The implementation of \(\mathcal{J}\) is based on a family of interfaces, indexed by a couple of naturals \(n,k\), defined as follows:

```
interface ISnk<ET_1,...,ET_n,ET, QI_1 extends Throwable,...QI_k extends Throwable>{
  public ET invoke(FS_1 At_1 ET_1 V_1,...,FS_n At_n ET_n V_n) throws QI_1,...,QI_k;
}

interface ISnkvoid<ET_1,...,ET_n, QI_1 extends Throwable,...QI_k extends Throwable>{
  public void invoke(FS_1 At_1 ET_1 V_1,...,FS_n At_n ET_n V_n) throws QI_1,...,QI_k;
}
```
\[ \mathcal{F}[[CB]]_\tau = \{ \mathcal{F}[[MB_1]]_\tau \ldots \mathcal{F}[[MB_n]]_\tau \} \text{ with } CB = \{ MB_1 \ldots MB_n \} \]

\[ \mathcal{F}[[P]]_\tau = \left\{ \begin{array}{l}
\text{where: } p \in \text{Identifier} | \text{this} \\
\text{new } $I$n $<$\mathcal{F}[[ET_1]]_\tau, \ldots, \mathcal{F}[[ET_n]]_\tau, \mathcal{F}[[ET]]_\tau, QI_1, \ldots, QI_k > () \{ \mathcal{F}[[public ET]]_\tau, \ldots, QI_k \}_{\mathbb{BB}} \}_{\tau/(\text{this}(/sSelf.on))} \\
\quad \text{with } P_I \equiv FS_I \wedge ET_I \wedge V_I \\
\quad \mathcal{F}[[ET]]_\tau = \left\{ \begin{array}{l}
\text{new } $I$void $<$\mathcal{F}[[ET_1]]_\tau, \ldots, \mathcal{F}[[ET_n]]_\tau, QI_1, \ldots, QI_k > () \{ \mathcal{F}[[public void ET]]_\tau, \ldots, QI_k \}_{\mathbb{BB}} \}_{\tau/(\text{this}(/sSelf.on))} \\
\quad \text{with } P_I \equiv FS_I \wedge ET_I \wedge V_I \\
\quad \mathcal{F}[[L]]_\tau = \left\{ \begin{array}{l}
\text{with } TR \equiv \text{throws } QI_1, \ldots, QI_k \\
\quad \text{with } TR \equiv \text{throws } QI_1, \ldots, QI_k \\
\text{final } TI = \text{new } T(\mathcal{F}[[ET]]_\tau) \\
\quad \text{with } L \equiv \text{final } ET I \equiv E \wedge E \in \text{Closure} \\
\text{final } TI = \text{new } T(\mathcal{F}[[ET]]_\tau) \\
\quad \text{with } L \equiv \text{final } ET I \equiv E \wedge E \in \text{Closure} \\
\quad \mathcal{F}[[MB]]_\tau = \left\{ \begin{array}{l}
\quad R_I \equiv FR_I \wedge ER_I \wedge V_I \wedge ER_I = \mathcal{F}[[ET_n]]_\tau \\
\text{and } L_o \equiv \text{final } object sSelf = \text{this}; \\
\text{and } \lambda_i = \left\{ \begin{array}{l}
\text{final } T \mathcal{G}(I_i) O_i = \text{new } T(V_i); \\
\text{if } FS_i \equiv \text{shared} \\
\lambda_i = \left\{ \begin{array}{l}
\text{final } I_i \equiv FS_i \\
\text{if } FS_i \equiv \text{final} \\
\mathcal{G}(I_i).value \equiv \left\{ \begin{array}{l}
\text{if } FS_i \equiv \text{shared} \\
\tau \equiv \left( \begin{array}{l}
\tau(\text{this}) = 1 \\
\tau \equiv (I_i/E_1, \ldots, E_n) \\
\tau(\text{this}) = (sSelf, on) \\
\text{if } \tau(\text{this}) = (sSelf, on) \\
\end{array} \right) \\
\end{array} \right) \\
\end{array} \right) \\
\end{array} \right) \\
\text{with } BB \equiv S:BB_r \wedge L \equiv \text{shared } ET I \{ I \equiv E \} \\
\mathcal{F}[[ET]]_\tau = \mathcal{F}[[L]]_\tau: \mathcal{F}[[BB_i]]_\tau \}_{\tau/(l.value)} \\
\mathcal{F}[[L]]_\tau = \mathcal{F}[[L]]_\tau: \mathcal{F}[[BB_i]]_\tau \}_{\tau/(l.value)} \\
\text{with } E \equiv I = E_1 \wedge E_1 \in \text{Closure} \\
\text{where: } \\
\tau(\text{this}) = (I_i/E_1, \ldots, E_n) \equiv \left\{ \begin{array}{l}
E_i \equiv x \equiv I_i \wedge E_i \notin \{ (sSelf, on), (sSelf, off) \} \\
\text{this} = I_i \wedge E_i = (sSelf, on) \\
(\text{this} = I_i \wedge E_i = (sSelf, off) \\
\tau(x) \equiv x \notin \{ I_1, \ldots, I_n \} \\
\end{array} \right) \\
\text{Legend: } \text{ For arbitrary index } p, \text{ At,At}_p \in \text{Annotations, BB,BB}_p \in \text{BlockStatements, CB} \in \text{ClassBody, D, D}_p \in \text{ModifierOpt, E, E}_p \in \text{Expression, ET, ET}_p \in \text{ExtendedType, FS,FS}_p,FR_p \in \{ \text{final | shared} \}, I, I_p \in \text{Identifier, L \in LocalVariableDeclarationStatement, MB,MB}_p \in \text{MemberDecl, O,O}_p \in \{ \}, P, P_p, R_p \in \text{FormalParameter, Py} \in \text{Primary, Q, Q}_p \in \text{TPs, Ql} \in \text{QualifiedIdentifier, S, S}_p \in \text{Statement,T,T}_p \in \text{Type, TR} \in \text{ThrowOpt, TV,T_V} \in \{ \text{Type | void} \}, V,V_p \in \text{VariableDeclaratorId} \]

\[ \text{Figure 1: Closures: } \mathcal{F}[[ET]]_\tau: \text{Translation Semantics.} \]
Each interface represents the functions with \( n \) arguments raising exceptions of \( k \) different types. In the first case the represented functions return a value of type \( ET \). In the second case they do not compute any value (i.e. compute the nullary, unit value). According to the semantics defined for closures, \( \mathcal{F}[\square]_{\tau} \) translates:

(i) the closure type \( \{ (ET_1, \ldots, ET_n) : ET \ [\text{throws } QI_1, \ldots, QI_k] \} \) into an instantiation of the interface \( I\$nk \) in which the type variables \( ET_i \) are instantiated to the corresponding type \( ES_i \) obtained translating \( ET_i \).

(ii) a closure literal \( \{(FS_1At_1 ET_1 V_1, \ldots, FS_nAt_n ET_n V_n) : ET \ [\text{throws } QI_1, \ldots, QI_k] \Rightarrow BB \} \) of type \( \{(ET_1, \ldots, ET_n) : ET \ [\text{throws } QI_1, \ldots, QI_k] \} \) into an instance of the class implementing the corresponding interface, namely:

\[
\begin{align*}
\text{new } I\$nk &<ES_1, \ldots, ES_n, ES, QI_1, \ldots, QI_k> ()\{ \\
\text{public } ES &\text{ invoke}(FS_1At_1 ES_1 V_1, \ldots, FS_nAt_n ES_n V_n) \\
&\text{throws } QI_1, \ldots, QI_k \{BB\}\}
\end{align*}
\]

where \( BB \), results from the translation of \( BB \) and \( ES_i \) results from the translation of \( ET_i \) according to \( \mathcal{F}[\square]_{\tau} \).

