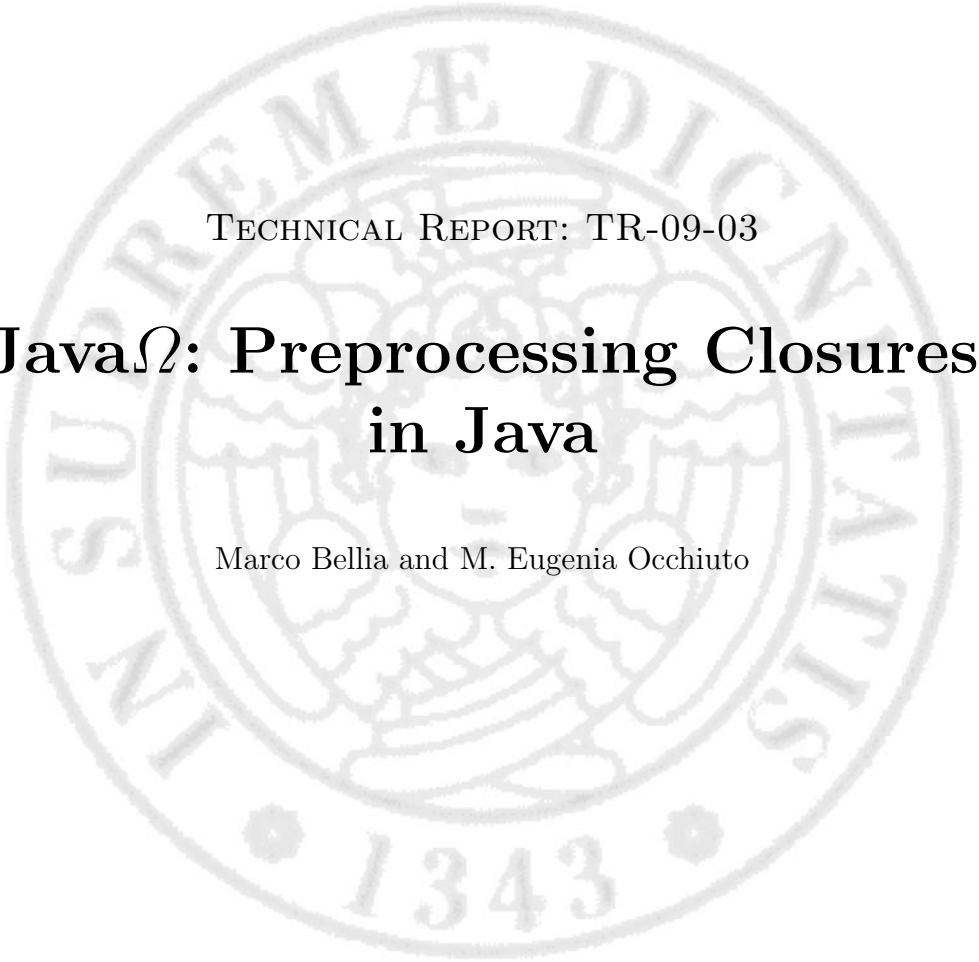
Università di Pisa
Dipartimento di Informatica



Technical Report: TR-09-03

Java∩: Preprocessing Closures in Java

Marco Bellia and M. Eugenia Occhiuto

February 12 2009

address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. tel: +39 050 2212700 fax: +39 050 2212726

JavaΩ: Preprocessing Closures in Java

Marco Bellia and M. Eugenia Occhiuto

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

Abstract. The paper adds a mechanism of *closure* to Java. We apply to closures the same technique we exploited in extending Java with methods as parameters [BO08c,BO08a] and we obtain a formal definition and a prototype of Java with closures. The formal definition consists of a set of source to source translation rules that state the meaning of the new construct in terms of compositions of well known ordinary Java structures. Two variants of the transformation are discussed to allow recursive defined closures and other mechanisms. The notion of *shared variable* as a local variable that is allocated in the heap is also discussed. Eventually, since the resulting transformation is one pass process, it can be implemented through a source to source, one pass, preprocessor [BO07,BO08b], easy to write using standard development tools [LMB95,CS06a].

