JavaΩ: Higher Order Programming in Java – The Technical Complements

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

Contents

1 Introduction **2**

2 Translation Semantics and Source to Source Translations: Principles **2**

3 Closures: Lambda Expressions in Java **4**
3.1 Closure basic Structure 4
3.2 Syntax of Closures. 4
3.3 Semantics of Closures. 5
3.4 The Translation Semantics $\mathcal{F}[\![]\!]_\tau$. 6
3.5 The Formal Definition of $\mathcal{F}[\![]\!]_\tau$ 8
Theorem: `this` Transparency 9
3.6 Closure Sub-typing 10

4 Higher Order Methods: MC_parameters. **11**
4.1 Syntax and Semantics of MC_parameters 11
4.2 The Callback Methodology: A semi-Formal Description 13
4.3 $\mathcal{E}[\![]\!]_\rho$: A Callback based, Translation Semantics of MC_ parameters . . . 14
Theorem: Wrapped Methods for MC_parameters 16
Lemma: Existence and Access of MC_parameters 16
Lemma: MC_parameters are Contracovariant 16
Theorem: Type Safety 16

5 Examples **16**
5.1 HO Programming with a Class of Geometric Shapes 16
5.2 Writing a Method that Maps Closures into Memozed Closures 19

6 Putting Translations Together **19**
Theorem: Completeness $\mathcal{E}[\![]\!]_\emptyset$ 20
Theorem: Completeness $\mathcal{F}[\![]\!]_\emptyset$ 20
Theorem: Orthogonality 20
Theorem: Completeness 21

7 Conclusions **21**

A Appendix - An Unambiguous Grammar for Java **21**

B Appendix - Proofs of Lemmas and Theorems **23**

C Appendix - MC_parameters: Class Literal and Variable Arity Methods **24**
C.1 The Type Constructor `funClass` and the Value Costructor `absClass` . . 24
C.2 Semantics Structures for `funClass` and `absClass` 25
C.3 $\mathcal{E}[\![]\!]_\rho$: Translation Semantics for `funClass` and `absClass` 27

1. INTRODUCTION

The paper contains the technical complements, of a paper that describes the extensions of Java 1.5 with Higher Order 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 are passed as arguments, and code as first class value, that can be assigned to variables, passed as argument, returned by method invocations, and executed in different points of a Java program. The main motivations are code reusability [BT96] and code expressivity [Fel91]. Main aims are [Gaf07; BO09b; BO09a]. Several languages integrate Object Oriented (OO) and HO features: C# [WH03], F# [Sym03], J# [Lak04], Python [Com10], Scala [Ode10], Ruby [FM08]. All such languages allow to define closures and a few ones, $C\#, F\#, J\#$, Scala, also to pass methods as arguments but the construct features are different according to the specific language structures. Hence, the project JavaΩ is motivated to study solutions for the design and implementation of HO constructs for and based on the structure of language Java. The project originated in 2004 [BO04], other results are in[BO05; BO07b; BO08c; BO08a; BO09b; BO09a; BO10]. The main topics considered in the project include 1) the structure of a closure as an anonymous function value; 2) the meaning of closure declaration (as a closure expression), of closure creation (as the closure value obtained evaluating a closure expression) and of closure evaluation (as the application of a closure value to a list of arguments for it); 3) the possibility of occurrences of non local variables and their value update; 4) the structure of closure type and 5) the resulting Java type system including closure sub-typing; 6) the use of closure in assignment, value return, and as arguments of closures and of methods; 7) the meaning of *this* (transparency vs. opacity); 8) the definition of recursive closures9 9) the meaning of methods when used as method values in HO method arguments; 10) the types of the (formal) parameters that can be bound to method values; 11) the form of expressions of the (actual) parameters when they specify a class method value or an object method value; 12) the interaction between closures and method values; 13) the extension of Object Calculus models [AC96; ABW01] for the study of the properties of method values and 14) of closures; 15) the study of implementation techniques for the Java extensions that ensure semantics correctness, allow rapid prototyping, preserve Java re-targetability. For details see the companion paper or the cited papers.

2. 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). 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 [AGG$^+$80; KLV03], reverse engineering [War94].

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 $C_0 ::= C_1 \dots C_n$ of the grammar, with, $C_0, \dots, C_n$, $n+1$ (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 [GJSB05]). 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 $\Sigma_{\mathcal{E}}^{G}$ be the rule system that defines $\mathcal{E}$ using grammar G. Then, in correspondence to production p, $\Sigma_{\mathcal{E}}^{G}$ contains $k >= 1$ rules of the form:

$$\mathcal{E}(A_0) = r_i(A_1, \mathcal{E}(A_1), \dots, A_n, \mathcal{E}(A_n)) \quad \texttt{with}\ b_i(A_0, A_1, \dots, A_n)$$

Symbols $A_0, \dots, A_n$ are variables and are ranging on the structures of the set defined by the corresponding categories $C_0, \dots, C_n$. 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 $A_0, A_1, \dots, A_n$ 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 $A_1, \dots$ 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 $\mathcal{E}(A_1), \dots, \mathcal{E}(A_n)$ 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 characterizing the translation defined by rules system. The most part of rules have the following form:

$$\mathcal{E}(X_0) = f(X_1, \mathcal{E}(X_1)), \dots, f(X_n, \mathcal{E}(X_n))) \quad \texttt{with}\ b_i(X_0, X_1, \dots, X_n)$$

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,

$$\mathcal{E}(X_0) = f(X_1, \mathcal{E}(X_1)), \dots, f(X_n, \mathcal{E}(X_n))) \quad \textit{otherwise}$$

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.

3. CLOSURES: LAMBDA EXPRESSIONS 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 [Gaf08]) that can be computed by and passed to methods: They allow the definition of methods whose code, hence whose behavior, 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.

3.1. Closure basic Structure

A basic common structure for Java closures emerges from [BGGvdA08; CSC08; Rei09]. This structure can be described by the following characteristics: Closures

(1) encapsulate an arbitrary piece of Java code and generalize it through parameters (as anonymous functions [Lan66], 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 arguments, 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 behaviors (i.e. intended semantics) and programming uses (i.e. methodologies) for closures [Gaf08; CSC08], and includes those in [BLB06]. As a matter of fact, such a closure basic structure is adopted in [BO10] and also here.

3.2. Syntax of Closures.

A closure is syntactically an expression. The following example shows an expression which is 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`. Its evaluation results a closure value (closure for short) which is a function that applies to an an integer argument updating a non local variable of name `x` and returning the value of `x`, after incrementing it by 1.

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

The declaration of the return type in the closure definition follows the syntactic structure of method definition in Java and does not force the implementation of the closure construct to infer the return type as in [BGGvdA08]. However, we defer to type checking, of ordinary Java compilers, the burden of checking, in the translated programs, the correctness of the types declared in the closure definition. As a matter of fact, our translation maps closure definitions with incorrect types into programs with incorrect types too. A closure (value) has a type: A closure may be assigned to a variable, passed

as a parameter, returned as a value, invoked as a method (using the reserved selector `invoke`) – checking the correctness of the types involved in each of the above operations.

All the extensions needed to Java grammar are reported in Appendix A. 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* [Lan66] but unlike *functional abstractions*, closures 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 [LY96] at the time the closure is invoked, see [Tro08] and the use of the hashtable in the example in Fig. 4.a.

Example 3.1. In Fig. 4.a, the statement `return memo_f` of method `memoized` returns a closure that when invoked, would access and modify the hash table which is bound to the variable `table`

Since local variables are allocated in a Java frame [LY96] 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.

Example 3.2. The variable `table`, of Fig. 4.a, is declared in the body block of method `memoized`. When statement `return memo_f` is executed, the execution of `memoized` terminates and the frame allocated at runtime in the Java Virtual Machine, to contain the variable `table` is deallocated. Then no access to the value bound to variable `table` is allowed throw the machine stacks. Nevertheless, such a value must be accesses and modified at each invocation, in the program, of the closure bound to `memo_f`.

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 [Gaf08] 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 [LY96]), access and modification of the variable. Hence the syntax of both the local variable declarations and the formal parameters [GJSB00] is extended adding:

LocalVariableDeclarationStatement::= [`final` | `shared`] *Type VariableDeclarators*
FormalParameter::= [`final` | `shared`] [*Annotations*] *ExtendedType VariableDeclaratorId*

3.3. Semantics of Closures.

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 $\Im$, a reference type $\Im(ES_1, ..., ES_n, [QI_1, ..., QI_k,]ES)$. 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` $QI_1, ..., QI_k$] ⇒ $BB\}$, a *function object*, i.e. an object, of type $\Im(ES_1, ..., ES_n, QI_1, ..., QI_k, ES)$, that wraps the function $\mathcal{H}_{BB}$ below. Type $\Im(ES_1, ..., ES_n, [QI_1, ..., QI_k,]ES)$ 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 $\mathcal{H}_{BB}$ is the function that, applied to a n-tuple of values $\mathcal{V}_1, ..., \mathcal{V}_n$ of type $ES_1, ..., ES_n$ respectively, either computes a value $\mathcal{V}$ of type ES or fails depending on code BB_r, executed of in a frame where V_1 is bound to value $\mathcal{V}_1$ and,..., and V_n is bound to value $\mathcal{V}_n$. Eventually, BB_r is the ordinary Java code resulting from the sequence of statements (and declarations) of closure body BB in which free (i.e. non local) variables, if any, are bound to the corresponding `shared` (or `final`) variables of the lexical scope in which the closure occurs (and nested closures, if any, are interpreted according to the points (i)-(iv) of the present intended semantics). If the computation fails then either a checked exception of a type in $QI_1, ..., QI_k$) or an unchecked exception is thrown.

(iii) closure invocation E`.invoke`$(E_1, ..., E_n)$, where E is an expression computing a function object c of type $\Im(ES_1, ..., ES_n, QI_1, ..., QI_k, ES)$, the application of the function, that c is wrapping, to the n-tuple of values $\mathcal{V}_1, ..., \mathcal{V}_n$ that results from the evaluation of the argument list $E_1, ..., E_n$. Arguments $E_1, ..., E_n$ must be of the corresponding types $ES_1, ..., ES_n$.

(iv) `shared` variable (resp. parameter) V of type ET, possibly initialized with expression E, a variable of a class of *variable objects*, i.e. objects that wrap variables of a given type and are allocated in the heap. The wrapped variable is possibly initialized to the value of E. The `shared` variable V is then, accessed and modified through such a reference.