(iii) closure invocation \( E . \text{invoke}(E_1, \ldots, E_n) \) into \( \mathcal{F}[\square]_{\tau}. \text{invoke}(\mathcal{F}[\square_{E_1}], \ldots, \mathcal{F}[\square_{E_n}]) \). As a matter of fact, closure invocation is not considered in the rules defined in Figure 1 since it is trivially dealt with by the otherwise metarule.

(iv) shared variables of type \( ET \) into final variables of the class:

\[
\text{class } C\$Shared <ET> \{ ET \text{ value; } C\$Shared (ET n) \{ value=n; \}\}
\]

Each access to a variable \( V \), declared shared, is replaced by an access to \( V.\text{value} \). Class \( C\$Shared <ET> \) is also used to translate shared parameters. If a shared formal parameter of type \( ET \) is declared in a method with body \( \{BB\} \), it is translated into a parameter without modifier final. Moreover a final variable with a new name, is declared in the translation of \( BB \) being of type \( C\$Shared <ES> \), where \( ES \) is the type resulting by the translation of \( ET \), and is initialised to contain the value of the parameter. All occurrences of the parameter in the translation of \( BB \) are translated into accesses to the field \( \text{value} \) of the new variable. The function \( \mathcal{G} : \text{Identifier} \rightarrow \text{Identifier} \) is used to generate the new name that does not clash with the other identifiers in \( BB \). \( \mathcal{G} \) is an injective function that applied to a name of a parameter generates a new name. In this way the translation \( \mathcal{F}[\square]_{\tau} \) preserves the method headers.

### 4.5 The Formal Definition of \( \mathcal{F}[\square]_{\tau} \)

The translation semantics is defined using the syntax directed rule system described in Section 3. Hence, it has at least one rule for each production of the unambiguous grammar of Appendix, and the translation computes applying to each program construct the rule that is associated to the production with which the construct is parsed. As a matter of fact, the rule system \( \mathcal{F}[\square]_{\tau} \) contains only 12 significant rules and one otherwise metarule: The metarule applies when the production with which the construct is parsed has no significant rules associated to it, and the translation leaves unchanged the construct structure but replaces each component with the result of its translation. In the sequel, we illustrate the translation discussing rule by rule (only for the significant ones) how they work. Firstly we introduce the environment \( \tau \) used as a parameter in the translation \( \mathcal{F}[\square]_{\tau} \), whose definition is contained at the end of Figure 1. The environment
τ associates an identifier $I \in \text{Identifier}$ to an expression $E \in \text{Expression}$. The empty environment is denoted by $\emptyset$ and does not contain bindings. Any other environment is obtained by $\tau \uparrow (I_1/E_1, \ldots, I_k/E_k)$ that adds $k$ new bindings to a given environment $\tau$ (possibly, the empty environment). Each binding associates to the name $I_i$ an expression $E_i$, and we have that $\tau(I_i) = E_i$. Eventually, $(\forall I) \varnothing(I) = \perp$ where $\perp$ is the undefined term. The environment $\tau$ contains a binding for:

1. all the shared local variables or shared formal parameters that have the construct (to which $\mathcal{F}[[[]]]$ applies) in their (lexical) scope. In this case the value of the binding is $I_i.value$ as explained in the previous subsection 4.4.

2. a local variable whose value is a closure. When a closure is assigned to a local variable identifier, all occurrences of such identifier, in the closure body are recursive invocations of the closure. Hence they must be replaced by the self reference $\texttt{this}$ to the function object that the translation produces.

3. $\texttt{this}$: Since the translation maps closures into function objects and occurrences of $\texttt{this}$ are recursive invocations, we need a different identifier to refer to the object on which the method that defines the closure is invoked. Hence we define a reserved identifier $\texttt{s$elf}$ and add a declaration $\texttt{final Object s$elf = this}$ at the beginning of each method body $BB$ see the fifth rule. The environment $\tau$ is consequently extended to support an on-off mechanism which says when $\mathcal{F}[[[]]]$ has to replace the self reference $\texttt{this}$ with the reference to the object bound to variable $\texttt{s$elf}$. In this way, in the translated program, $\texttt{this}$ in the method Apply, refers to the function object implementing the closure and allows recursive invocations, while the object on which the method that defines the closure is invoked is reached through the reserved identifier $\texttt{s$elf}$.

**Theorem 1 (this Transparency).** Occurrences of $\texttt{this}$ in a closure are references to the object on which the method that defines the closure is invoked.

The rules are placed in Figure 1 in an order which is quite near to the order of their use in translating a program containing closures (if we ignore the otherwise metarule that is omitted in the figure and should be placed at the top since it will be the most frequently used).

**ClassBody:** $CB$. Rule 1 applies to constructs of the category $\text{ClassBody}$ and resets to empty, $\emptyset$, the environment of the translation of each $\text{MemberDecl}$ construct which occurs in the $\text{ClassBody}$, since no closure can have such constructs in its scope.

**Primary:** $Py$. The rule 2 replaces the identifier with its value in $\tau$, while rules 3 and 4 replace a closure with the function object obtained instantiating the interface $I$ onk on the types obtained translating each formal parameter type hence defining and translating the method $\texttt{invoke}$ in the environment $\tau$ extended with the binding $(\texttt{this/(s$elf, on))}$, needed to guarantee the $\texttt{this}$ transparency property.

**ExtendedType:** $ET$. Rules 5 and 6 replace the closure type with the interface type $I$ onk instantiated on types obtained applying $\mathcal{F}[[[]]]$ to the formal parameter types. Two different cases are considered, depending whether a return type or $\texttt{void}$ is defined.

**LocalVariableDeclarationStatement:** $L$. Rule 7 considers the case in which a local variable is declared and initialised to a closure. In this case the closure is to be translated using a $\tau$ in which $\texttt{this}$ is bound to the declared identifier to allow recursive invocations of the closure, case 2 above for $\tau$. Rule 8 considers the declaration of a shared variable. Such declaration is replaced with the declaration of
an object of type \texttt{C\$shared} whose variable type is instantiated on the translated type of the declared variable. Rule 9 combine first and second case defining a \texttt{shared} closure.

\textbf{BlockStatements:BB}. The extension of the environment \(\tau\) to replace the name \(I\) of the \texttt{shared} variable with \(I\.value\) is dealt with in rule 11.

\textbf{MemberDecl:MB}. Rule 10 considers the case of a constructor or method declaration. In this case the translation must add a declaration for the local variable \texttt{s$elf} which must contain the reference to \texttt{this}, deal with \texttt{shared} formal parameters which are treated analogously to \texttt{shared} local variables adding for each \texttt{shared} parameter a local variable declaration whose type is \texttt{C\$share} instantiated to the translated type parameter. The name of the new variable is obtained translating the parameter name through function \(G\). The rule also updates the environment \(\tau\) in which the method body is translated, adding the binding for each formal parameter \(I_i/G(I_i).value\) for the \texttt{shared} parameters and the identity for the non \texttt{shared} ones. Eventually the binding \((\texttt{this}/\texttt{s$elf}, \texttt{off})\) is added if \(\tau(\texttt{this})=1\), that is \texttt{this} is not bound in \(\tau\). In fact, if \(\tau\) contains a binding for \texttt{this} such binding can only be \((\texttt{s$elf}, \texttt{on})\) which means that we are translating an \texttt{invoke} method of a function object representing a closure hence the binding for \texttt{this} is not to be modified.