1 Introduction

In the last few years extensions to Java focus on higher order mechanisms to enhance expressivity, conciseness, good structuring, reusability, and factoring of code [OW97,Mei01,McC01,Set03,Cor04,Mic04,BO05,Bri05,GHJV05,SS07]. In [BO08c] Java is extended with mechanisms which allow methods to have other methods as parameters. In that paper we also argued the improvement of the expressive power of languages including such kind of mechanisms, in particular code structuring, reusability and as a consequence code correctness. The extended language semantics is defined through a meaning preserving $\mathcal{E}[\![]\!]_\rho$ transformation, which is defined through a set of source to source translation rules that map extended programs into programs of ordinary Java. The implementation is obtained encapsulating such rules into a one pass preprocessing [BO07], designed using standard tools [LMB95,CS06a].

In this paper, we apply the same approach to the definition and implementation of *closures* in Java. A mechanism of Closures for Java has been discussed since 2006 [BLB06,CS06b,BGGvdA06], next section revisits aims and results of those proposals. Section 3 provides for closures and related mechanisms to use them: i) syntax definition, inspired to above mentioned proposals [BLB06,CS06b,CSC07,BGGvdA06,Goe07,Gaf07], and ii) formal semantics definition. Section 4 introduces transformation $\mathcal{E}[\![]\!]_{\rho,\tau}$ through a set of source to source rules that map closures and the language structures extended to use them, into the meaning equivalent, ordinary, Java structures. Hence, these rules fix the semantics of closures in terms of the well known structures of the language Java itself and help in having a precise view on the use of some constructions, for instance self reference `this`, that are discussed later on in Section 6. The transformation appears to be parametric at some extent and this definition allows to obtain, in a ease way, two variants. One for recursive defined closures. The other one for the use of self reference `this`. Section 5 collects in one program the definition and use of several, simple, different closures with the aim of showing how $\mathcal{E}[\![]\!]_{\rho,\tau}$ works. Section 6 addresses special

situations arising in Java extended closures: recursive defined closures, access and modification of non local identifiers (variables and parameters). Eventually, transformation $\mathcal{E}[\![]\!]_\rho$ is defined for Java 1.4 [GJSB00], and so is its implementation [BO07] and its extension $\mathcal{E}[\![]\!]_{\rho,\tau}$. However, the translation rules we give in Section 4 are not tailored for Java 1.4. Section 7 discusses how $\mathcal{E}[\![]\!]_{\rho,\tau}$ can be formulated to cope with generic programming of Java 1.6 [GJSB05]. The new formulation appears as a third variant of $\mathcal{E}[\![]\!]_{\rho,\tau}$ since it consists of an extension, to generic types, of the structures introduced for generating new class and interface names. $\mathcal{E}[\![]\!]_{\rho,\tau}$, extended in this way, is applied to the definition of closures for streamed linear Fibonacci numbers, proposed in [Dup08].

2 Closures

In programming languages, *closures* were introduced by Peter J. Landin as structures [Lan66] that comprise a $\lambda-$expression (identifier, function abstraction or application) and an environment, containing bindings for identifiers (that possibly, occur free in the expression), relative to which the expression is evaluated. In fact, John McCarty had already, briefly addressed the problem of the evaluation of functional values and had introduced *funarg* lists, a quite similar structure, in the LISP interpreter [McC60]. In that case, the use was limited to handle functional arguments and the context was different, since the language is the calculus of S-expressions [MAD$^+$62] and uses the dynamic scope rule. Accordingly, a closure is a semantic (and also implementation) counterpart for program evaluation, in functional programming. With respect to other evaluation techniques [Jon87,PZ00], closures have the advantage, among others, to furnish support for freezing, suspending and resuming evaluations (which is at the basis of delayed evaluation strategies such as call-by-name, call-by-need and lazy evaluation [Pie02]). In other words closures make evaluation a value (of the machine running the language). It is in this respect that objects (including Java objects), though containing bindings (for variables) and also program code (namely, methods) are structures that behave different from closures. In effect, "adding closures to Java" means making available in Java, a construct for function abstraction (also called, anonymous function definitions), that is, a construct that encloses a piece of code, possibly parameterized, and that makes available for possibly, many and different invocations in the program, the "function" defined by such code. This is a (Java) closure and it must behave as a value (closure objects [Gaf07]) that can be computed by and passed to Java methods. A (Java) closure, hereafter called closure, may contain free variables (not identifiers) that are bound in the lexical (i.e. static) scope of the blocks enclosing the closure definition, are allocated either in the heap or in the stack, and are accessed and updated everywhere from the code in the variables scope. Closures can be very usefully employed in server side applications and aim to: (a) simplifying many kinds of programs that currently rely on anonymous inner classes; (b) enabling more powerful libraries with methods (*Control APIs*) in which the controlled statements are received as closures; (c) improving concurrent programming and simplifying applications that rely on existing concurrency APIs; (d) supporting a programming style that enhances the use of aggregate operations, i.e. operations that apply an operation to collection members, in the concurrency framework; (e) enabling future API design to replace language design for extending the Java platform. In fact, at the base of all the above points is the verbosity and the awkwardness of anonymous inner classes in expressing anonymous functions and more in general, parameterized code. In