3.4. The Translation Semantics $\mathcal{F}[\![]\!]_\tau$.

The translation semantics is defined by translation $\mathcal{F}[\![]\!]_\tau$ in Fig. 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 $\Im$ of the closure intended meaning is implemented as a bijective function *ClosType* $\rightarrow$ *Interface*, between the closure type and the interface for objects wrapping functions of that type. The implementation of $\Im$ is based on a family of interfaces, indexed by a couple of naturals n, k, defined as follows:

. `interface I$nk`$<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 I$nkvoid`$<ET_1, ..., ET_n,$
QI_1 `extends Throwable`,...,QI_k `extends Throwable`$>\{$
`public void invoke`$(FS_1\ At_1 \qquad\qquad ET_1\ V_1, ..., FS_n\ At_n\ ET_n\ V_n)$
`throws` $QI_1, ..., T_k;\}$

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, $\mathcal{F}[\![]\!]_\tau$ translates:

(i) the closure type $\{(ET_1, ..., ET_n) : ET\ [$`throws` $QI_1, ..., QI_k]\}$ 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 $\{(FS_1 At_1\ ET_1 V_1, ..., FS_n\ At_n\ ET_n V_n) : ET\ [$`throws` $QI_1, ..., QI_k] \Rightarrow BB\}$ of type $\{(ET_1, ..., ET_n) : ET\ [$`throws` $QI_1, ..., QI_k]\}$ into an instance of the class implementing the corresponding interface, namely:

`new I$nk` $<ES_1, ..., ES_n, ES, QI_1, ..., QI_k>\ ()\{$
`public` ES `invoke`$(FS_1\ At_1\ ES_1\ V_1, ..., FS_n\ At_n\ ES_n\ V_n)$
`throws` $QI_1, ...QI_k\ \{BB_r\}\}$

$$\mathcal{F}[\![CB]\!]_\tau = \{\mathcal{F}[\![MB_1]\!]_\emptyset \ \ldots \mathcal{F}[\![MB_n]\!]_\emptyset\} \ \text{ with } CB = \{MB_1 \ \ldots MB_n\}$$

$$\mathcal{F}[\![Py]\!]_\tau = \begin{cases} \tau(Py) & \text{with } Py \quad \in \mathit{Identifier} \mid \texttt{this} \\ \texttt{new I\$n} <\mathcal{F}[\![ET_1]\!]_\tau, ..., \mathcal{F}[\![ET_n]\!]_\tau, \mathcal{F}[\![ET]\!]_\tau, QI_1, .. & \text{with } P_i \equiv FS_i\ At_i\ ET_i\ V_i \\ \quad .., QI_k>()\{\mathcal{F}[\![\texttt{public}\ ET\ \texttt{invoke}(P_1, \ldots, P_n)\ TR\{ & \wedge\ TR \equiv \texttt{throws}\ QI_1, ..., QI_k \\ \qquad\qquad BB\}]\!]_{\tau\uparrow(\{\texttt{this/(s\$self,on)}\})}\} & \wedge\ Py \equiv \{(P_1, ..., P_n) : ET\ TR \Rightarrow BB\} \\ \texttt{new I\$nvoid} <\mathcal{F}[\![ET_1]\!]_\tau, ..., \mathcal{F}[\![ET_n]\!]_\tau, QI_1, ..., & \text{with } P_i \equiv FS_i\ At_i\ ET_i\ V_i \\ \quad QI_k>()\{\mathcal{F}[\![\texttt{public void invoke}(P_1, \ldots, P_n)\ TR\ \{ & \wedge\ TR \equiv \texttt{throws}\ QI_1, ..., QI_k \\ \qquad\qquad BB\}]\!]_{\tau\uparrow(\texttt{this/(s\$self,on)})}\} & \wedge\ Py \equiv \{(P_1, ..., P_n) : \texttt{void}\ TR \Rightarrow BB\} \end{cases}$$

$$\mathcal{F}[\![ET]\!]_\tau = \begin{cases} \texttt{I\$nk} < \mathcal{F}[\![ET_1]\!]_\tau, \ldots, \mathcal{F}[\![ET_n]\!]_\tau, \mathcal{F}[\![ET_r]\!]_\tau\ TR> & \text{with } TR \equiv \texttt{throws}\ QI_1, ..., QI_k \\ & \wedge\ ET \equiv \{(ET_1, \ldots, ET_n) : ET_r\ TR\} \\ \texttt{I\$nkvoid} < \mathcal{F}[\![ET_1]\!]_\tau, \ldots, \mathcal{F}[\![ET_n]\!]_\tau\ TR> & \text{with } TR \equiv \texttt{throws}\ QI_1, ..., QI_k \\ & \wedge\ ET \equiv \{(ET_1, \ldots, ET_n) : \texttt{void}\ TR\} \end{cases}$$

$$\mathcal{F}[\![L]\!]_\tau \ = \begin{cases} [\,\texttt{final}\,]\ \mathcal{F}[\![ET]\!]_\tau\ I\,[= \mathcal{F}[\![E]\!]_{\tau\uparrow(I/\texttt{this})}] & \text{with} \quad L \equiv [\texttt{final}]\ ET\ I\,[= E] \wedge E \in \mathit{Closure} \\ \texttt{final}\ T\ I = \texttt{new}\ T([\mathcal{F}[\![E]\!]_\tau]) & \text{with} \quad L \equiv \texttt{shared}\ ET\ V\,[= E] \wedge E \notin \mathit{Closure} \\ & \wedge T = \ \texttt{C\$Shared} < \mathcal{F}[\![ET]\!]_\tau> \\ \texttt{final}\ T\ I = \texttt{new}\ T([\mathcal{F}[\![E]\!]_{\tau\uparrow(I/\texttt{this})}]) & \text{with} \quad L \equiv \texttt{shared}\ ET\ I\,[= E] \wedge E \in \mathit{Closure} \\ & \wedge T = \ \texttt{C\$Shared} < \mathcal{F}[\![ET]\!]_\tau> \end{cases}$$

$$\mathcal{F}[\![MB]\!]_\tau = \begin{cases} D\ Q\ \mathcal{F}[\![TV]\!]_\tau\ I(R_1, ..., R_n)O\ TR\{ \quad L_0\ L_1 \ldots L_n\ \ \mathcal{F}[\![BB]\!]_{\overline{\tau}}\ \} & \text{with } MB \equiv D\ Q\ \ TV\ I(P_1, ..., P_n)O\ \ TR\ \{BB\} \\ & \wedge\ \ P_i \equiv FS_i\ At_i\ ET_i\ V_i\ \wedge\ \ V_i \equiv I_i\,O_i \\ \textit{where}: & \\ \quad R_i \equiv FR_i\ At_i\ ER_i\ V_i \textit{ and } ER_i = \mathcal{F}[\![ET_i]\!]_\tau & \\ \quad \textit{and } L_0 = \texttt{final Object s\$self} \ = \texttt{this;} & \\ \quad \textit{and } L_i = \begin{cases} \texttt{final}\ T\ \mathcal{G}(I_i)O_i = \texttt{new}\ T(V_i); & \text{if } FS_i \equiv \texttt{shared} \\ & \wedge T \equiv \texttt{C\$Shared} < ER_i> \\ \lambda & \text{if } FS_i \in [\texttt{final}] \end{cases} & \\ \quad \textit{and } FR_i = \begin{cases} FS_i & \text{if } FS_i \in [\texttt{final}] \\ \lambda & \text{if } FS_i \equiv \texttt{shared} \end{cases} & \\ \quad \textit{and } E_i = \begin{cases} I_i & \text{if } FS_i \in [\texttt{final}] \\ \mathcal{G}(I_i).\texttt{value} & \text{if } FS_i \equiv \texttt{shared} \end{cases} & \\ \quad \textit{and } \overline{\tau} = \begin{cases} \tau \uparrow (\texttt{this}/(\texttt{s\$self}, \texttt{off}), I_1/E_1, ..., I_n/E_n) & \text{if } \tau(\texttt{this}) = \bot \\ \tau \uparrow (I_1/E_1, ..., I_n/E_n) & \text{if } \tau(\texttt{this}) = (\texttt{s\$self}, \texttt{on}) \end{cases} & \end{cases}$$

$$\mathcal{F}[\![BB]\!]_\tau = \mathcal{F}[\![L]\!]_\tau; \mathcal{F}[\![BB_r]\!]_{\tau\uparrow(I/I.\texttt{value})} \ \text{ with } BB \equiv S; BB_r \ \wedge L \equiv \texttt{shared}\ ET\ I([\,])^*\,[= E]$$

$$\mathcal{F}[\![E]\!]_\tau = \tau(I) = \mathcal{F}[\![E]\!]_{\tau\uparrow(I/\texttt{this})} \ \text{ with } \ E \equiv I = E_1 \wedge E_1 \in \mathit{Closure}$$

where:

$$\tau \uparrow (I_1/E_1, \ldots, I_n/E_n)(x) = \begin{cases} E_i & \text{if } x = I_i \equiv \wedge\ E_i \notin \{\texttt{(S\$self,on)}, \texttt{(S\$self,off)}\} \\ \texttt{s\$self} & \text{if } x = \texttt{this} = I_i \wedge E_i = (\texttt{s\$self}, \texttt{on}) \\ \texttt{this} & \text{if } x = \texttt{this} = I_i \wedge E_i = (\texttt{s\$self}, \texttt{off}) \\ \tau(x) & \text{if } x \notin \{I_1, \ldots, I_n\} \end{cases}$$

Legenda: For arbitrary index p, At, $\text{At}_p \in \mathit{Annotations}$, BB, $\text{BB}_p \in \mathit{BlockStatements}$, CB$\in$ $\mathit{ClassBody}$, D, $\text{D}_p \in \mathit{ModifierOpt}$, E, $\text{E}_p \in \mathit{Expression}$, ET, ET_p, $\text{ER}_p \in \mathit{ExtendedType}$, FS, FS_p, $\text{FR}_p \in [\texttt{final} \mid \texttt{shared}]$, x, x_p, y, I, $\text{I}_p \in \mathit{Identifier}$, L$\in$ $\mathit{LocalVariableDeclarationStatement}$, MB, $\text{MB}_p \in \mathit{MemberDecl}$, O, $\text{O}_p \in []^*$, P, P_p,$\text{R}_p \in \mathit{FormalParameter}$, Py$\in$ $\mathit{Primary}$, Q, $\text{Q}_p \in \mathit{TPs}$, QI$\in$ $\mathit{QualifiedIdentifier}$, S, $\text{S}_p \in \mathit{Statement}$, T, $\text{T}_p \in \mathit{Type}$, TR$\in$ $\mathit{ThrowOpt}$, TV, $\text{TV}_p \in [\mathit{Type} \mid \texttt{void}]$ V, $\text{V}_p \in \mathit{VariableDeclaratorId}$,

Fig. 1. Closures: $\mathcal{F}[\![]\!]_\tau$ Translation Semantics.

where BB_r results from the translation of BB and ES_i results from the translation of ET_i according to $\mathcal{F}[\![]\!]_\tau$.

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

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

`class C$Shared` $<ET>$ {ET `value; C$Shared (`ET `n)`{`value=n;`}}

Each access to a variable V, declared `shared`, is replaced by an access to V.`value`. Class `I$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 initialized to contain the value of the parameter. All occurrences of the parameter in the translation of BB are translated into accesses to the field `value` of the new variable. The function $\mathcal{G}$: *Identifier* $\rightarrow$ *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 $\mathcal{F}[\![]\!]_\tau$ preserves the method headers.

3.5. The Formal Definition of $\mathcal{F}[\![]\!]_\tau$

The translation semantics is defined using the syntax directed rule system described in Section 2. Hence, it has at least one rule for each production of the unambiguous grammar of Appendix A, 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 $\mathcal{F}[\![]\!]_\tau$ 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 illustrates the translation discussing rule by rule (only for the significant ones) how they work. Firstly we introduce the environment τ used as a parameter in the translation $\mathcal{F}[\![]\!]_\tau$, whose definition is contained at the end of Fig. 1. The environment τ associates an identifier $I \in$ *Identifier* to an expression $E \in$ *Expression*. The empty environment is denoted by $\emptyset$ and does not contain bindings. Any other environment is obtained by $\tau \uparrow (I_1/E_1, ...I_k/E_k)$ that adds k new bindings to a given environment τ (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, $(\forall I)\ \emptyset(I) = \perp$ where $\perp$ is the undefined term. The environment τ contains a binding for:

(1) all the `shared` local variables or `shared` formal parameters that have the construct (to which $\mathcal{F}[\![]\!]_\tau$ applies) in their (lexical) scope. In this case the value of the binding is I_i.`value` as explained in the previous subsection 3.4.