\textbf{Expression:E}. Rule 12 deals with an assignment expression in case the expression on the righthand side is a closure. In such a case the identifier \(I\) on the lefthand side is replaced by its binding in \(\tau\) and the closure expression, analogously to local variable declaration, is translated using a \(\tau\) in which \(I\) is replaced by \texttt{this}.

As anticipated in Section 2, in (Bellia & Occhiuto, 2010) we define a minimal core calculus extending Featherweight Java (Igarashi et al., 2001) with a form of closures which differ from the one adopted in Java\(\Omega\) only syntactically. In that paper we define a reduction semantics for the calculus and prove type safety. Other interesting properties, like the abstraction property will be investigated in future works with the final goal to prove that the two semantics (reduction and translation semantics) commute.

\subsection{Example}

An example, Figure 2, taken from (Goetz, 2007), defines a method \texttt{memoized} that maps closures of type \{\(T_1\) : \(T\}\} into \texttt{memoized} closures of the same type and same \textit{external} behaviour but of different performance since a memoized closure avoids repeating computation for previously processed inputs. To obtain \texttt{memoized} has a closure parameter \(f\), declared \texttt{final}, of type \{\(T_1\) : \(T\}\}, bound to the closure to be memoized, and a \texttt{shared} variable \texttt{table} initialised to an empty hash table. \texttt{memoized} returns a closure that behaves as follows: It has a parameter \(x\) of type \(T_1\) and asks \texttt{table} for a key equal to the value of \(x\). If the key is found then it returns the value bound in \texttt{table} to \(x\), otherwise it behaves like \(f\), invoking \(f.invoke(x)\), \texttt{updates} \texttt{table}, adding the key/value pair \(x/f.invoke(x)\), \texttt{returns} the value \(f.invoke(x)\). Note that, in this way, each hash table \texttt{table} is a private resource of the memoized closure since each time \texttt{memoized} returns the closure and ends, thus making \texttt{table} no more accessible outside the returned closure. The translated program is shown in Figure 2. Eventually, note that such a solution works properly for closures without the recursion operator. For recursive closures the reader is invited to consider the code for the Fibonacci numbers in Figure 2, whose translation and execution are left to the reader:
public class Memoize<T1,T>{//maps closures into memoized closures
class Memoize<T1,T>
{
    //maps closures into memoized closures
    public {{T1}:T} memoized{final {{T1}:T} f}{
        shared Hashtable<T1,T> table = new Hashtable<T1,T>();
        {{T1}:T} memo_f = {{T1} x}: T ⇒ T res = table.get(x);
        if (res == null){res = f.invoke(x); table.put(x,res);} return res;}
        return memo_f;}}

Figure 2: Class Memoize: A Method that Memoizes Closures

class Memoize<T1,T>{//maps closures into memoized closures
    public I$1<T1,T> memoized{final I$1<T1,T> f}{
        final Shared<Hashtable<T1,T>> table =
        new Shared<Hashtable<T1,T>>(new Hashtable<T1,T>());
        I$1<T1,T> memo_f = new I$1<T1,T>(){
            public T invoke(T1 x){T res = table.get(x);
            if (res == null){res = f.invoke(x); table.put(x,res);}
            return res;}
        return memo_f;}}

Figure 3: Class Memoize: The Translated Program – $\mathcal{F}[\text{Memoize}]_\tau$

5 Higher Order Methods: MC_parameters.

The mechanism here defined is a revised version of the one described in (Bellia & Occhiuto, 2008a; Bellia & Occhiuto, 2009b). It introduces constructs to define HO methods for both class (static) and object methods. HO methods have at least one mc parameter that is a parameter that is bound, during the invocation of the HO method, to a method of an arbitrary class or object of the program. Moreover, the parameter can be used, inside the body of the HO method, as if it were any method: It is invoked on a class or an object and applies to arguments of the right types for the bound method.

5.1 Syntax and Intended Meaning for MC_parameters

The main extensions are concerned with formal and actual parameters to include methods as parameters in the definition and invocation of HO methods: Hence, new type expressions for formal mc.parameters and new (value) expressions for the actual ones. A complete definition of the extended Java grammar is in Appendix, while the syntactic categories directly affected are listed below.

ExtendedType ::= Type | FType
FType ::= Fun RootClass : ( [ ExtendedTypeList ] ) ->(void|Type ) ThrowOpt
RootClass ::= Identifier | ParameterizedType
ParameterizedType ::= Identifier ParsOpt (. Identifier ParsOpt )* [ ]
* ParsOpt ::= [ <TypeArguments*> ]
ThrowOpt ::= [ throws QualifiedIdentifier* ]

FType is the new category that we add to Java grammar to extend the type system with a type for mc.parameters. The new type begins with the keyword Fun and specifies a class type and a signature for the methods that can be bound to the mc.parameter. The class type can be either an Identifier, namely
a bound type variable, or a ParameterizedType. It specifies the root class of the hierarchy to which the classes, containing the methods that can be passed as parameter, belong. Then, the signature specifies number and types of the arguments of such methods and the type of the computed value, and possibly, of the throwable exceptions. Note that FType is not a sub-category of Type, which contains the types of ordinary Java, but FType and Type are both sub-categories of ExtendedType. This new category allows to constraint, already at syntactic level, the use of methods as values. In fact, methods can only be used as values of mc_parameters, hence passed as parameters to HO methods and to closures (see the definition of FormalParameter in Appendix, where ExtendedType is used instead of Type). However, methods cannot be assigned to variables (or fields), nor can be returned as values computed by other methods (or closures) (see for instance, definition of MethodOrConstructorDecl where category Type is still used for the return type).

To deal with actual mc_parameters we extend Java expressions adding the syntactic category AExp below:

\[ AExp ::= \text{Abs MethodSpecifier from Type} \]
\[ \text{MethodSpecifier ::= Identifier ([ExtendedTypeList])} \to \text{(void|Type) ThrowOpt} \]

Hence, an actual mc_parameter is an expression that begins with the keyword Abs and computes a possibly overloaded method having name, signature and declaration class as specified in the expression. Given a program, we denote by \(\mathcal{M}(I, EL, T, TL, T_c)\) the method, if any, that has name \(I\), is either defined (possibly overridden) or inherited in class \(T_c\), is applicable (by subtyping, conversion and variable arity, see Section 15.12 (Gosling et al., 2005)) to invocation with argument type list \(EL\), has return type \(T\) (possibly void), and exception type list \(TL\). If method \(\mathcal{M}(I, EL, T, TL, T_c)\) exists then such a method is unique (see Section 8.2 (Gosling et al., 2005)). Otherwise \(\mathcal{M}(I, EL, T, TL, T_c)\) is the undefined \(\bot\). The intended meaning of mc_parameters associates to:\(^1\)