addition, the Java scope rules limit, in inner classes, the use of non local variables (or class members) to those that are declared `final` in the method code.

The idea of adding closures to Java was in three specific proposals since 2006 [BLB06,BGGvdA06,CS06b]. Brian Goetz in [Goe07] summarizes the main points of the first two, while [Blo07] discusses some of the most controversial points.

2.1 The BGGA proposal

In [BGGvdA06], the closure has the syntactic form of an expression and the intended meaning of an *anonymous function* value. These values have *function types* that extend Java type structure. A function type defines a signature for the types of the arguments list, on which each closure with such a function type may be invoked, a type for the result and possibly, a list of exception types that can be thrown. The syntax (and the notation) of closures and of function types is close to the one given in 3, and used in our formalization. A function type has the meaning of a generic interface whose type parameters only depend on the reference types involved in the function type. The interface is single method and the method has name *invoke* and signature, result type and throwable exceptions those defined in the function type. Function types are subtyped by contra-covariance [AC96]. Function type variables and parameters may be declared and closures may be assigned to a variable or passed to a method as well as any other Java value. In addition, a mechanism for *Closure conversion* makes a closure assignable to a variable (parameter) whose type is a single method interface, provided that the interface method has signature compatible with the closure function type. The mechanism is useful in using closures with current Java APIs. Eventually, the approach has a mechanism for *loop abstraction* [Gaf08,Tro08] which on one hand 1) provides `break`, `continue` and `return`, occurring inside a closure, with a new semantic that transfers control outside the closure, on the other hand 2) it makes closures more similar to C macros, where invocation is by macro expansion (i.e. syntactic source code replacing). In this way, closures do not have a standard semantics since they have different meanings in correspondence of the different constructs in which the closure invocation occurs. For further details and material see [BGGvdA08].

2.2 The CICE proposal

The approach [BLB06] is quite conservative and starts from the observation that *single-abstract-method types* or SAM types, are single method interfaces or abstract classes, whose anonymous instances are expressions that already behave as a form of closures in Java. Hence, the approach focuses on reducing the verbosity and the awkwardness of such expressions. The *concise instance creation expression*, CICE for short, is an expression with the following syntactic structure and associated semantics, on the right.

```
Sam({Pars})Body   means  new Sam(){
                         modifier ResultType Ide({Pars})Throws Body}
```

Sam is an interface or abstract class of SAM type, *Pars* and *Body* are the formal parameter list and the parameterized code, respectively, of the closure. The expression is translated into the anonymous class on the right. The translation retrieves the access modifier *modifier*, the computed type *ResultType*, the throwable exception list *Throws*, and a name *Ide* for invoking the closure, in the interface declaration. The use of *Ide* in

the invocation makes CICE closures a bit less anonymous since there is not a uniform invocation form as in the BGGA's use of *invoke* for all the closures, and it requires the knowledge of the *Ide* which is specific for the SAM type. Given a closure, the corresponding anonymous class is generated at compile time. Closures have only SAM types hence there is no need for *Closure conversion* in using closures in the API's. Eventually, closures may access and modify non local variables which are declared *public* in the scope of the enclosing block. As a matter of fact, the proposal does not contain details on the semantics of these variables, in particular on the allocation: hence it is not clear what happens when the execution of the closure accesses public non local variables of an activation frame which is no more active, as in the examples in 6 later on. Further considerations are in [Lee06].