(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 `this` to the function object that the translation produces.

(3) `this`: Since the translation maps closures into function objects and occurrences of `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 `s$elf` and add a declaration `final Object s$elf = this` at the beginning of each method body BB see the fifth rule. The environment τ is consequently

extended to support an on-off mechanism which says when $\mathcal{F}[\![]\!]_\tau$ has to replace the self reference `this` with the reference to the object bound to variable `s$elf`. In this way, in the translated program, `this` in the method `Apply`, refers to the function object implementing the closure and allows recursive invocations, while the object on which the method that defines the closure is invoked is reached through the reserved identifier `s$elf`.

THEOREM 3.3 (`this` TRANSPARENCY). *Occurrences of* `this` *in a closure are references to the object on which the method that defines the closure is invoked.* [1] □

The rules are placed in Fig. 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 *ClassBody* and resets to empty, $\emptyset$, the environment of the translation of each *MemberDecl* construct which occurs in the *ClassBody*, since no closure can have such constructs in its scope.
. *Primary:Py.* The rule 2 replaces the identifier with its value in τ, while rules 3 and 4 replace a closure with the function object obtained instantiating the interface `I$nk` on the types obtained translating each formal parameter type hence defining and translating the method `invoke` in the environment τ extended with the binding `(this/(s$elf,on))`, needed to guarantee the `this` transparency property.
. *ExtendedType:ET.* Rules 5 and 6 replace the closure type with the interface type `I$nk` instantiated on types obtained applying $\mathcal{F}[\![]\!]_\tau$ to the formal parameter types. Two different cases are considered, depending whether a return type or `void` is defined.
. *LocalVariableDeclarationStatement:L* Rule 7 considers the case in which a local variable is declared and initialized to a closure. In this case the closure is to be translated using a τ in which `this` is bound to the declared identifier to allow recursive invocations of the closure, case 2 above for τ. Rule 8 considers the declaration of a `shared` variable. Such declaration is replaced with the declaration of an object of type `C$shared` whose variable type is instantiated on the translated type of the declared variable. Rule 9 combine first and second case defining a `shared` closure.
. *BlockStatements:BB.* The extension of the environment τ to replace the name I of the `shared` variable with I.`value` is dealt with in rule 11.
. *MemberDecl:MB.* Rule 10 considers the case of a constructor or method declaration. In this case the translation must add a declaration for the local variable `s$elf` which must contain the reference to `this`, deal with `shared` formal parameters which are treated analogously to `shared` local variables adding for each `shared` parameter a local variable declaration whose type is `C$share` instantiated to the translated type parameter. The name of the new variable is obtained translating the parameter name through function $\mathcal{G}$. The rule also updates the environment τ in which the method body is translated, adding the binding for each formal parameter $I_i/\mathcal{G}(I_i)$.`value` for the `shared` parameters and the identity for the non `shared` ones. Eventually the binding (`this`/s$elf, off) is added if τ(`this`)$= \perp$, that is `this` is not bound in τ. In fact, if τ contains a binding for `this` such binding can only be `(s$elf,on)` which means that we are translating an `invoke` method of a function object representing a closure hence the binding for `this` is not to be modified.
. *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

[1] The proofs of the Lemmas and Theorems enunciated in the paper are in Appendix B

side is replaced by its binding in τ and the closure expression, analogously to local variable declaration, is translated using a τ in which I is replaced by `this`.

3.6. Closure Sub-typing

In [BO10] we define a minimal core calculus extending Featherweight Java [ABW01] with a form of closures which differ from the one adopted in JavaΩ only syntactically. In that paper we define a reduction semantics for the calculus and prove type safety. Other interesting properties, like sub-typing and the abstraction property will be investigated in future works. The final goal is to prove that the two semantics (reduction and translation semantics) commute so that the properties proved in the reduction semantics hold in the translation semantics and in the implementation obtained from it. About sub-typing, it is ease to note that both the intended semantics of Section 3.3 and the translation semantics defined by $\mathcal{F}[\![]\!]_\tau$ do not consider closure sub-typing. Hence, if `x` is a variable of closure type `T` then in the assignment `x = e`, expression `e` must be a closure of type `T`: more in general, any expression of closure type can be replaced only by an expression having the same exact closure type. However *contracovariance* (`TC` for short) is a good way to have closure sub-typing in order to weaken the rigidity previously exemplified. Rule Clos`TC` is for `TC` on closure types and is intended to extend

Closure Sub-typing Rules

$$\frac{S_1 \preceq T_1, ..., S_n \preceq T_n \qquad T \preceq S \qquad \overline{Q} \sqsubseteq \overline{P}}{\{(T_1, ..., T_n) : T \ \lceil \Uparrow \overline{Q} \rfloor\} \preceq \{(S_1, ..., S_n) : S \ \lceil \Uparrow \overline{P} \rfloor\}} \quad (\textsc{ClosTC})$$

$$\frac{\{(T_1, ..., T_n) : T \ \lceil \Uparrow \overline{Q} \rfloor\} \preceq \{(S_1, ..., S_n) : S \ \lceil \Uparrow \overline{P} \rfloor\}}{\amalg \texttt{I\$nk}\texttt{<}T_1, ..., T_n, T, \overline{Q\text{“}\texttt{u}\text{”}}\texttt{>} \ \lhd \ \amalg \ \texttt{I\$nh}\texttt{<}S_1, ..., S_n, S, \overline{P\text{“}\texttt{u}\text{”}}\texttt{>} \ \texttt{in PC}} \quad (\mathcal{F}\textsc{ClosSub})$$

Legenda: `Abbreviations and symbol conventions:`

- $\overline{P}$ `=` $P_1, ..., P_h$`,` $\overline{Q}$ `=` $Q_1, ..., Q_k$`,` $\overline{P\text{“}u\text{”}}$ `=` $P_1u, ..., P_hu$
- $\overline{Q\text{“}u\text{”}}$ `=` $Q_1u, ..., Q_ku$`, u = extends Throwable,` $\Uparrow$ `= throws`
- $\lhd$ `= extends, PC = Translated Program Classes`
- `judgements :` $\sqsubseteq =$ `is subtype, in =is in`

the inference rules of the Java sub-typing system. Moreover to have `TC` in the translation semantics we need to define $\mathcal{F}[\![]\!]_\tau$ to satisfy the rule $\mathcal{F}$ClosSub below. Rule $\mathcal{F}$ClosSub says that if the source program asserts that closure type $T_{sub} = \{(T_1, ..., T_n) : T \ \lceil \Uparrow \overline{Q} \rfloor\}$ is sub-type of the closure type $T_{super} = \{(S_1, ..., S_n) : S \ \lceil \Uparrow \overline{P} \rfloor\}$. Then the translated program must contain that the interface $\amalg \texttt{I\$nk}\texttt{<}T_1, ..., T_n, T, \overline{Q\text{“}\texttt{u}\text{”}}\texttt{>}$, associated, by $\mathcal{F}[\![]\!]_\tau$, to T_{sub}, is declared `extends` the interface $\amalg \texttt{I\$nh}\texttt{<}S_1, ..., S_n, S, \overline{P\text{“}\texttt{u}\text{”}}\texttt{>}$, associated, by $\mathcal{F}[\![]\!]_\tau$, to T_{sub}. Noting that Clos`TC` and $\mathcal{F}$ClosSub are inference rules but are not translation rules. Moreover, rule $\mathcal{F}$ClosSub requires the inference rule system defining the Java sub-typing relation $\preceq$ (see rule premises). This fact makes difficult to design translation rules that implement (i.e. translate code that satisfies) $\mathcal{F}$ClosSub. A way is modifies each translation rule so that: For each expression of a closure type T_{sub}, all the uses of the expression in the program are considered and the expected types are collected. Let $T_{super_1}, ..., T_{super_n}$ be the expected types. Then, all these types are closure types (or should be it) and are super-type of T_{sub}. So The interface associate to T_{sub} has the component `extends` declared consequently. However the problem is represented by the collection of the uses. The possible uses are: invocation, assignment, argument, return value. We have the following expected type:

— Invocation – the type itself;

— assignment – the type of the variable (or more in general, of the expression on the assignment left side). The selection of such a type is obtained by a contextual analysis of the assignment in which the expression occurs as assignment right side;
— argument – the type of the corresponding formal parameter of the method/closure that is invoked by the invocation in which the argument occurs. The selection of the invoked method requires the construction and use of the program symbol tables. This may results a cumbersome duplication of the symbol tables construction that a Java compiler front-end must produce;
— return value – the type of the type return declared by the method/closure in whose body the expression occurs. The selection of such a type is obtained by a contextual analysis of the method/closure declaration in which the expression occurs as return value.

4. HIGHER ORDER METHODS: MC_PARAMETERS.

The Java extension here defined is a revised version of the one described in [BO08a; BO09b]. 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.

4.1. Syntax and Semantics of MC_parameters

The main extensions are concerned with formal and actual parameters to include methods as arguments 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 A, while the syntactic categories directly affected are listed below.

ExtendedType::= *Type* | *FType*
FType::= `fun` *RootClass*`:(`[*ExtendedTypeList*]`)→(``void``|`*Type*`)`*ThrowOpt*
RootClass::= *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 argument, 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 arguments to HO methods and to closures (see the definition of *FormalParameter* in Appendix A, where *ExtendedType* is used instead of *Type*). In contrast to closures, as defined in Section 3, 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::= `abs` *MethodSpecifier* `from` *Type*
MethodSpecifier::=*Identifier* `(`*[ExtendedTypeList]*`)`→*(*`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 [GJSB05]) to invocations 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 [GJSB05]). Otherwise $\mathcal{M}(I, EL, T, TL, T_c)$ is the undefined $\perp$. The intended meaning of mc_parameters associates to:[2]

(i) Expression `abs` *I* `(`*EL*`)`→*T* *[*`throws` *TL]* `from` T_a, the partial function below:

$$\lambda f{:}C \to \lambda c{:}C \to \mathcal{M}(I,EL_{T_a}, T, TL_{T_c}, T_c) \textbf{ if } c \preceq f \preceq T_a \wedge \mathcal{M}(I,EL_{T_a}, T, TL_{T_a}, T_a) \neq \perp$$

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 [GJSB05]) in TL_{T_a}. 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 [GJSB05]). If $\mathcal{M}(I,EL_{T_a}, T, TL_{T_c}, T_c) \neq \mathcal{M}(I,EL_{T_a}, T, TL_{T_a}, T_a)$ then $\mathcal{M}(I,EL_{T_a}, T, TL_{T_c}, T_c)$ is an overriding of $\mathcal{M}(I,EL_{T_a}, T, TL_{T_a}, T_a)$. If $\mathcal{M}(I,EL_{T_a}, T, TL_{T_a}, T_a)$ does not exist then the program is not legal.