(i) Expression Abs \(I\) (\(EL\)) \to \(T\) [throws \(TL\)] from \(T_a\), the partial function below:

\[ \lambda f : C \to \lambda c : C \to \mathcal{M}(I, EL_{T_a}, T, TL_{T_c}, T_c) \text{ if } c \leq f \leq T_a \land \mathcal{M}(I, EL_{T_a}, T, TL_{T_c}, T_a) \neq \bot, \]

that, given \(f\) and \(c\), selects the method \(\mathcal{M}(I, EL_{T_a}, T, TL_{T_c}, T_c)\), i.e. the method named \(I\) in class \(T_c\), that can be invoked with argument type list \(EL_{T_a}\), return type \(T\) and has exceptions \(TL_{T_c}\), where \(TL_{T_c}\) is included (see Section 8.4.6 in (Gosling et al., 2005)) in \(TL_{T_c}\). If \(I\) is not overloaded in \(T_a\) then \(EL_{T_a} \equiv EL\). Otherwise, \(EL_{T_a}\) is the list of types computed in class \(T_a\) by Java overloading resolution (see Section 15.2 in (Gosling et al., 2005)). If \(\mathcal{M}(I, EL_{T_a}, T, TL_{T_c}, T_c) \neq \mathcal{M}(I, EL_{T_a}, T, TL_{T_c}, T_a)\) then

\(^1\)\(\leq\) (and \(\geq\)) is the Java subtyping relation (see Section 4.10 (Gosling et al., 2005)) possibly extended on lists of types in the obvious way, i.e. \(t_1 \ldots t_n \leq s_1 \ldots s_n\) means \(t_1 \leq s_1 \land \cdots \land t_n \leq s_n\). Eventually, \(t_1 \ldots t_n \in s_1 \ldots s_k\) means \((\exists t_1 \leq u_1, \ldots, t_n \leq u_n)\{u_1, \ldots, u_n\} \subseteq \{s_1, \ldots, s_k\}\), where \(\subseteq\) is set inclusion.
\( M(I,EL_{T_a}, T, TL_T, T_c) \) is an overriding of \( M(I,EL_{T_a}, T, TL_T, T_a) \). If \( M(I,EL_{T_a}, T, TL_T, T_a) \) does not exist then the program is not legal.

(ii) Parameter \( \text{Fun} \; T_f: (EL_T) \rightarrow T_f \; [\text{throws} \; TL_T] \) \( p \), together with the argument \( \text{Abs} \; I \; (EL) \rightarrow T \; [\text{throws} \; TL] \) from \( T_a \) supplied for it, the partial function below:

\[
\lambda c : C \rightarrow M(I,EL_{T_a}, T, TL_T, T_c) \text{ if } c \leq T_f \leq T_a \land M(I,EL_{T_a}, T, TL_T, T_a) \\

\]

that, given a class \( c \), subclass of \( T_f \), selects method \( M(I,EL_{T_a}, T, TL_T, T_c) \), provided that both following conditions hold: (a) \( T_f \leq T_a \); (b) \( M(I,EL_{T_a}, T, TL_T, T_a) \) is a contracovariant (Liskov & Wing, 1993; Abadi & Cardelli, 1996; Bellia & Occhiuto, 2005) of the expected method, i.e. \( T \leq T_f \land TL_T \subseteq TL_T_f \land EL_T_a \geq EL_T_f \).

(iii) Occurrence \( E.p(E_1,...,E_n) \) inside the body of a HO method invoked with argument \( \text{Abs} \; I \; (EL) \rightarrow T \; [\text{throws} \; TL] \) from \( T_a \) supplied for parameter \( \text{Fun} \; T_f: (EL_T) \rightarrow T_f \; [\text{throws} \; TL_T] \) \( p \), the invocation of method \( M(I,EL_{T_a}, T, TL_T, T_c) \) where \( T_c \) is the effective (i.e. run-time class) class of the object computed by \( E \) and \( EL_{T_a} \) are correct types for the argument list \( E_1,...,E_n \).

The intended meaning anticipates the following relevant semantics properties:

- The method bound to a mc\_parameter has: a) the effective signature that results resolving overloading in the root class \( T_a \), and b) the effective definition that is found in the class \( T_c \) of the object on which the mc\_parameter is invoked (see Java run-time method dispatch in Section 15.12.2.5 in (Gosling et al., 2005));

- the class \( T_c \) is a subclass of \( T_f \) which must be a subclass of \( T_a \);

- the method effectively invoked in the invocation of an mc\_parameter is a contracovariant of the method expected. Then invocation either computes a value which is subtype of the value expected or throws one exception among those expected. Moreover, invocation applies to arguments that have types that are subtypes of the signature of the effectively invoked method. Hence, in all the cases, invocation behaves well with any program context in which the invocation of a mc\_parameter occurs and which is correct with respect to Java ordinary type checking.

The translation semantics defined in this paper is a constructive formalisation of this semantics using Java structures: It replaces code containing definitions and invocations of mc\_parameters with code that contains the invocations of the methods that are effectively bound to such mc\_parameters according to the semantics. Moreover, it is based on the computational structures of the callback methodology (Gamma et al., 2005).

5.2 The Callback Methodology: A Semi-Formal Description

Callback is a way to pass executable code to procedures in event driven programming. In OO languages callback is implemented through function objects (Meyer, 2004; Horstmann, 2007) that wrap a code in order to treat it as an ordinary datum that can be passed anywhere in the program and unwrap it when the code must be executed. The methodology, here discussed for objects wrapping methods, method objects, can be summarised in four points.

1. Interfaces for Method Objects. We introduce an interface representing classes of objects wrapping methods, that can be passed, in the program, as parameters of HO methods. In fact, we need a family
of interfaces ApplyClass$nk, where the suffixes $n$ and $k$ stand for the number of arguments and for the number of different types of throwable exceptions of the wrapped methods for which the interface is defined. Each interface has only one method Apply whose first parameter is an object, namely the object on which the wrapped method must be invoked, while the following $n$ parameters are the $n$ parameters to which the wrapped method applies. Eventually, Apply throws the same $k$ different classes of exceptions of the wrapped method. Interface ApplyClass$nk has $n+k+2$ type variables: In addition to the $n$ type variables for the types of the arguments and the $k$ for those of the throwable exceptions, it has one type variable for the class of objects on which the wrapped method may be invoked, and one type variable for the type of the result that the wrapped method computes. Hence, interface ApplyClassV$nk has $n+k+1$ type variables and is the analog of ApplyClass$nk for void wrapped methods.

public interface ApplyClass$nk<RT,ET1,...,ETn,T,
   QI1 extends Throwable,...,QIk extends Throwable>{
   public T apply(RTo,ET1,x1,...,ETn,xn) throws QI1,...,QIk;
}

public interface ApplyClassV$nk<RT,ET1,...,ETn,
   QI1 extends Throwable,...,QIk extends Throwable>{
   apply(RTo,ET1,x1,...,ETn,xn) throws QI1,...,QIk;
}