2.3 The FCM proposal

The proposal [CS06b] contains also the introduction of two different ways to express class and object methods as parameters. They are: *method literals* which are methods obtained at run time as objects of *java.lang.reflect.Method* and *method references* which are methods obtained through anonymous instances of single method interfaces or abstract classes, generated at compile time. They share the same syntactic structure, hence have a non standard semantics that determines on the basis of the surrounding context, for which kind of method, i.e. literal or reference, the corresponding code must be generated. Java types structure is extended with *method types* which qualify parameters and variables that are bounds to method references or to closures. Closures are called *inner* methods and have a structure similar to the one in CICE, i.e. that of ordinary Java methods. The proposal includes a construct for *named* inner methods as a combination of closures assigned to final variables. The access to non local variables and the meaning of `this` is as in BGGA. Eventually, The proposal includes a mechanism similar to BGGA's closure conversion. The mechanism has a non standard semantics, as a matter of fact it generates two different codes for a same closure in correspondence of the type (namely, reference or method type) of the variable to which the closure is assigned to. Further details and material are in [CSC08] where a separate proposal, *JCA*, for control abstraction is compared to the BGGA's loop abstraction mentioned in Section 2.1.

3 Syntax and semantics of closures.

From a syntactic point of view, a closure is an expression. The syntax is similar to [BGGvdA08], but in addition we specify the *ResultType*, i.e. the type of the value that each invocation of the closure is expected to compute. Below, an expression for defining a closure which, given an integer value for parameter y returns a value of type `int`. The body contains a non local variable x.

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

The declaration of the result type does not constraint closure implementation, i.e. the semantics does not force the implementation to infer the result type as in the proposal of [BGGvdA08]. However, we defer to type checking, of the ordinary Java compilers, the burden of checking for the correctness of the types declared in the transformed programs. Hence, the syntax of *Expr* is further extended adding the following productions:

```
Exp::= Identifier | ClosureLiteral | OtherExpressions
ClosureLiteral::= ‘{FormalParameters:ResultType [ Throws ]⇒
                                                BlockStatements ‘}
ClosureLiteral::= ‘{FormalParameters:void [ Throws ]⇒
                                                BlockStatements ‘}
ResultType::= FType
BlockStatements::={BlockStatement}
BlockStatement::=Statement | LocalVariableDeclarationStatement
```

A closure has a type and it may be assigned to a variable, passed as a parameter of the right type, returned as a value, invoked as a method using selector *invoke*. The syntax of *FType* and the syntax of *Selector* are further extended adding:

```
FType::= ‘{ FList : Type [ Throws ]‘}
FType::= ‘{{FTList} : void [ Throws ]‘}
Selector::= .invoke Arguments
Throws::= throws ExceptionList
FList::=[FType {,FType}]
```

Eventually, closures generalize *functional abstraction* [Pie02] since 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, avoiding the cumbersome lambda lifting technique [Jon87]. In particular a closure can use variables bound in a scope that is not active [LY96] at the time the closure is invoked (see [Tro08] and example in Fig. 4. A solution of this problem leads to a new notion of local variables that must be allocated in the heap [Tro08] instead of in the control stack. In [Gaf08] this kind of variables are annotated *@shared* to mark the difference with ordinary local variables. In our proposal, *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 involves memory allocation (heap vs. frame [LY96]), access and modification of the variable. Hence the syntax of the local variable declaration [GJSB00] is extended adding:

```
LocalVariableDeclarationStatement::= [final] FType VariableDeclarator
LocalVariableDeclarationStatement::= shared FType VariableDeclarator
FormalParameter::= [final] FType VariableDeclaratorId
FormalParameter::= shared FType VariableDeclaratorId
```

The semantics of closures associates to:

(i) closure type $\{t_1, ..., t_n : t_0\}$, a one-to-one correspondence with a reference type named $s_1\$...\$s_n\$s_0$. Where, for each i, s_i is the type associated to t_i from such a correspondence.