(ii) Parameter `fun` T_f`:(`EL_{T_f}`)`→T_{T_f} *[*`throws` TL_{T_f}*]* p, together with the argument *(*`abs` | `absClass`*)* *I* `(`*EL*`)`→*T* *[*`throws` *TL]* `from` T_a supplied for it, the partial function below:

$$\lambda c : C \to \mathcal{M}(I,EL_{T_a}, T, TL_{T_c}, T_c) \textbf{ if } c \preceq T_f \preceq T_a \wedge \mathcal{M}(I,EL_{T_a}, T, TL_{T_a}, T_a) \neq \perp$$

that, given a class c, subclass of T_f, selects method $\mathcal{M}(I,EL_{T_a}, T, TL_{T_c}, T_c)$, provided that both following conditions hold: *(a)* $T_f \preceq T_a$; *(b)* $\mathcal{M}(I,EL_{T_a}, T, TL_{T_a}, T_a)$ is a contracovariant [LW93; AC96; BO05]of the expected method, i.e. $T \preceq T_{T_f} \wedge TL_{T_a} \sqsubseteq TL_{T_f} \wedge EL_{T_a} \succeq EL_{T_f}$

(iii) Occurrence $e.p(e_1,...,e_n)$ inside the body of a HO method invoked with argument `abs` *I* `(`*EL*`)`→*T* *[*`throws` *TL]* `from` T_a supplied for parameter `fun` T_f`:(`EL_{T_f}`)`→T_{T_f} *[*`throws` TL_{T_f}*]* p, the invocation of method $\mathcal{M}(I,EL_{T_a}, T, TL_{T_c}, T_c)$ where T_c is the effective (i.e. run-time) 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 states 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 [GJSB05]);
— The class T_c is a subclass of T_f which must be a subclass of T_a;

[2] $\preceq$ (and $\succeq$) is the Java subtyping relation (see Section 4.10 [GJSB05]) possibly extended on lists of types in the obvious way, i.e. $t_1 \dots t_n \preceq s_1 \dots s_n$ means $t_1 \preceq s_1 \wedge \dots \wedge t_n \preceq s_n$. Eventually, $t_1 \dots t_n \sqsubseteq s_1 \dots s_k$ means $(\exists t_1 \preceq u_1, \dots, t_n \preceq u_n)\{u_1, \dots, u_n\} \subseteq \{s_1, \dots, s_k\}$, where $\subseteq$ is set inclusion

— The method effectively invoked in the invocation of an mc_parameter is a contra-covariant 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 formalization, using Java structures, of the intended meaning: It replaces code containing definitions and invocations of mc_parameters with code that contains definitions and invocations that yield the invocations of the methods that are effectively bound to such mc_parameters, according to the intended meaning. Moreover, it is based on the computational structures of the callback methodology [GHJV05].

4.2. 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 [Mey04; Hor07] 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 summarized 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 arguments of HO methods. In fact, we need a family of interfaces `ApplyClass$nk`, where the suffices `n` and `k` stand for the number of arguments and for the number of different types of throwable exceptions of the wrapped methods for which the interface is defined. Each interface has only one method `Apply` whose first parameter is an object, namely the object on which the wrapped method must be invoked, while the following n parameters are the n parameters to which the wrapped method applies. Eventually, `Apply` throws the same k different classes of exceptions of the wrapped method. Interface `ApplyClass$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, ET_1, ..., ET_n, T,$
QI_1 `extends Throwable,...,`QI_k `extends Throwable>{`
`public` T `apply(`$RTo, ET_1\, x_1, ..., ET_n\, x_n$`) throws` $QI_1, ..., QI_k$`;}`

. `public interface ApplyClassV$nk<`$RT, ET_1, ..., ET_n,$
QI_1 `extends Throwable,...,`QI_k `extends Throwable>{`
`public void apply(`$RTo, ET_1\, x_1, ..., ET_n\, x_n$`) throws` $QI_1, ..., QI_k$`;}`

(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<`$Ta_1, ..., Ta_p$`>` (where $Ta_1, ..., Ta_p$ are the type arguments supplied to the generic class `C<`$Q_1, ..., Q_p$`>`, of the program, with type parameters $Q_1, ..., Q_p$) having methods named `m`$_1$,...`m`$_q$, which are to be passed as parameters in the program. Then q classes are defined. In particular, for each, possibly generic, `m`$_i$, let $ET_{ai_1}, ..., ET_{ai_{n_i}}$ be the types of the arguments to

which the method applies in the invocation, once passed as parameter, let T_i be the invocation return type, $T_{ti_1}, ..., T_{ti_{k_i}}$ be the list of exception classes that invocation can throws. Then, a class $\mathcal{I}[\mathtt{m}_i]$ is defined[3] as follows:

```
static class I[m_i] implements ApplyClass$n_ik_i<C<Ta_1, ..., Ta_p>,
              ET_ai_1, ..., ET_ai_n_i, T_i, T_ti_1, ..., T_ti_k_i > {
    public T_i Apply(C<Ta_1, ..., Ta_p> o, ET_ai_1 x_1, ..., ET_ai_n_i x_n_i)
                                  throws T_ti_1, ..., T_ti_k_i {
        return (o.m_i(x_1,...,x_n_i);}}
```

(3) **HO Methods.** Every HO method $\mathtt{I_{ho}}$ is defined having one `ApplyClass$nk` parameter `o` for each object wrapping one of the method $\mathtt{m}_i$ which will be bound to it during an invocation of the HO method. This is expressed as below

```
public ET_I_ho I_ho (...ApplyClass$nk<ET_E, ET_1, ..., ET_n, T, T_1, ..., T_k> o...)
      {... o.Apply(E, E_1, .., E_n) ...}
```

where E is the expression of reference type (i.e. computes an object) on which $\mathtt{m}_i$ must be invoked, and E_j ($j \in [1, n_i]$) are the arguments of $\mathtt{m}_i$. If E is a qualified identifier naming a class then we cannot use interfaces defined in this way (see Appendix C).

(4) **Creation of Method Objects.** The invocation of $\mathtt{I_{ho}}$ requires the construction of one object of the class $\mathcal{I}[\mathtt{m}_i]$. This is accomplished in the invocation below, under the assumption, in point 2, that class $\mathcal{I}[\mathtt{m}_i]$ 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. If the passed methods are variable arity methods we can simply use the interfaces as defined in Table 1 of Appendix C in which thr method `apply` contains one more, optional, parameter *[,ET...y]* for methods with variable arity parameters of type *ET*. The Java static invocation mode (see Section 12.4 in [GJSB05]) for class methods bound to mc_parameters, is considered, in Point 3, by resorting to the use of a default object of the class. This is further discussed in Appendix C. 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 [BO09b]. In particular, the use of anonymous classes avoids the need of introducing names for the classes (as for $\mathcal{I}[\mathtt{m}_i]$ in point 2 of the methodology.). Hence, unlike the one presented in [BO09b], the translation presented in next section uses anonymous classes and combines points 2 and 4.

4.3. $\mathcal{E}[\![]\!]_\rho$: A Callback based, Translation Semantics of MC_ parameters

The translation semantics is defined using the syntax directed rule system described in Section 2. Hence, it has at least one rule for each production of the unambiguous grammar of Appendix A, 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 $\mathcal{E}[\![]\!]_\rho$ contains only 8 significant rules and the *otherwise* metarule of Section 2. 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

[3] $\mathcal{I}[\mathtt{m}_i]$ is a class identifier which must be unique for each different use of method $\mathtt{m}_i$ as argument in the HO methods of the program. Then, the specific name chosen for $\mathcal{I}[\mathtt{m}_i]$ depends from the method identifier, the list of types used in the invocation and the class in which $\mathtt{m}_i$ is defined, and the technique [BO09b] used for the definition of the class: inner, anonymous or stand alone.

in the translation $\mathcal{E}[\![]\!]_\rho$. 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 $\emptyset$ and does not contain bindings. Any other environment is obtained by $\mathcal{R}[\![T\ I]\!]_\rho$ 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 Fig. 2).

. *CompilationUnit:CU*. Rule 1 applies to constructs of the category *CompilationUnit* and resets to empty, $\emptyset$, 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 `ApplyClass$nk` and `ApplyClassV$nk` used to model method objects as described in Section 4.2, 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 4.2 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 `abs` $I\,EL{\rightarrow}T\,[$`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 [GJSB05], 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 PS maps reference type expressions into Java reference type expressions and primitive type (i.e. non reference type) expressions into primitive type expressions. 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 contracovariance of the wrapped method (see Theorem4.1).

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. Moreover, the existence and access of the methods passed using mc_parameters in the source program, can be checked, at compile time, checking for the existence and access of the wrapped methods in the translated program. Again, the methods that are passed by mc_parameters are checked to be

contracovariant with the expected methods: Hence their invocation, in the body of an HO method, is type safe. Eventually, the execution of the translations of Java programs that use mc_parameters and HO methods never goes in unexpected, unrecoverable, computation states. This furnishes an indirect proof of type safety of Java extended with mc_parameters and HO methods. The proof is indirect since it is not obtained resorting to the type system of Java extended with the new types (for mc_parameters and HO methods) and proving that well typed programs of the extended language compute values and never get stuck. Instead, type safety of the extended programs is reduced to the type safety, in standard Java, of the corresponding translated programs. Eventually, Corollary 4.5 asserts that the translation semantics $\mathcal{E}[\![]\!]_\rho$ does not allow static invocation mode for mc_parameters: It means that even if the method that is bound to a mc_parameter is a class method, the invocation of the mc_parameter must contain a target reference (as it happens for Java virtual invocation mode, Section 15.12.3 in [GJSB05]).

THEOREM 4.1 (WRAPPED METHODS FOR MC_PARAMETERS). *Let* $E \equiv$`abs` I $EL_s \rightarrow T$ *[*`throws` TL*]* `from` T_a *in the scope of an environment* ρ*, and* $EL \equiv \mathcal{E}[\![EL_s]\!]_\rho$*. Let* M *be the wrapped method of the object, if any, created executing* $\mathcal{E}[\![E]\!]_\rho$*. Then* M *is (possibly an overriding of)* $\mathcal{M}(I, EL_{T_a}, T, TL_{T_a}, T_a)$*, 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 4.2 (EXISTENCE AND ACCESS OF MC_PARAMETERS). *Let* $E \equiv$`abs` I $EL_s \rightarrow T$ *[*`throws` TL*]* `from` T_a *in the scope of an environment* ρ*, and* $EL \equiv \mathcal{E}[\![EL_s]\!]_\rho$*. Let* $\mathcal{M}(I, EL_{T_a}, T, TL_c, T_c)$ *be the method that results, for any given* $T_c \preceq T_f \preceq T_a$*, from the intended semantics of* E*. Then* $\mathcal{M}(I, EL_{T_a}, T, TL_c, T_c)$ (i) *exists if and only if* $\mathcal{E}[\![E]\!]_\rho$ *has no compile-time type errors;* (ii) *can be invoked only if the* $\mathcal{E}[\![E]\!]_\rho$ *has no method access violations.* □

LEMMA 4.3 (MC_PARAMETERS ARE CONTRACOVARIANT). *Let* p *be a mc_parameter of a HO method* I_{ho} *(in a source program) and* `fun` RT`:` $EL \rightarrow T$ *[*`throws` TL*]* *be its type with* $EL \equiv ET_1, \ldots, ET_n$*. Let* o *be a binding for* p *in an invocation of* I_{ho} *in the translated program obtained by* $\mathcal{E}[\![]\!]_\perp$ *and* $e.p(e_1, ..., e_n)$ *any invocation of* p *occurring in the body of* I_{ho}*, and* e *be a reference type expression. Then,* o *is wrapping a method which is* contracovariant *with* $EL \rightarrow T$ *(i.e. has return type which is a subtype of* T *and type signature* $ET_{o_1}, \ldots, ET_{o_n}$ *such that* ET_{o_i} *is a supertype of* ET_i *(*$\forall i \in [1..n]$*)), if and only if the translation of* $e.p(e_1, ..., e_n)$ *is a Java well typed term.* □