2. Classes of Method Objects. For each use of the method, which is to be passed as parameter, a class, which implements ApplyClass$nk (resp. ApplyClassV$nk, if it is a void method) and consequently Apply, must be defined. Apply invokes the method on the object passed to it as first argument, with the arguments passed in the rest of its arguments list. Suppose we have C<Ta1,...,Ta$p> (where Ta1,...,Ta$p are the type arguments supplied to the generic class C<Q1,...,Q$p>, of the program, with type parameters Q1,...,Q$p) having methods named m1,...,mq, which are to be passed as parameters in the program. Then q classes are defined. In particular, for each, possibly generic, mi, let ETami1,...,ETami$nu$ be the types of the arguments to which the method applies in the invocation, once passed as parameter, let Tmi be the invocation return type, Tmi1,...,Tmi$nu$ be the list of exception classes that invocation can throws. Then, a class I[m_i] is defined as follows:

static class I[m_i] implements ApplyClass$nmi<kmi$<C<Ta1,...,Ta$p>,
   ETami1,...,ETami$nu$,Tmi1,...,Tmi$nu$ > {
   public Tmi Apply(C<Ta1,...,Ta$p> o, ETami1,x1,...,ETami$nu$,xn)  
       throws Tmi1,...,Tmi$nu$ {
       return (o.m_i(x1,...,xn));
   }
}

3. HO Methods. Every HO method hm is defined having one ApplyClass$nk parameter o for each object wrapping one of the method mi which will be bound to it during an invocation of the HO method. This is expressed as below

public ET hm (...ApplyClass$nk<ET,E1,...,ETn,T,T1,...,Tk> o...)  
   {
   ... o.Apply(E,E1,...,En) ...
}

where E is the expression on which one of the methods mi must be invoked and E_j (j ∈ [1,n]) are the arguments of mi.

2 I[m_i] is a class identifier which must be unique for each different use of method mi as parameter in the HO methods of the program. Then, the specific name chosen for I[m_i] depends from the method identifier, the list of types used in the invocation and the class in which mi is defined, and the technique used for the definition of the class: inner, anonymous or stand alone.
4. **Creation of Method Objects.** The invocation of `hm` requires the construction of one object of the class `I[m_i]`. This is accomplished in the invocation below, under the assumption, in point 2, that class `I[m_i]` is defined as an inner class of class `C`:

```
... E.hm(...new C.I[m_i]()...)
```

Point 4 concludes the description of the callback methodology that has been presented in the case that the passed methods are void and non-void object and class (fixed arity) methods. The latter ones are considered only if the methods are invoked on objects of the class. If the passed class method is invoked directly on the identifier of the class: point 1 requires a specific interface `ApplyClassS$nk` (resp. `ApplyClassSV$nk`) without the first parameter; point 2 has method `Apply` without first argument and the occurrence of `o` in the body replaced by the name of the class that contains the passed static method; point 3 has invocation of `Apply` without first argument. Also, if the passed method is a *variable arity* method interfaces change and point 1-3 are modified to consider the use of the syntactic form `'...'` for the occurrence of a *variable arity parameter* (see Section 8.4.1 (Gosling et al., 2005)). Moreover, different solutions can be considered for the definition of the classes of objects wrapping a method, given in point 2. In the presentation given above we use *inner classes* but *anonymous classes* and *stand alone classes* offer alternative supports for this aim with different advantages as discussed in (Bellia & Occhiuto, 2009b). In particular, the use of anonymous classes avoids the need of introducing names for the classes (as for `I[m_i]` in point 2 of the methodology). Hence, unlike the one presented in (Bellia & Occhiuto, 2009b), the translation presented in next section uses anonymous classes and combines points 2 and 4.

5.3 **E[^][]_ρ:** A Callback Based, Translation Semantics of MC_ parameters

The translation semantics is defined using the syntax directed rule system described in Section 3. Hence, it has at least one rule for each production of the unambiguous grammar of Appendix, and the translation computes applying to each program construct the rule that is associated to the production with which the construct is parsed. However, the rule system `E[^][]_ρ` contains only 8 significant rules and the *otherwise* metarule of Section 3. Hence we illustrate the translation discussing rule by rule (only for the significant ones) how the rules work. However, firstly we introduce the parameter `ρ` used in the translation `E[^][]_ρ`. This parameter is an environment for the application of the translation to a construct: It contains one binding for each parameter (of constructor or method) that has the considered construct in its (lexical) scope. The empty environment is expressed by `Ω` and does not contain bindings. Any other environment is obtained by `R[[T I]]_ρ` that adds a new binding `[[T I]]` to a given environment `ρ` (possibly, the empty environment): The new binding associates to the name `I` a type `T`. Environments are used in the translation in order to discover which identifiers, in the considered program construct, are parameters that are bound to a `FType`, hence are `mc_` parameters (see the two rules 7 and 8 for `PrimarySelector` in Figure 5).

*CompilationUnit:* `CU`. Rule 1 applies to constructs of the category `CompilationUnit` and resets to empty, `Ω`, the environment of the translation of each `ClassOrInterfaceDeclaration` construct which occurs in the `CompilationUnit`, since no method parameter can have such constructs in the scope.

*MethodOrConstructorDecl:* `MB`. Rule 2 extends the environment of the translation of the *block* with the method parameters, since the block is the method body and such parameters have it in their scope.

*ExtendedType:* `ET`. Rules 3 and 4 apply to type expressions of the new category `FType` and replace the type with the type interface `ApplyClassS$nk` and `ApplyClassSV$nk` used to model method objects as described in Section 5.2, point 1.
Expression: $E$. Rules 5 and 6 apply to the value expressions $AExp$ that compute an actual mc parameter. The rules replace the expression with an instance creation expression of an anonymous class. Similarly to what we did in Section 5.2 using inner classes, the two rules insert the code to create an anonymous class that implements the appropriate interface, (point 2), and, at the same time, to create a method object that instantiates such a class, (point 4). Hence, rule 5 considers each expression $AbsI\ EL\rightarrow T[\text{throws } TL] \from T_c$ computing a non void method. It replaces the expression with the code that creates, (point 4), a method object wrapping a method that has name $I$, belongs to a class in the hierarchy rooted at class $T_c$, applies to argument type list $TL$ and has return type $T$. Analogously the rule 6 creates a void method object.

PrimarySelector: $PS$. The last two rules, 7 and 8, apply to value expressions of the category PrimarySelector. This category reformulates, in an unambiguous way, the syntactic structures defined, in the grammar of JSL (Gosling et al., 2005), combining the two categories Primary and Selector. The two rules consider each invocation $PS\. I\. A$ where a formal mc parameter $I$ is invoked on expression $PS$ and applies to the arguments in $A$. Since, in the translated program, $I$ is bound to an object wrapping the method to be invoked, the rules replace the invocation with an invocation on $I$ of method $Apply$ that applies to the argument list containing the translation of $PS$ followed by the translations of the arguments in $A$. The translation of the arguments as well as that of the entire invocation, in rule 8, are cast to the expected types, in order to guarantee the contravariance of the wrapped method (see Theorem 2).

The following properties hold for: (i) the translation semantics of mc parameters, (ii) the rule system with which the semantics is defined, (iii) the programs obtained applying the rule system. The properties assert that the wrapped methods are essentially the methods of the intended semantics of mc parameters. Existence and access of the methods passed using mc parameter in the source program can be checked, at compile time, checking existence and access of the wrapped methods in the translated program. The methods that are passed by mc parameters are checked to be contravariant with the expected methods, hence their invocation is type safe. Eventually, since the translation semantics $E[\cdot]_\rho$ gives meaning to the types of the extended source language mapping them into the types of the target language, Java 1.5, Theorem 3 asserts that in order to check programs of the extended language, for type correctness, it is enough to check the corresponding translated programs of Java 1.5 for type correctness (using the type system of the target language).