(ii) closure $\{t_1v_1, ..., t_nv_n{:}t_0 \Rightarrow B\}$, an anonymous function object of type $s_1\$...\$s_n\$s_0$ that computes the value of type s_0 that results from the execution of the ordinary Java code $\{B'\}$ assumed that x_1 is bound to a value of type s_1 and,..., and x_n is bound to a value of type s_n. Where $s_1\$...\$s_n\$s_0$ is the type associated to the closure type $\{t_1, ..., t_n : t_0\}$, and B' is the code for the sequence of statements (and/or declarations) B such that each free variable, if any, is bound to the corresponding shared variable of the lexical scope in which the closure occurs.

(iii) closure invocation $C.\texttt{invoke}(e_1, ..., e_n)$, the invocation of the anonymous function object C with arguments $e_1, ..., e_n$.

(iv) shared variable x of type t, a reference to a variable of a class of variables of type t that are allocated in the heap. The shared variable associated to x is accessed and modified through such a reference.

4 The transformation $\mathcal{E}[\![]\!]_{\rho,\tau}$.

The transformation $\mathcal{E}[\![]\!]_{\rho,\tau}$ for *closures*, defined in Fig. 1, is based on the structures of *interfaces*, *anonymous inner classes*, and *classes* of variables. The transformation associates to each closure type an interface, generated at preprocessing time, having only one method named `invoke` and typed as below:

$$(i)\ \texttt{interface}\, s_1\$...\$s_n\$s_0\{\texttt{public}\, s_0\, \texttt{invoke}(s_1\, x_1, ..., s_n\, x_n)\}$$

The interface has name computed by a function $\mathcal{N}(s_1, ..., s_n, s_0)$. The name depends on the types $s_1, ..., s_n$ and s_0 (or `void`) that are associated to closure parameter types and to result type.

A closure is transformed into an anonymous inner class of the right interface with method `invoke` defined as below:

$$(ii)\ \texttt{new}\ s_1\$...\$s_n\$s_0\{\texttt{public}\ s_0\ \mathit{invoke}\,(s_1\, x_1, ..., s_n\, x_n)\{B'\}\}$$

with B' as the code that results from preprocessing B according to $\mathcal{E}[\![]\!]_{\rho,\tau}$.

Closure invocation is transformed by preprocessing C, and each argument e_i

$$(iii)\ C'.\texttt{invoke}(e'_1, ..., e'_n)$$

with C' as the anonymous inner class and each e'_i as the code that results from preprocessing C and each e_i according to $\mathcal{E}[\![]\!]_{\rho,\tau}$.

Shared variables of type t are transformed into references to objects of a class, generated at preprocessing time, having name, $\mathcal{L}(s)$, that depends on the type s associated to t. Such an object has only one field of name *value* and of type s, and one argument constructor that initializes *value* to the value that is passed to it. The declaration

$$(iv)\ \texttt{shared}\ t\ x = e \quad \mathit{is\,replaced\,by} \quad \texttt{final}\ \mathcal{L}(s)\ x = \mathit{new}\ \mathcal{L}(s)(e')$$

where e' is the preprocessing of e according to $\mathcal{E}[\![]\!]_{\rho,\tau}$.

Eventually, each occurrence of a variable x declared shared, is replaced by the reference to the shared variable, namely *x.value*. Analogously, shared parameters of type t are transformed in references to objects of a class $\mathcal{L}(s)$, where s is the type that the transformation associates to t. For each shared parameter p which is declared shared in the header of a method, in the top of its body is inserted:

$$(iv')\ \texttt{final}\ \mathcal{L}(s)\ \mathit{par\$x} = \mathit{new}\ \mathcal{L}(s)(p)$$

where *par*\$*x* is the name for a new variable for p, and each occurrence of parameter p in the method body (and in the closures defined in it) is replaced by the reference to the new shared variable, namely *par\$p.value*. In the transformation $\mathcal{E}[\![]\!]_{\rho,\tau}$, τ is the list of variables (and parameters) that are declared shared in the current scope.