THEOREM 4.4 (TYPE SAFETY). *Let* P *be any Java program extended with mc_parameters. Then* P *is type safe only if its translation* $\mathcal{E}[\![P]\!]_\perp$ *is a Java type safe program, i.e.* (i) *it is well-typed and,* (ii) *its execution never gets stuck.* □

COROLLARY 4.5 (STATIC MODE INVOCATION). *Static mode invocation of mc_parameters must have a target reference: Each invocation* $e.p(e_1, ..., e_n)$*, where* p *is a mc_parameter, has* e *computing an object (i.e.* $\mathcal{E}[\![e]\!]_\rho$ *is a reference type expression) even if* p *can be bound to a class method (namely, a method that is invoked in static mode).* □

5. EXAMPLES

5.1. HO Programming with a Class of Geometric Shapes

A first example is a classical problem for higher order programming already discussed in [BO04], but here modified to use generics an classes in Java APIs such as `LinkedList` and `ListIterator`. The example defines an abstract class `Shape` for geometric shapes

$$\mathcal{E}[\![CU]\!]_\rho = [\![At]\!]\ \texttt{package}\ QI;]\ IM\ \mathcal{E}[\![CD_1]\!]_\emptyset \ldots \mathcal{E}[\![CD_n]\!]_\emptyset \quad \text{with } CU = [\![At]\!]\ \texttt{package}\ QI;]\ IM\ CD_1 \ldots CD_n$$

$$\mathcal{E}[\![MB]\!]_\rho = D\ [Q]\ [TV]\ I\ \mathcal{E}[\![FP]\!]_\rho\ O_0\ TR\ \mathcal{E}[\![B]\!]_{\rho\uparrow FP} \text{ with } MB \in MethodOrConstructorDecl$$

$$\mathcal{E}[\![ET]\!]_\rho = \begin{cases} \texttt{ApplyClass\$nk} <\mathcal{E}[\![RT]\!]_\rho[,\mathcal{E}[\![ET_1]\!]_\rho,..,\mathcal{E}[\![ET_n]\!]_\rho],\mathcal{E}[\![T_o]\!]_\rho[,\mathcal{E}[\![QI_1]\!]_\rho,..,\mathcal{E}[\![QI_k]\!]_\rho]> \\ \quad \text{with } ET = \texttt{fun}\ RT\colon ([ET_1,\ldots,ET_n]) \to T_o\ [\texttt{throws}\ QI_1,\ldots,QI_k] \\ \texttt{ApplyClassV\$nk} <\mathcal{E}[\![RT]\!]_\rho[,\mathcal{E}[\![ET_1]\!]_\rho,..,\mathcal{E}[\![ET_n]\!]_\rho,\mathcal{E}[\![QI_1]\!]_\rho,..,\mathcal{E}[\![QI_k]\!]_\rho]> \\ \quad \text{with } ET = \texttt{fun}\ RT\colon ([ET_1,\ldots,ET_n]) \to \texttt{void}\ [\texttt{throws}\ QI_1,\ldots,QI_k] \end{cases}$$

$$\mathcal{E}[\![E]\!]_\rho = \begin{cases} \texttt{new ApplyClass\$nk<}TL_r\texttt{>()\{} \quad \text{with } E = \texttt{abs}\ I([EL]) \to T\ [\texttt{throws}\ TL]\ \texttt{from}\ T_c \\ \ \texttt{public}\ T\ \texttt{Apply}\ FP_r[\texttt{throws}\ TL]\texttt{\{} \qquad \wedge EL = ET_1,\ldots,ET_n \\ \ \ \texttt{return}\ o.I(x_1,\ldots,x_n);\texttt{\}\}} \qquad \wedge TL = QI_1,\ldots,QI_k \\ where \\ \quad TL_r = T_c,\mathcal{E}[\![ET_1]\!],\ldots,\mathcal{E}[\![ET_n]\!],T,QI_1,\ldots,QI_k \\ \quad FP_r = (T_c\ o,\mathcal{E}[\![ET_1]\!]\ x_1,\ldots,\mathcal{E}[\![ET_n]\!]\ x_n) \\ \texttt{new ApplyClassV\$nk<}TL_r\texttt{>()\{} \quad \text{with} E = \texttt{abs}\ I([EL]) \to \texttt{void}\ [\texttt{throws}\ TL]\ \texttt{from}\ T_c \\ \ \texttt{public void Apply}\ FP_r[\texttt{throws}\ TL]\texttt{\{} \qquad \wedge EL = ET_1,\ldots,ET_n \\ \ \ o.I(x_1,\ldots,x_n)\texttt{\}\}} \qquad \wedge TL = QI_1,\ldots,QI_k \\ where \\ \quad TL_r = T_c,\mathcal{E}[\![ET_1]\!],\ldots,\mathcal{E}[\![ET_n]\!],QI_1,\ldots,QI_k \\ \quad FP_r = (T_c\ o,\mathcal{E}[\![ET_1]\!]\ x_1,\ldots,\mathcal{E}[\![ET_n]\!]\ x_n) \end{cases}$$

$$\mathcal{E}[\![PS]\!]_\rho = \begin{cases} (T)(I.\texttt{Apply}(\mathcal{E}[\![PS_1]\!]_\rho,(ET_1)\mathcal{E}[\![E_1]\!]_\rho,\ldots,(ET_n)\mathcal{E}[\![E_n]\!]_\rho)) \\ \quad \text{with } PS = PS_1\ .I\ ([E_1,\ldots,E_n]) \wedge \rho(I) = \texttt{fun}\ RC\colon([ET_1,\ldots,ET_n]) \to T\ TR \\ I.\texttt{Apply}(\mathcal{E}[\![PS_1]\!]_\rho,(ET_1)\mathcal{E}[\![E_1]\!]_\rho,\ldots,(ET_n)\mathcal{E}[\![E_n]\!]_\rho) \\ \quad \text{with } PS = PS_1\ .I\ ([E_1,\ldots,E_n]) \wedge \rho(I) = \texttt{fun}\ RC\colon([ET_1,\ldots,ET_n]) \to \texttt{void}\ TR \end{cases}$$

where: $\mathcal{R}[\![T\ I]\!]_\rho(x) = T$ if $I = x \wedge T \in Ftype$ $\quad \mathcal{R}[\![T\ I]\!]_\rho(x) = \bot$ if $I = x \wedge T \notin Ftype$
$\mathcal{R}[\![T\ I]\!]_\rho(x) = \rho(x)$ if $I \neq x$ $\quad \emptyset(x) = \bot$

$$\rho \uparrow () = \rho$$
$$\rho \uparrow (P_1 P_2 \ldots P_n) = \mathcal{R}[\![ET_1\ I_1]\!]_{\rho\uparrow(P_2\ldots P_n)} \quad \text{with } P_i = [\texttt{final}]\ At_i\ ET_i\ I_i([\,])^*$$

Legenda: For arbitrary index p, A$\in$ *Arguments*, AR$\in$ *ArrayCreatorRest*, At$\in$ *Annotations*, B, B$_p\in$*Block*, BB, BB$_p \in$ *BlockStatements*, BS, BS$_p \in$ *BlockStatement*, BT$\in$ *BasicType*, CB$\in$ *ClassBody*, CD, CD$_p \in$ *ClassOrInterfaceDeclaration*, CN$\in$ *CreatedName*, CR$\in$ *ClassCreatorRest*, CU$\in$ *CompilationUnit*, D, D$_p \in$ *ModifierOpt*, E, E$_p \in$ *Expression*, EL, EL$_p \in$ *ExtendedTypeList*, ET, ET$_p \in$ *ExtendedType*, F, F$_p\in$*ITs*, FP, FP$_q \in$ *FParameters*, FS, FS$_p \in$ [`final` | `shared`], GS$\in$ *ExplicitGenericInvocationSuffix*, x, x$_p$, I, I$_p\in$*Identifier*, IM$\in$ *ImportDeclaration*, IO$\in$ *InfixOp*, K$\in$ *Literal*, L$\in$ *LocalVariableDeclarationStatement*, MB, MB$_p \in$ *MemberDecl*, NI$\in$ *NonInvocationSelector*, O$_i \in []^*$, P, P$_p \in$ *FormalParameter*, PO$\in$ *PostfixOp*, PS, PS$_q \in$ *PrimarySelector*, Py$\in$ *Primary*, Q, Q$_p\in$*TPs*, QI$\in$ *QualifiedIdentifier*, RT$\in$*RootClass*, S, S$_p \in$ *Statement*, Sr$_p \in$ *Selector*, T, T$_p\in$*Type*, Ta$_p$, $\tilde{T}a_p \in$ *TypeArgument*, TA$\in$ *ParsOpt*, TL, TL$_p \in$ *TypeList*, TR$\in$ *ThrowOpt*, TV, TV$_p \in$ [*Type* | `void`], U, U$_p \in$ *ST*, V, V$_p \in$ *VariableDeclaratorId*,

Fig. 2. MC_parameters: $\mathcal{E}[\![]\!]_\rho$ 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 Rectangle(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); }}
```

Fig. 3. a. HO Programming with a Class of Geometric Shapes: `P`

```
// 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);}
```

Fig. 3. **b.** HO Programming: The Translated Program – $\mathcal{E}[\![\texttt{P}]\!]_\rho$

and several concrete classes `Rectangle, Circle, Triangle` etc .and an extension of `LinkedList` in which a HO method `map` is defined. The HO method is used to compute the list of areas of all the shape contained in a list, even though the the method to compute the area is a different one according to the type of the element in the listth that is being processed.

```
public 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;}}
```

Fig. 4. a. Class `Memoize`: A Method that Memoizes Closures

```
public 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;}}
```

Fig. 4. b. Class `Memoize`: The Translated Program – $\mathcal{F}[\![\texttt{Memoize}]\!]_\tau$