Theorem 2 (Wrapped Methods for MC.parameters). Let $E \equiv Abs\ I\ EL\rightarrow T[\text{throws } TL] \from T_a$ in the scope of an environment $\rho$, and $EL \equiv E[\cdot]_\rho$. Let $M$ be the wrapped method of the object, if any, created executing $E[\cdot]_\rho$. Then $M$ is (possibly an overriding of) $M(I,EL,T,TL,T,T_a)$, where $EL_{T_a}$ and $TL_{T_a}$ are the overloading resolution, in class $T_a$, of an invocation of a method with name $I$, argument type list $EL$, return type $T$ and exception type list $TL$. 

Lemma 1 (Existence and Access of MC.parameters). Let $E \equiv Abs\ I\ EL\rightarrow T[\text{throws } TL] \from T_a$ in the scope of an environment $\rho$, and $EL \equiv E[\cdot]_\rho$. Let $M(I,EL_{T_a},T,TL_{T_a},T_a)$ be the method that results, for any given $T_c \leq T_f \leq T_a$, from the intended semantics of $E$. Then $M(I,EL_{T_a},T,TL_{T_a},T_c)$ (i) exists if and only if $E[\cdot]_\rho$ has no compile-time type errors; (ii) can be invoked only if the $E[\cdot]_\rho$ has no method access violations.

Lemma 2 (MC.parameters are Contravariant). Let $p$ be a mc parameter of a HO method (in a source
program) and \( \text{Fun } RT_p : EL_p \rightarrow T_p \) be its type with \( EL_p \equiv ET_{p_1}, \ldots, ET_{p_n} \). Let \( o \) be a binding for \( p \) in an invocation of the HO method in the translated program obtained by \( E[[P]]_\varnothing \). Then, \( o \) is an object wrapping a method which is contracovariant with \( EL_p \rightarrow T_p \), i.e. has return type which is a subtype of \( T_p \) and type signature \( ET_{o_1}, \ldots, ET_{o_n} \) such that \( ET_{o_i} \) is a supertype of \( ET_{p_i} \) (\( \forall i \in [1..n] \)).

**Theorem 3** (Type Safety). Let \( P \) be any Java program extended with \texttt{mc}_\texttt{parameters}. Then \( P \) is type safe only if its translation \( E[[P]]_\varnothing \) is a type safe program (of Java), i.e. (i) it is well-typed and, (ii) its execution does not generate untrapped errors.

### 5.4 Example

The example in Figure 6,7 is a classical problem for higher order programming already discussed in (Bellia & Occhiuto, 2004), but here modified to use generics and classes in Java APIs such as 
\texttt{LinkedList} and \texttt{ListIterator}. The example defines an abstract class \texttt{Shape} for geometric shapes and several concrete classes \texttt{Rectangle}, \texttt{Circle}, \texttt{Triangle} etc. and an extension of \texttt{LinkedList} in which a HO method \texttt{map} is defined. The HO method is used to compute the list of areas of all the shapes contained in a list, even though the method to compute the area is a different one according to the type of the element in the list that is being processed.

### 6 Putting Translations Together

We have introduced and discussed, in a separate way, the extension of Java 1.5 with \texttt{mc}_\texttt{parameters} and the extension of Java 1.5 with closures. For each extension, we have followed the same methodology and used the same techniques, obtaining two separated translation semantics. One of the most interesting aspects of our project is how the two extensions can be integrated in order to obtain only one language which extends Java with both \texttt{mc}_\texttt{parameters} and closures. Before delving into this aspect, we consider two properties of the semantics we have defined. The first one is concerned with each translation semantics, individually, and asserts the completeness of the translation: It guarantees that the translated programs are pure Java (1.5).

**Theorem 4** (Completeness I). Translation \( E[[P]]_\varnothing \) is complete, i.e. it maps each program of Java extended with closures into a program of ordinary Java 1.5 which is its semantics under \( E[[P]]_\varnothing \).

**Theorem 5** (Completeness II). Translation \( F[[P]]_\varnothing \) is complete.

The following property is concerned with the two translations together and says that they are one another orthogonal: Hence one definition does not affect the other one. In particular, modifications on one translation could be treated without any consideration on the other one.

**Theorem 6** (Orthogonality). The composition of the two translations is commutative, i.e.:

\[
F[[E[[P]]_\varnothing]]_\varnothing = E[[F[[P]]_\varnothing]]_\varnothing,
\]

The language, obtained integrating the two extensions, allows to define programs that contain: closures having also \texttt{mc}_\texttt{parameters}, and vice versa, HO methods having closures as parameters, or computing
\[ E[[CU]]_p = [[\text{At} \ \text{package} \ QI_1; \ IM \ E[[CD_1]]\emptyset \ldots E[[CD_n]]\emptyset \]
  \quad \text{with} \ CU = [[\text{At} \ \text{package} \ QI_1; \ IM \ CD_1 \ldots CD_n]] \]
\]

\[ E[[MB]]_p = D [Q] [TV] I E[[FP]]_p O_0 TR E[[B]]_p|_{|_pFP} \quad \text{with} \ MB \in \text{MethodOrConstructorDecl} \]

\[ E[[ET]]_p = \begin{cases} 
\text{new ApplyClass}\_\&\text{nk} <E[[RT]]_p, E[[ET_1]]_p, \ldots, E[[ET_n]]_p, E[[T_o]]_p, E[[QI_1]]_p, \ldots, E[[QI_k]]_p > \\
\text{with} \ ET = \text{Fun} \ RT: ([ET_1, \ldots, ET_n]) \rightarrow T_o [\text{throws} QI_1, \ldots, QI_k] 
\end{cases} \]

\[ E[[E]]_p = \begin{cases} 
\text{new ApplyClass}\_\&\text{nk} <TL_\& > () \quad \text{with} \ E = \text{Abs} I ([EL]) \rightarrow TL [\text{throws} TL] \text{ from} T_c \\
\text{public} T \text{Apply} FP_p [\text{throws} TL] \\
\text{return} o J (x_1, \ldots, x_n) \\
\text{where} \\
TL_o = T_c, E[[ET_1]]_p, \ldots, E[[ET_n]]_p, T_o, QI_1, \ldots, QI_k \\
FP_o = (T_c, o, E[[ET_1]]_p x_1, \ldots, E[[ET_n]]_p x_n) 
\end{cases} \]

\[ E[[PS]]_p = \begin{cases} 
(T) (I \text{Apply} (E[[PS_1]]_p, (ET_1) E[[E_1]]_p, \ldots, (ET_n) E[[E_n]]_p)) \\
\text{with} \ PS = PS_1, I ([E_1, \ldots, E_n]) \land p(I) = \text{fun} RC; [ET_1, \ldots, ET_n] \rightarrow T \ TR \\
I \text{Apply} (E[[PS_1]]_p, (ET_1) E[[E_1]]_p, \ldots, (ET_n) E[[E_n]]_p) \\
\text{with} \ PS = PS_1, I ([E_1, \ldots, E_n]) \land p(I) = \text{fun} RC; [ET_1, \ldots, ET_n] \rightarrow \text{void} TR 
\end{cases} \]

\[ \begin{align*}
\mathcal{R}[[T I]]_p(x) &= T \quad \text{if} \ I = x \land T \in F\text{type} \\
\mathcal{R}[[T I]]_p(x) &= \bot \quad \text{if} \ I \neq x \land T \notin F\text{type} \\
\mathcal{R}[[T I]]_p(x) &= p(x) \quad \text{if} \ I = x \\
\emptyset(x) &= 1 \\
\rho \uparrow () &= \rho \\
\rho \uparrow (P_1, P_2, \ldots, P_n) &= \mathcal{R}[[ET_1 I_1]]_p \uparrow (P_1, P_2, \ldots, P_n) \text{ with } P_i = [\text{final} / \text{At} i ET i I_i [\text{type}]] 
\end{align*} \]

Legend: For arbitrary index \(p<\), \(A\in \text{Arguments}\), \(A\in \text{Annotations}\), \(B,B_p \in \text{Block}\), \(BB,B_B \in \text{BlockStatements}\), \(BS,BS_p \in \text{BlockStatement}\), \(BT \in \text{BasicType}\), \(CB \in \text{ClassBody}\), \(CD,CD_p \in \text{ClassOrInterfaceDeclaration}\), \(CN \in \text{CreatedName}\), \(CRC \in \text{ClassCreatorRest}\), \(CU \in \text{CompilationUnit}\), \(D,D_p \in \text{ModifierOpt}\), \(E,E_p \in \text{Expression}\), \(EL,EL_p \in \text{ExtendedTypeList}\), \(ET,ET_p \in \text{ExtendedType}\), \(F,F_p \in ITs\), \(FP,FP_p \in \text{FParameters}\), \(FS,FS_p \in [\text{final} | \text{shared}]\), \(GS \in \text{ExplicitGenericInvocationSuffix}\), \(I,I_p \in \text{Identifier}\), \(IM \in \text{ImportDeclaration}\), \(IO \in \text{InfixOp}\), \(K \in \text{Literal}\), \(L \in \text{LocalVariableDeclarationStatement}\), \(MB,MB_p \in \text{MemberDecl}\), \(N \in \text{NonInfixSelector}\), \(O_i \in [\ast]_{i=0}^{n}\), \(P,P_p \in \text{FormalParameter}\), \(PO \in \text{PostfixOp}\), \(PS,PS_p \in \text{PrimarySelector}\), \(P_e \in \text{Primary}\), \(Q,Q_p \in TPs\), \(Q \in \text{QualifiedIdentifier}\), \(RT \in \text{RootClass}\), \(S,S_p \in \text{Statement}\), \(Sp,Sp_p \in \text{Selector}\), \(T,T_p \in \text{Type}\), \(T_o \in \text{TypeArgument}\), \(T\in \text{ParsOpt}\), \(TL,TL_p \in \text{TypeList}\), \(TR \in \text{ThrowOpt}\), \(TV,TV_p \in [\text{Type} | \text{void}]\), \(U,U_p \in \text{ST}\), \(V,V_p \in \text{VariableDeclaratorId}\).

**Figure 5:** MC parameters: \(E[[\cdot]]_p\) Translation Semantics.
public abstract class Shape {
    public abstract Double Area();
    public abstract Double Perimeter();
}

public class Rectangle extends Shape {
    private double base;
    private double height;
    public Rectangle(double b, double h){base=b;height=h;}
    public Double Perimeter() {return new Double(2*base+2*height);}
    public Double Area() {return new Double(base*height);}
}

public class Circle extends Shape {
    private double radius;
    public Circle(double r){radius=r;}
    public Double Perimeter() {return new Double(2*3.14*radius);}
    public Double Area() {return new Double(3.14*radius*radius);}
}

public class Triangle extends Shape {...}

public class FList <C> extends LinkedList <C> {
    public <T> FList<T> Map(Fun C:() -> T s){
        FList<T> L=new FList<T>();
        for (C g: this) L.add(g.s());
        return L;}
}

public class Compute {
    public static void main (String [] args){
        FList <Shape> L= new FList<Shape>();
        ... L.add(new Rectangle(3.0,4.2)); L.add(new circle(1.5));...
        ... L.Map(Abs Area ()->Double from Shape); }}

Figure 6: HO Programming: A Program $P$ using a Class of Geometric Shapes

// first 15 lines are unchanged
public class FList <C> extends LinkedList <C> {
    public <T> FList<T> Map(ApplyClass$00<C,T> s){
        FList<T> L=new FList<T>();
        for (C h: this) L.add((T)(s.Apply(h)));
        return L;}
}

public class Compute {
    public static void main (String [] args){
        FList <Shape> L= new FList<Shape>();
        ... L.add(new Rectangle(3.0,4.2));L.add(new circle(1.5));...
        ... L.Map(new ApplyClass$00<Shape,Double>(){
            public Double Apply (Shape o){
                return o.Area(); })};
    }
    public interface ApplyClass$00 <T,S> {
        public S Apply(T m);}
}

Figure 7: HO Programming: The Translated Program – $\mathcal{E}[[P]]\rho$
a closure as return value. If we look at the syntactic extensions, introduced for the closures and those introduced for the mc_parameters, we see that these extensions already are designed to be merged into a grammar, without conflicts or ambiguities. Therefore, the grammar of the integrated language is simply obtained putting together all the productions, introduced, separately, by each extension. At the semantics level, on the other hand, we resort to the translation semantics we have defined through the rule system \( F[[\tau]] \) and through the rule system \( E[[\tau]]_\rho \) in order to obtain, for every production of the integrated grammar, the translation semantics of the integrated language. The translation rules of the integrated language must apply, to each production of the integrated grammar, the rule of \( F[[\tau]]_\tau \) and/or that of \( E[[\tau]]_\rho \), depending on if \( F[[\tau]]_\tau \) and/or \( E[[\tau]]_\rho \) is defined for such a production. Then, we see that, in correspondence to a production \( p \) of the form \( C_0 ::= C_1 \ldots C_n \), four cases arise. Let \( E F[[\tau]]_\rho \) be the translation semantics of Java extended with HO methods, mc_parameters and closures. Then, we define \( E F[[\tau]]_\rho \) examining, case by case, how to compose translations \( E[[\tau]] \) and \( F[[\tau]]_\tau \) in each of the four cases.

Case 1. Only one, between \( F[[\tau]]_\tau \) and \( E[[\tau]]_\rho \), is defined on \( p \). This is the case for rules 3, 4, 5, 6, 7, 8 of \( E[[\tau]]_\rho \), and rules 2, 3, 4, 5, 6, 7, 8, 9, 11, 12 of \( F[[\tau]]_\tau \). All these rules become rules of \( E F[[\tau]]_\rho \) when each occurrence of \( E[[\tau]]_\rho \) and \( F[[\tau]]_\tau \) are replaced by \( E F[[\tau]]_\rho \) for any argument \( U \).

Case 2. \( F[[A_0]]_\tau = r_i(A_1, F[[A_1]]_\tau), \ldots, A_n, F[[A_n]]_\tau \) with \( b_i(A_0, A_1, \ldots, A_n) \)
\( E[[A_0]]_\rho = f(A_1, E[[A_1]]_\rho), \ldots, f(A_n, E[[A_n]]_\rho) \) with \( b_i(A_0, A_1, \ldots, A_n) \)
(Symmetric case: changing \( F[[\tau]]_\tau \) with \( E[[\tau]]_\rho \)) This is the case of the rule 1 of \( E[[\tau]]_\rho \), and of the rule 1 of \( F[[\tau]]_\tau \). In fact, both rules would be otherwise rules if it were not for the environments that change. Each rule becomes a rule of \( E F[[\tau]]_\rho \) provided that \( E[[\tau]]_\rho \) is replaced by \( E F[[\tau]]_\rho \) and \( F[[\tau]]_\rho \) by \( E F[[\tau]]_\rho \) for any \( U \) and environment \( \rho \) changing into \( \sigma \) and \( \tau \) changing into \( \theta \).