```
  shared {(Integer):Integer} Yfib;
  {(Integer):Integer}fib = {(Integer x):Integer ⇒ if (x==0 || x==1) return 1;
                          else Yfib.invoke(x-1)+ Yfib.invoke(x-2)}
  Yfib = new Memoize<Integer,Integer>().Memoized(fib);
  // Yfib = fib; without memoization
```

Fig. 4. **c.** Class `Memoize`: Memoizing Fibonacci Numbers

5.2. Writing a Method that Maps Closures into Memozed Closures

A second example, taken from [Goe07], defines, in Fig.4.a, a method `memoized` that maps closures of type $\{(T_1) : T\}$ into *memoized* closures of the same type and same *external* behavior but of different performance since a memoized closure avoids repeating computation for previously processed inputs. To obtain this, `memoized` has a closure parameter *f*, declared `final`, of type $\{(T_1) : T\}$, bound to the closure to be memoized, and a `shared` variable *table* initialized to an empty hash table. `memoized` returns a closure that behaves as follows. It has a parameter x of type T_1 and asks table *table* for a key equals to the value of x. If the key is found then it returns the value bound in *table* to x, otherwise it behaves like *f*, invoking f.`invoke`(x), updates *table*, adding the key/value pair x/f.`invoke`(x), returns the value f.`invoke`(x). Note that, in this way, each hash table *table* is a private resource of the memoized closure since each time `memoized` returns the closure and ends, thus making *table* no more accessible outside the returned closure. 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 Fig. 4.c, whose translation and execution are left to the reader:

6. PUTTING TRANSLATIONS TOGETHER

We have introduced and discussed, in a separate way, the extension of Java 1.5 with mc_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 mc_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 6.1 (COMPLETENESS $\mathcal{E}[\![]\!]_\emptyset$). *Translation $\mathcal{E}[\![]\!]_\emptyset$ is complete, i.e. it maps each program of Java extended with closures into an equivalent program of ordinary Java 1.5.* □

THEOREM 6.2 (COMPLETENESS $\mathcal{F}[\![]\!]_\emptyset$). *Translation $\mathcal{F}[\![]\!]_\emptyset$ 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 6.3 (ORTHOGONALITY). *The composition of the two translations is commutative, i.e.: $\mathcal{F}[\![\mathcal{E}[\![P]\!]_\emptyset]\!]_\emptyset = \mathcal{E}[\![\mathcal{F}[\![P]\!]_\emptyset]\!]_\emptyset$.* □

The language, obtained integrating the two extensions, allows to define programs that contain: closures having also mc_parameters, and vice versa, HO methods having closures as parameters, or computing 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 $\mathcal{F}[\![]\!]_\tau$ and through the rule system $\mathcal{E}[\![]\!]_\rho$ 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 $\mathcal{F}[\![]\!]_\tau$ and/or that of $\mathcal{E}[\![]\!]_\rho$, depending on if $\mathcal{F}[\![]\!]_\tau$ and/or $\mathcal{E}[\![]\!]_\rho$ is defined for such a production. Then, we see that, in correspondence to a production p of the form $C_0 ::= C_1 \dots C_n$, four cases arise. Let $\mathcal{E}F[\![]\!]_{\rho,\tau}$ be the translation semantics of Java extended with HO methods, mc_parameters and closures. Then, we define $\mathcal{E}F[\![]\!]_{\rho,\tau}$ examining, case by case, how to compose translations $\mathcal{E}[\![]\!]_\rho$ and $\mathcal{F}[\![]\!]_\tau$ in each of the four cases.

Case1. Only one, between $\mathcal{F}[\![]\!]_\tau$ and $\mathcal{E}[\![]\!]_\rho$, is defined on p. This is the case for rules 3, 4, 5, 6, 7, 8 of $\mathcal{E}[\![]\!]_\rho$, and rules 2, 3, 4, 5, 6, 7, 8, 9, 11, 12 of $\mathcal{F}[\![]\!]_\tau$. All these rules become rules of $\mathcal{E}F[\![]\!]_{\rho,\tau}$ when each occurrence of $\mathcal{E}[\![U]\!]_\rho$ and $\mathcal{F}[\![U]\!]_\tau$ are replaced by $\mathcal{E}F[\![U]\!]_{\rho,\tau}$ for any argument U.

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

Case 3. $\mathcal{F}[\![A_0]\!]_\tau = f(A_1, \mathcal{F}[\![A_1]\!]_\tau), \dots, f(A_n, \mathcal{F}[\![A_n]\!]_\tau)$ `with` $b_i(A_0, A_1, \dots, A_n)$
$\mathcal{E}[\![A_0]\!]_\rho = f(A_1, \mathcal{E}[\![A_1]\!]_\rho), \dots, f(A_n, \mathcal{E}[\![A_n]\!]_\rho)$ `with` $b_i(A_0, A_1, \dots, A_n)$
This is the case of the *otherwise* metarule of the two systems which is replaced by the *otherwise* metarule of $\mathcal{E}F[\![U]\!]_{\sigma,\tau}$.

Case 4. $\mathcal{F}[\![A_0]\!]_\tau = r_i(A_1, \mathcal{F}[\![A_1]\!]_\tau, \dots, A_n, \mathcal{F}[\![A_n]\!]_\tau$ `with` $b_i(A_0, A_1, \dots, A_n)$
$\mathcal{E}[\![A_0]\!]_\tau = s_i(A_1, \mathcal{E}[\![A_1]\!]_\tau, \dots, A_n, \mathcal{E}[\![A_n]\!]_\tau$ `with` $b_i(A_0, A_1, \dots, A_n)$

(Symmetric case: Changing $\mathcal{F}[\![]\!]_\tau$ with $\mathcal{E}[\![]\!]_\rho$) Differently than we would expect, since Theorem 6.3 on orthogonality, this case contains two rules: Rule 2 of $\mathcal{E}[\![]\!]_\rho$ and rule 10 of system $\mathcal{F}[\![]\!]_\tau$. 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, $\mathcal{E}F[\![]\!]_{\rho,\tau}$ contains only one rule. The rule is rule 10 of $\mathcal{F}[\![]\!]_\tau$ where: *(i)* $\mathcal{F}[\![MB]\!]_\tau$ (on the left side) is replaced by $\mathcal{E}F[\![MB]\!]_{\rho,\tau}$; *(ii)* $[\mathcal{F}[\![TV]\!]_\tau]$ is replaced by $[\mathcal{E}F[\![TV]\!]_{\rho,\tau}]$; *(iii)* $\mathcal{F}[\![BB]\!]_{\overline{\tau}}$ is replaced by $\mathcal{E}F[\![BB]\!]_{\rho\uparrow FP,\overline{\tau}}$

THEOREM 6.4 (COMPLETENESS). *Translation $\mathcal{E}F[\![]\!]_{\emptyset,\emptyset}$ is complete: It maps each program of the extended language onto an equivalent program of ordinary Java 1.5.* □

7. CONCLUSIONS

In this work (chapter) we have described two HO mechanisms: mc_parameters and closures, to extend Java, defining a translation semantics and proving some fundamental properties including completeness and orthogonality. The implementation of the extended language, along the lines already experimented in JavaΩ [BO07a; BO08b], is immediate: The translation semantics $\mathcal{E}F[\![]\!]_{\rho,\tau}$, can be formally converted into a source-to-souce translation and implemented as a one-pass preprocessor [ALSU07] that is developed using Lex & Yacc [LMB95] and GNU Bison [CS06]. This implementation allows to quickly develop a prototype that can be used to test the programming features of the extended language and runs in combination with any Java compiler, including *Javac* of SUN, *GCJ* of GNU, *ECJ* of Eclipse. Moreover, for competitive compilations of the extended language, 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 $\mathcal{E}F[\![u]\!]_{\rho,\tau}$ in correspondence of the parsing of any language structure u, for suitable contexts (producing the environments) ρ and τ. We are currently investigating the possibility to apply software engineering techniques and Java annotations to support error localization in non-native constructs. The aim is that errors in a non-native construct of a program, for instance a closure type, found by the Java compiler during the analysis of the code, obtained by program preprocessing, are recognized as errors of the construct and localized (for possibly error recovery) on the source program, i.e. the program before preprocessing. Moreover, as mentioned in section 3, we are still investigating the definition of core Java sub-languages with reduction semantics to obtain different modelizations of a same construct, for instance closures, to experiment the adequacy to the initial aims and to prove properties of each modelization and eventually prove that all such properties are preserved in the full language, extended with the new construct, when the reduction semantics of the sub-language and the translation semantics of the extended language commute.

A. APPENDIX - AN UNAMBIGUOUS GRAMMAR FOR JAVA

The syntactic categories that are used, but not defined, are underlined and are those in chapter 18 of the Java JSL[GJSB05]. We use the same syntactic EBNF conventions of JSL 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) but we use * for the iteration operator (a^* means zero, one or more occurrences of a). Eventually, a^+ is an abbreviation for aa^*.

CompilationUnit ::= *[[Annotations]* `package` *QualifiedIdentifier* `;]` *ImportDeclaration** *(ClassOrInterfaceDeclaration* | `;`*)*

ClassDeclaration ::= `public class` *Identifier [TPs] [ST] [ITs] ClassBody*

TPs::=`<`*TypeParameter* (`,` *TypeParameter*)*`>`

ST::=`extends` *Type*

ITs::=`implements` *TypeList*
ClassBody::=`{` *(MemberDecl)** `}`
MemberDecl::= `;`
|*[*`static`*]* *Block*
|*ModsOpt FieldDeclarator*
|*MethodOrConstructorDecl*
|*ClassOrInterfaceDeclaration*
FieldDeclarator::= *Type Identifier VariableDeclaratorRest*
MethodOrConstructorDecl::=*ModsOpt [TPs] [(*`void`|*Type)] Identifier FParameters* `[]`*
ThrowOpt (Block | `;`*)*
ModsOpt::=*Modifier**
ThrowOpt::= *[*`throws` *QualifiedIdentifier*+*]*
ExtendedType::= *Type* | *FType*
Type::= *ParameterizedType* | *BasicType* | *ClosType*
ExtendedTypeList::=*ExtendedType (*`,` *ExtendedType)**
FParameters::= (*[*`final` | `shared`*]* *[Annotations] ExtendedType FormalParDeclsRest*)
FormalParDeclsRest::= *VariableDeclaratorId [*`,` *FParameters]*
| `...` *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*
|*((Expression*|*Type)) Expression3*
|*PrimarySelector PostfixOp**
PrimarySelector::= *PrimarySelector NonInvocationSelector*
|*[PrimarySelector* `.`*] [*<*TypeList*>*] Identifier Arguments*
|*Primary*
Primary::=*ParExpression*
|<*TypeList*> *(ExplicitGenericInvocationSuffix* | `this` *Arguments)*
|*[QualifiedIdentifier* `.`*]* `this` *[Arguments]*
|*[QualifiedIdentifier* `.`*]* `super` *(Arguments* | `.` *Identifier [Arguments])*
|*Literal*
|*[QualifiedIdentifier* `.`*]* `new` *[*<*TypeList*>*] CreatedName CreatorRest*
|*QualifiedIdentifier [*`[]`*`.class` | *[Expression]]*
|*BasicType* `[]`*`.class`
|`void.class`
|*Closure*
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]*

B. APPENDIX - PROOFS OF LEMMAS AND THEOREMS

PROOF. Theorem 3.3. Entering a method, τ is extended with a binding of the fictitious identifier `this` with the pair (`s$elf`,`off`) where `s$elf` is the name of a fresh variable introduced in the assignment above, and `off` is a flag specifying that the binding for `this` is not active. In fact, the binding is set active when $\mathcal{F}[\![]\!]_\tau$ traverses a closure updating τ to set the flag to `on`, thus making the binding for `this` active when $\mathcal{F}[\![]\!]_\tau$ traverses a closure updating τ to set the flag to `on` thus making the the binding for `this` active in the third and fourth rule. □