Case 3. \( F[[A_0]]_\tau = f(A_1, F[[A_1]]_\tau), \ldots, f(A_n, F[[A_n]]_\tau) \) with \( b_i(A_0, A_1, \ldots, A_n) \)
\( E[[A_0]]_\rho = f(A_1, E[[A_1]]_\rho), \ldots, f(A_n, E[[A_n]]_\rho) \) with \( b_i(A_0, A_1, \ldots, A_n) \)
This is the case of the otherwise metarule of the two systems which is replaced by the otherwise metarule of \( E F[[\tau]]_\rho \).

Case 4. \( F[[A_0]]_\tau = r_i(A_1, F[[A_1]]_\tau), \ldots, A_n, F[[A_n]]_\tau \) with \( b_i(A_0, A_1, \ldots, A_n) \)
\( E[[A_0]]_\rho = f(A_1, E[[A_1]]_\rho), \ldots, f(A_n, E[[A_n]]_\rho) \) with \( b_i(A_0, A_1, \ldots, A_n) \)
(Symmetric case: changing \( F[[\tau]]_\tau \) with \( E[[\tau]]_\rho \)) Differently than we would expect, since Theorem 6 on orthogonality, this case contains two rules: Rule 2 of \( E[[\tau]]_\rho \) and rule 10 of system \( F[[\tau]]_\tau \). Nevertheless, rule 2 is not an otherwise metarule because of the environment that changes but, like an otherwise metarule, its \( r_i \) constructor is the identity constructor, hence the rule leaves unchanged the structure of \( MB \) and applies the translation to the components. Then, \( E F[[\tau]]_\rho \) contains only one rule. The rule is rule 10 of \( F[[\tau]]_\tau \) where: (i) \( F[[MB]]_\tau \) (on the left side) is replaced by \( E F[[MB]]_\rho \); (ii) \( F[[TV]]_\tau \) is replaced by \( E F[[TV]]_\rho \); (iii) \( F[[BB]]_\tau \) is replaced by \( E F[[BB]]_\rho \).

**Theorem 7** (Completeness). Translation \( E F[[\tau]]_\rho \) is complete: It maps each program of the extended language into a program of ordinary Java 1.5 which is its semantics under \( E F[[\tau]]_\rho \).

### 7 Conclusions

In this chapter we extend Java with two HO mechanisms: mc_parameters and closures, for which we define a translation semantics and prove some fundamental properties including completeness and orthogonality.
The implementation of the extended language, along the lines already experimented (Bellia & Occhiuto, 2008b), is immediate: The translation semantics $\mathcal{EF}[u]_{\rho,\tau}$ can be formally converted into a source-to-source translation and implemented as a one-pass preprocessor (Aho et al., 2007) that can be quickly developed using Lex & Yacc (Levine et al., 1995) and GNU Bison (Donnelly & Stallman, 2006). The prototype used in combination with any Java compiler, including Javac of SUN, GCJ of GNU, ECJ of Eclipse, allows to test the programming features of the extended language. Moreover, for competitive compilations, the compiler modifications can be obtained, in a straightforward way, from the translation semantics: The abstract syntax generation phase of the compiler parser creates the abstract tree of $\mathcal{EF}[u]_{\rho,\tau}$ in correspondence of the parsing of any language structure $u$, for suitable contexts (producing the environments) $\rho$ and $\tau$. We plan to investigate the possibility to apply software engineering techniques and Java annotations to support error localisation in non-native constructs. The idea is that errors detected in a target program $P_t$ by the Java compiler are recognised as errors on a non-native construct and localised (for possibly recovery) on the source program $P_s$, from which $P_t$ has been obtained (by preprocessing). Eventually, as mentioned in section 4, we are still investigating the definition of core Java sub-languages with reduction semantics. Such approach allows to define different modelizations of a same construct, for instance closures, and prove their properties. In this way, it is possible to prove that the reduction semantics and the translation semantics of the sub-language commute, hence prove that the properties are maintained in the full language.

**Appendix - An Unambiguous Grammar for Java**

The syntactic categories that are used but not defined are those in chapter 18 of the Java JSL (Gosling et al., 2005). We use the same syntactic conventions of JSL, in particular for the alternation operator ($a/b$ means one occurrence of either $a$ or $b$) and for option operator ($[a]$ means zero or one occurrence of $a$).

ClassDeclaration ::= public class Identifier [TPs] [ST] [ITs] ClassBody
TPs ::= <TypeParameter (, TypeParameter)>^ ST ::= extends Type ITs ::= implements TypeList ClassBody ::= {(MemberDecl)*}
MemberDecl ::= ;
   | [static] Block
   | Modifier* FieldDeclarator
   | MethodOrConstructorDecl
   | ClassOrInterfaceDeclaration
FieldDeclarator ::= Type Identifier VariableDeclaratorRest
MethodOrConstructorDecl ::= Modifier* [TPs] [TPs] [(void|Type)] Identifier FParameters [*]
   | ThrowOpt (Block | ;)
ThrowOpt ::= { throws QualifiedIdentifier (, QualifiedIdentifier)*
ExtendedType ::= Type | FType
   | ParameterizedType | BasicType | ClosType
Type ::= ParameterizedType | BasicType | ClosType
ExtendedTypeList ::= ExtendedType (, ExtendedType)*
FParameters ::= ( FormalParDecls )
FormalParDecls ::= [final | shared] [Annotations] ExtendedType FormalParDeclsRest
FormalParDeclsRest ::= VariableDeclaratorId [, FormalParDecls]
   | ... VariableDeclaratorId
ClosType ::= ( (ExtendedTypeList) ) (void|Type) ThrowOpt
ParameterizedType ::= Identifier ParsOpt ( Identifier ParsOpt )^ [*]
FType ::= fun RootClass ::= ( [ExtendedTypeList] ) → (void|Type) ThrowOpt
RootClass ::= ParameterizedType

```
ParsOpt::= [<TypeArguments>*] 
Expression::= AExp / Expression1 / AssignmentOperator Expression1 
Expression3::= PrefixOp Expression3 
            | (Type) Expression3 
            | PrimarySelector PostfixOp * 
PrimarySelector::= PrimarySelector NonInvocationSelector 
            | [Primary . [<TypeList>] Identifier Arguments] 
            | Primary 
            | ParExpression / Literal 
            | [<TypeList>] (ExplicitGenericInvocationSuffix / this Arguments) 
            | this [Arguments] 
            | new [<TypeList>] CreatedName CreatorRest 
            | QualifiedIdentifier [][]* . class 
            | BasicType [][] . class | void . class 
CreatorRest::= ArrayCreatorRest / ClassCreatorRest 
Closure::= { [ FParameters: (Type | void) ThrowOpt => Block ] } 
AExp::= abs MethodSpecifier from Type 
MethodSpecifier::= Identifier (ExtendedTypeList) -> (void | Type) ThrowOpt 
LocalVariableDeclarationStatement::= [ final | shared ] Type VariableDeclarators 
NonInvocationSelector::= . this 
            | . super (Arguments | . Identifier [Arguments]) 
            | . new [<TypeList>] Identifier ClassCreatorRest 
            | [ Expression ] 
            | . invoke [Arguments] 

References


Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2005). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.