PROOF. Theorem 4.1. Assume that execution of $\mathcal{E}[\![E]\!]_\rho$ creates an object. By rule 5 (6) of $\mathcal{E}[\![]\!]_\rho$, the created object wraps a method that can be invoked by expression $o.I(x_1,...,x_n)$. Hence, the method must be invoked on objects of the class of o, namely (any subclass of) T_a, has the type signature that results from the overloading solution for a method named I, in the class T_a, of the type list of $x_1,...,x_n$. This type list is EL: Let EL_{T_a} be its overloading solution (possibly, $EL_{T_a} \equiv EL$). Again, the wrapped method must compute a value of type T, since the expression is the argument of a return statement. Eventually, the method may throw exceptions of types included in the types of the list TL_a, since such a return statement is the body of an `Apply` method. Hence, the wrapped method is a possibly overriding method of a class $T_c \preceq T_a$, has name I, arguments type list EL_{T_a} and throwable types TL_{T_a}, and return type T. This completes the proof. □

PROOF. Lemma 4.2. *(i)*-if part. In this case $\mathcal{E}[\![E]\!]_\rho$ has no type errors. Hence, $\mathcal{E}[\![E]\!]_\rho$ execution creates an object on which Theorem 4.1 holds for a wrapped method. Such a method is, possibly an overriding of, $\mathcal{M}(I,EL_{T_a},T,TL_a,T_a)$ for any class T_c such that $T_c \preceq T_f \preceq T_a$, namely $\mathcal{M}(I,EL_{T_a},T,TL_c,T_c)$.
(i)-only if part. Assume that method $\mathcal{M}(I,EL_{T_a},T,TL_c,T_c)$ exists. Then, T_c, T_a, EL_{T_a}, T and TL_c are correct types.
(ii) Assume that method $\mathcal{M}(I,EL_{T_a},T,TL_c,T_c)$ exists for $T_c \preceq T_f \preceq T_a$. Then, the method can be invoked where E occurs in the program. This completes the proof. □

PROOF. Lemma 4.3. By the rule 7 (8) of $\mathcal{E}[\![]\!]_\rho$ any invocation $e.p(e_1,...,e_n)$ of a mc_parameter, p, of the source program is replaced, in the translated program, with an invocation of the `Apply` method on the object bound to p: $(T)(p.\texttt{Apply}(\mathcal{E}[\![e]\!]_\rho,(ET_1)\mathcal{E}[\![e_1]\!]_\rho,...,(ET_n)\mathcal{E}[\![e_n]\!]_\rho))$. This object must be a method object of type `ApplyClass$nk` and is inserted, in the translated program, by rule 5 (6). Moreover, according to such a rule and by definition of the Interface `ApplyClass$nk`, this object is wrapping a method m. Let T_o be the return type of m and $ET_{o_1},\ldots,ET_{o_n}$ be the argument type list with which m can be invoked. Then, when `Apply` invocation applies to arguments $(u,e_1,\ldots,e_n)$, m (is invoked on the object u and) applies to arguments $(e_1,\ldots,e_n)$, i.e. $u.m(e_1,...,e_n)$ is executed.

- *If part.* Since `Apply` invocation is cast to T, m must compute a value of a subtype of T. Otherwise, a compile-time type error occurs in the translated program and the invocation is not a Java well typed term. This completes the proof as far as covariance of the return type. Moreover, since m applies to $(e_1,...,e_n)$, it has type signature $ET_{o_1},...,ET_{o_n}$ such that ET_{o_i} is a supertype of the type of e_i. But each argument e_i is cast to ET_i, hence for each i, ET_{o_i} is a supertype of ET_i. This completes the proof as far as contravariance and of the *if part*.

- *Only If part.* If m is contracovariant with $ET_1,...,ET_n \rightarrow T$, for all method objects that are bound to p then invocation $(T)(p.\texttt{Apply}(\mathcal{E}[\![e]\!]_\rho,(ET_1)\mathcal{E}[\![e_1]\!]_\rho,...,(ET_n)\mathcal{E}[\![e_n]\!]_\rho))$ is a Java well typed term since each m results Java *potentially applicable* and satisfies compile-time step 2 of Java method invocation (see Section 15.12.2 [GJSB05]) □

PROOF. Theorem 4.4. *(i)* By definition when we assume that each program in Java extended with mc-parameters is well typed if and only if its $\mathcal{E}[\![]\!]_\perp$ translation is a Java well typed program. *(ii)* Immediate from Lemma 4.2 and Lemma 4.3 which guarantee that the invocation of a mc_parameter computes the invocation of a method that exists, can be accessed, and is applicable. □

PROOF. Corollary 4.5. Let p be a formal mc_parameter and $e \equiv e_0.p(e_1, ..., e_n)$ be an invocation of p. Let $e_r = \mathcal{E}[\![e_0]\!]_\rho$ be the translation of e_0 for a right ρ. If e_r is a non-reference type expression then in $\mathcal{E}[\![e]\!]_\rho$, invocation of method `Apply` contains a type error on its first argument since a type reference is expected. Hence, by Theorem 4.4, e cannot be typed. □

PROOF. Theorem 6.1. It is enough proving that $\mathcal{E}[\![]\!]_\emptyset$ is idempotent, i.e. $\mathcal{E}[\![\mathcal{E}[\![P]\!]_\emptyset]\!]_\emptyset = \mathcal{E}[\![P]\!]_\emptyset$ for each program P (namely, compilation unit, in Java). None of $\mathcal{E}[\![]\!]_\rho$ rules inserts types `fun` or constructs `abs` in the translated program. Instead, rules 3 and 4 remove types `fun` from each extended type ET and each of the types occurring inside ET. Similarly, rules 5 and 6 act on the construct `abs`. Hence, if $\mathcal{E}[\![P]\!]_\emptyset$ contains an occurrence of either `fun` or `abs` then such an occurrence was already in P. But it cannot be since rules 3, 4 or 5, 6 would have removed the occurrence. □

PROOF. Theorem 6.2. It can be given following the arguments of the proof of Theorem 6.1 reformulated on $\mathcal{F}[\![]\!]_\emptyset$ about the removal of `shared`, closure type and the closure construct. □

PROOF. Theorem 6.3. By induction on the structure of P: Proving that, rule by rule, $\mathcal{F}[\![\mathcal{E}[\![U]\!]_\rho]\!]_\tau = \mathcal{E}[\![\mathcal{F}[\![U]\!]_\tau]\!]_\rho$ holds for each component U to which the rule applies (and any correct pair of environment ρ and τ). □

PROOF. Theorem 6.4. Immediate from Theorems 6.1 and 6.2 on completeness and Theorem 6.3 on orthogonality of the component translations. □

C. APPENDIX - MC_PARAMETERS: CLASS LITERAL AND VARIABLE ARITY METHODS

We confine mc_parameters invocations to the syntax below (which essentially forbids the use of mc_parameter in invocation mode `super` (see Section 15.12.3 in [GJSB05])):

PrimarySelector::= *PrimarySelector* . *[<TypeList>] Identifier Arguments*

where *Identifier* may be the name of either a method or a mc_parameter. *Primary Selector* is either a target reference or a class literal (see Section 15.12.4 in [GJSB05]). In the first case, the expression computes the object on which the method or the mc_parameter must be invoked. This is the case considered in Section 4.2 and Section subsec:transform for mc_invocations. In particular, interfaces `ApplyClass$nk` and `ApplyClassV$nk`, and the translation rules apply correctly to the mc_parameter when it is bound to an object (fixed arity) method as well as to a class (fixed arity) method. However, when the bound method has variable arity arguments the interfaces `ApplyClass$nk` and `ApplyClassV$nk` are unusefull since do not furnish way to deal with variable arity arguments. Again, when *Primary Selector* is a class literal, both the interfaces and the translation rules do not apply correctly since rule $\mathcal{E}[\![PS]\!]_\rho$, for instance, assumes that PS_1 is an expression computing an object but it is a class literal (thus its use yelds a wrong translated program with a type error in the first argument of `Apply`).

C.1. The Type Constructor `funClass` and the Value Costructor `absClass`

In the use of class methods that are bound to a mc_parameter, we distinguish, at the user level, between the case in which parameter invocations are with *PrimarySelector*

computing an object from those in which *PrimarySelector* is a class literal. This is accomplished extending the syntax as below:

FType::= *(*`fun` | `funClass`*)RootClass*:`(`*[ExtendedTypeList]*`)`→*(*`void`|*Type)ThrowOpt*
AExp::= *(*`abs` | `absClass`*) MethodSpecifier* `from` *Type*

Type constructor `funClass` is for the type of mc_parameters that are bound to class methods and invoked using class literals as *PrimarySelector*. Value constructor `absClass` is for class methods that can be passed to mc_parameters which are always invoked using class literals as *PrimarySelector*. Hence, `fun` and `abs` are used, as described in Section 4, for the type of mc_parameters and for the object and class methods bound to mc_parameters that are invoked using target references as *PrimarySelector*. Hence, in the following we revise and extend the intended semantics and the next subsection we consider how interfaces must be extended to support mc_parameters invoked using class literals and to support methods with variable arity arguments. Then we conclude the section with the rules for the translation semantics of these new forms.

(iii) Occurrence $e.p(e_1,...,e_n)$, where expression e is a target reference, inside the body of a HO method invoked with argument `abs` I `(`EL`)`→T *[*`throws` TL*]* `from` T_a supplied for parameter `fun` T_f`:(`EL_{T_f}`)`→T_{T_f} *[*`throws` TL_{T_f}*]* p, the invocation of method $\mathcal{M}(I,EL_{T_a},T,TL_{T_c},T_c)$ where T_c is the effective (i.e. run-time) class of the object computed by e and EL_{T_a} are correct types for the argument list $E_1,...,E_n$. If e is not a target reference then sentence (iv) below must apply.

(iv) Occurrence $c.p(e_1,...,e_n)$, where expression e is a class literal, inside the body of a HO method invoked with argument `absClass` I `(`EL`)`→T *[*`throws` TL*]* `from` T_a supplied for `funClass` T_f`:(`EL_{T_f}`)`→T_{T_f} *[*`throws` TL_{T_f}*]* p, the invocation of method $\mathcal{M}(I,EL_{T_a},T,TL_c,c)$ where $c \equiv c_f \equiv c_a$ and EL_{T_a} are correct types for the argument list $e_1,...,e_n$.

If none of (iii) and (iv) applies then the program contain an uncorrect use of mc_parameter.

C.2. Semantics Structures for `funClass` and `absClass`

If the passed class method is invoked using a class literal then point 1 of the call-back methodology (see Section 4.2) requires two specific interfaces `ApplyClassS$nk` and `ApplyClassSV$nk`, for nonvoid and void respectively, without the first parameter. Point 2 has method `Apply` without first argument and with 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. In addition to this, other modifications to the form of interfaces are needed in order to consider *variable arity* method: interfaces change a bit and points 1-3 are modified to consider the use of the syntactic form *'...'*, introduced in Java, for the occurrence of a *variable arity parameter* (see Section 8.4.1 [GJSB05]). Table 1 summarizes all these modifications: It shows the form of the four families of interfaces, the structure of the four families of classes of method objects, the form of the invocation of the method `Apply` for the four families. Note that, the needs

Interfaces with, possibly, a variable arity parameter *[,ET...y]*

```
public interface ApplyClass$nk<RT, ET[,ET], T, QI "extends Throwable"{
   public T apply(RT o, ET x[,ET...y]) throws QI;}
public interface ApplyClassV$nk<RT, ET[,ET], QI "extends Throwable"{
   public void apply(RT o, ET x[,ET...y]) throws QI;}
public interface ApplyClassS$nk<RT, ET[,ET], T, QI "extends Throwable"{
   public T apply(ET x[,ET...y]) throws QI;}
public interface ApplyClassSV$nk<RT, ET[,ET], QI "extends Throwable"{
   public void apply(ET x[,ET...y]) throws QI;}
```

Method Objects with, possibly, a variable arity parameter *[,ET...y]*

```
static class I[m_i] implements ApplyClass$n_ik_i<C<Ta>, ET_ai[,ET], T_i, T_ti>{
 public T_i Apply(C<Ta> o, ET_ai x [,ET...y]) throws T_ti{
     return (o.m_i(x [, y]);}}
static class I[m_i] implements ApplyClassV$n_ik_i<C<Ta>, ET_ai[,ET], T_ti>{
 public Apply(C<Ta> o, ET_ai x [,ET...y]) throws T_ti{
     o.m_i(x [, y];}}
static class I[m_i] implements ApplyClassS$n_ik_i<C<Ta>, ET_ai[,ET], T_i, T_ti>{
 public T_i Apply(ET_ai x [,ET...y]) throws T_ti{
     return (C<Ta>.m_i(x [, y]);}}
static class I[m_i] implements ApplyClassSV$n_ik_i<C<Ta>, ET_ai[,ET], T_ti>{
 public T_i Apply(ET_ai x [,ET...y]) throws T_ti{
     C<Ta>.m_i(x [, y];}}
```

HO Method $\mathtt{I_{ho}}$ with an invocation of a method object o

```
public ET_Iho Iho (... ApplyClass$nk<ET_e, ET [,...ET_y], T, T> o ...)
   {... o.Apply(E_e, E [,...E_y]) ...}
public ET_Iho Iho (... ApplyClassV$nk<ET_e, ET [,...ET_y], T> o ...)
   {... o.Apply(E_e, E [,...E_y]) ...}
public ET_Iho Iho (... ApplyClassS$nk<ET_e, ET [,...ET_y], T, T> o ...)
   {... o.Apply(E [,...E_y]) ...}
public ET_Iho Iho (... ApplyClassSV$nk<ET_e, ET [,...ET_y], T> o ...)
   {... o.Apply(E [,...E_y]) ...}
```

Legenda: Metavariables are ranging in the syntactic domains specified in Fig.1 and in Fig.2. Moreover we adopt the following conventions:

- $\overline{u}$ is a shortand for $u_1, ..., u_n$
- $\overline{u \text{ “}w\text{”}}$ is a shortand for $u_1 w, ..., u_n w$
- $\overline{u\ w}$ is a shortand for $u_1 w_1, ..., u_n w_n$

Table 1

for four distinct families of interfaces is due to the following two facts that hold in Java. (1) `void` is not a true type: Hence we have to distinguish between methods with a return type and methods without return type. (2) Class literal is not a true value: Hence in the invocation $e.m(e_1, ..., e_n)$ of a class method m we must distinguish between

the the following two cases: e is an expression that computes a target reference o; e is a class literal expressing the class of which m is a member.

C.3. $\mathcal{E}[\![]\!]_\rho$: Translation Semantics for `funClass` and `absClass`

Completing $\mathcal{E}[\![]\!]_\rho$ translation of ET for funClass mc_parameters

$\mathcal{E}[\![ET]\!]_\rho =$ `ApplyClassS$nk<`$\mathcal{E}[\![RT]\!]_\rho$ `[,`$\overline{\mathcal{E}[\![ET_s]\!]_\rho}$`],`$\mathcal{E}[\![T_o]\!]_\rho$ `[,`$\overline{QI}$`]>`
with $ET =$ `funClass` RT`: (`$\overline{ET_s}$`)` $\rightarrow T_o$ `[throws` $\overline{QI}$`]`

$\mathcal{E}[\![ET]\!]_\rho =$ `ApplyClassSV$nk<`$\mathcal{E}[\![RT]\!]_\rho$ `[,`$\overline{\mathcal{E}[\![ET_s]\!]_\rho}$`] [,`$\overline{QI}$`]>`
with $ET =$ `funClass` RT`: (`$\overline{ET_s}$`)` $\rightarrow$ `void [throws` $\overline{QI}$`]`

Completing $\mathcal{E}[\![]\!]_\rho$ translation of E for absClass mc_parameters

$\mathcal{E}[\![E]\!]_\rho =$ `new ApplyClassS$nk<`T_c`,`$\overline{\mathcal{E}[\![ET]\!]_\rho}$`,`$T$ `[,`$\overline{QI}$`]> (){`
`public` T `Apply(`$\overline{\mathcal{E}[\![ET]\!]_\rho}$ $\overline{x}$`) [throws` $\overline{QI}$`]{`
`return` $T_c.I(\overline{x})$`;}}`
with $E =$ `absClass` I`(`$\overline{ET}$`)` $\rightarrow T$ `[throws` $\overline{QI}$`] from` T_c

$\mathcal{E}[\![E]\!]_\rho =$ `new ApplyClassSV$nk<`T_c`,`$\overline{\mathcal{E}[\![ET]\!]_\rho}$ `[,`$\overline{QI}$`]> (){`
`public void Apply(`$\overline{\mathcal{E}[\![ET]\!]_\rho}$ $\overline{x}$`) [throws` $\overline{QI}$`]{`
$T_c.I(\overline{x})$`;}}`
with $E =$ `absClass` I`(`$\overline{ET}$`)` $\rightarrow$ `void [throws` $\overline{QI}$`] from` T_c

Completing $\mathcal{E}[\![]\!]_\rho$ translation of PS for funClass mc_parameters

$\mathcal{E}[\![PS]\!]_\rho = (T)($`I.Apply(`$\overline{\text{“(”}\ ET\ \text{“)”}\ \mathcal{E}[\![E]\!]_\rho}$`))`
with $PS =$ QI `.I (`$\overline{E}$`)` $\wedge$ $\rho(I) =$ `funClass` RC`:`$\overline{ET}$ $\rightarrow$ T TR

$\mathcal{E}[\![PS]\!]_\rho =$ `I.Apply(`$\overline{\text{“(”}\ ET\ \text{“)”}\ \mathcal{E}[\![E]\!]_\rho}$`)`
with $PS =$ QI `.I (`$\overline{E}$`)` $\wedge$ $\rho(I) =$ `funClass` RC`:`$\overline{ET}$ $\rightarrow$ `void` TR

Legenda: Metavariables are ranging in the syntactic domains specified in Fig.1 and in Fig.2. Moreover we adopt the following conventions:

- $\overline{u}$ is a shortand for $u_1, ..., u_n$
- $\overline{u\ \text{“}w\text{”}}$ is a shortand for $u_1 w, ..., u_n w$
- $\overline{u\ w}$ is a shortand for $u_1 w_1, ..., u_n w_n$
- $\overline{\mathcal{E}[\![u]\!]_\rho}$ is a shortand for $\mathcal{E}[\![u_1]\!]_\rho, ..., \mathcal{E}[\![u_n]\!]_\rho$

Table 2

REFERENCES

Igarashi A., Pierce B., and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM TOPLAS*, 23:396–450, 2001.

M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

Paul F. Albrecht, Phillip E. Garrison, Susan L. Graham, Robert H. Hyerle, Patricia Ip, and Bernd Krieg Brückner. Source-to-Source Translation: Ada to Pascal and Pascal to Ada. In *ACM-SIGPLAN symposium on The ADA programming language*, SIGPLAN '80, pages 183–193, New York, NY, USA, 1980. ACM.

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

G. Bracha, N. Gafter, J. Gosling, and P. von der Ahe. Closures for the Java Programming Language (aka BGGA), 2008. www.javac.info.

D. Lea B. Lee and J. Bloch. Concise Instance Creation Expressions: Closure without Complexity, 2006. crazybob.org/2006/10/java-closure-spectrum.html.

M. Bellia and M.E. Occhiuto. Higher Order Programming through Java Reflection. In *CS&P'2004*, volume 3, pages 447–459, 2004.

M. Bellia and M.E. Occhiuto. Higher Order Programming in Java: Introspection, Subsumption and Extraction. *Fundamenta Informaticae*, 67(1):29–44, 2005.

M. Bellia and M.E. Occhiuto. JH-Preprocessing, 2007. www.di.unipi.it/∼occhiuto/JH.

M. Bellia and M.E. Occhiuto. Methods as Parameters: A Preprocessing Approach in Java. In *CS&P'2007*, volume 1, pages 47–59, 2007.

M. Bellia and M.E. Occhiuto. JavaΩ: A Preprocessor for Java with M_parameters. In *CS&P'2008*, pages 25–34. Humboldt-Universitat zu Berlin, 2008.

M. Bellia and M.E. Occhiuto. JavaΩ: The Structures and the Implementation of a Preprocessor for Java with m_parameters, 2008.

M. Bellia and M.E. Occhiuto. Methods as Parameters: A Preprocessing Approach to Higher Order in Java. *Fundamenta Informaticae*, 85(1):35–50, 2008.

M. Bellia and M.E. Occhiuto. JavaΩ: A Translation Semantics for Closures in Java. In *CS&P'2009*, pages 72–83. Warsaw University, 2009.

M. Bellia and M.E. Occhiuto. A Preprocessing for Java with M_ and MC_parameters. *Fundamenta Informaticae*, 93(1):35–50, 2009.

M. Bellia and M.E. Occhiuto. JavaΩ: Proving Type Safety for Java Simple Closures. In *CS&P'2010*, pages 61–72. Humboldt-Universitat zu Berlin, 2010.

R. Biddle and E. Tempero. Understanding the Impact of Language Features on Reusability. In *Fourth International Conference on Software Reuse*. IEEE Computer, 1996.

Python Community. The Python Language 3.1, 2010. http://docs.python.org/py3k/reference/.

C.Donnely and R. Stallman. Bison: The YACC-compatible Parser Generator, 2006. www.gnu.org/software/bison/manual.

S. Colebourne, S. Shulz, and R. Clarkson. FCM+JCA, 2008. http://www.jroller.com/scolebourne/entry/fcm_closures_options_within.

M. Felleisen. On the Expressive Power of Programming Languages. *Sci. Comput. Program.*, 17:35–75, December 1991.

D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008.

N.M. Gafter. JSR Proposal: Closures for Java, 2007. JavaCommunity Process, www.javac.info/consensus-closure-jsr.html.

N.M. Gafter. Java Closures Prototype Feature-Complete, 2008. //gafter.blog- spot.com/2008/08/java-closures-prototype-feature.html.

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

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

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

B. Goetz. The Closures Debate: Should Closures be Added to the Java Language, and if so, How?, 2007. Java Theory and Practice, IBM Technical Library, www.ibm.com/developerworks/java/library/j-jtp04247.html.

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

J. Koser, H. Larsen, and J. A. Vaughan. SML2Java: a Source to Source Translator. In *Proceedings of DP-Cool, PLI03, Uppsala, Sweden*, 2003.

P. Lakshman. The Delegate Type in J#, 2004. http://msdn.microsoft.com/en-us/library.

P.J. Landin. A λ−Calculus Approach. In *Andvances in Programming and Non-numerical Computation, Ed. L. Fox*, pages 97–141. Pergamon Press, 1966.

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

B. Liskov and J.M. Wing. Definition of the Subtype Relation. In *ECOOP'93, Object Oriented Programming*, volume 707 of *LNCS*, pages 118–141. Springer, 1993.

T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series, Addison Wesley, 1996.

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.

M. Odersky. The Scala Language Specification, 2010. http://scala-lang.org/node/89.

M. Reinhold. Project Lambda: Straw-Man Proposal, 2009. //cr.openjdk.java.net/~ mr/lambda/straw-man/.

D. Syme. F# 2.0 Language Specification, 2003. http://fsharp.net.

Z. Tronicek08. Java Closures Tutorial, 2008. //gafter.blogspot.com/2008/08/java-closures-prototype-feature.html.

M.P. Ward. Reverse Engineering through Formal Transformation: Knuths 'Polynomial Addition' Algorithm. *Comput. J.*, 37(9):795–813, 1994.

S. Wiltamuth and A. Hejlsberg. C# Language Specification 2.8, 2003. http://msdn.microsoft.com/en-us/library.