$$\mathcal{E}[\![Exp]\!]_{\rho,\tau} = \begin{cases} \downarrow(\mathcal{E}[\![]\!]_{\rho,\tau})(OtherExpression) & \text{with } Exp = OtherExpression \\ \tau(Identifier) & \text{with } Exp = Identifier \\ \texttt{new}\ s_1\$...\$s_n\$s_0()\{ & \text{with } Exp = \{t_1x_1, ..., t_nx_n{:}t_0 \Rightarrow B\} \\ \quad \texttt{public}\ s_0\ \texttt{invoke(}\texttt{s}_1\ \texttt{x}_1\texttt{,...,s}_n\ \texttt{x}_n\texttt{)}\{ & \quad s_1\$...\$s_n\$s_0 = \mathcal{N}(s_1, ..., s_n, s_0) \\ \quad\quad \mathcal{E}[\![B]\!]_{\rho,\tau^{iv}}\} & \quad s_i = \mathcal{E}[\![t_i]\!]_{\rho,\tau} \\ \texttt{new}\ s_1\$...\$s_n\$void()\{ & \text{with } Exp = \{t_1v_1, ..., t_nv_n{:}\texttt{void} \Rightarrow B\ \} \\ \quad \texttt{public void invoke(}\texttt{s}_1\ \texttt{x}_1\texttt{,...,s}_n\ \texttt{x}_n\texttt{)}\{ & \quad s_1\$...\$s_n\$void = \mathcal{N}(s_1, ..., s_n, \texttt{void}) \\ \quad\quad \mathcal{E}[\![B]\!]_{\rho,\tau^{iv}}\} & \quad s_i = \mathcal{E}[\![t_i]\!]_{\rho,\tau} \end{cases}$$

where: $\downarrow(\mathcal{E}[\![]\!]_{\rho,\tau})(OExp)$ is the morphic application of $\mathcal{E}[\![]\!]_{\rho,\tau}$ on $OExp$ components, namely:
$\downarrow(\mathcal{E}[\![]\!]_{\rho,\tau})(f(e_1, ..., e_n)) = f(\mathcal{E}[\![e_1]\!]_{\rho,\tau}, ..., \mathcal{E}[\![e_1]\!]_{\rho,\tau})$ for any $OExp$ constructor (i.e. injection) f

$$\mathcal{E}[\![FType]\!]_{\rho,\tau} = \begin{cases} Type & \text{with } FType = Type \\ \mathcal{N}(s_1, ..., s_n, s_0) & \text{with } FType = \{t_1, ..., t_n : t_0\} \\ \mathcal{N}(s_1, ..., s_n, \texttt{void}) & \text{with } FType = \{t_1, ..., t_n : \texttt{void}\} \end{cases}$$

$$\mathcal{E}[\![LDecl]\!]_{\rho,\tau} = \begin{cases} [\texttt{final}]\mathcal{E}[\![FType]\!]_{\rho,\tau}\ \tau(Ide) = \mathcal{E}[\![Exp]\!]_{\rho,\tau} & \text{with } LDecl = [\texttt{final}]\ FType\ Ide = \text{Exp} \\ \texttt{final}\ A\ Ide = \texttt{new}\ A(\mathcal{E}[\![Exp]\!]_{\rho,\tau''}) & \text{with } LDecl = \texttt{shared}\ FType\ Ide = \text{Exp} \\ & \quad A = \mathcal{L}(\mathcal{E}[\![FType]\!]_{\rho,\tau}) \\ \texttt{final}\ A\ Ide = \texttt{new}\ A() & \text{with } LDecl = \texttt{shared}\ FType\ Ide \\ & \quad A = \mathcal{L}(\mathcal{E}[\![FType]\!]_{\rho,\tau}) \end{cases}$$

$$\mathcal{E}[\![MDecl]\!]_{\rho,\tau} = \begin{cases} \mathcal{E}[\![FType_1]\!]_{\rho,\tau}\ \ Ide([\ \texttt{final}]\ \ \mathcal{E}[\![FType_2\ ]\!]_{\rho,\tau}\ p)\ \{\mathcal{E}[\![BStms'\ ]\!]_{\rho,\tau'''}\} \\ \qquad\qquad\qquad\qquad \text{with} \quad MDecl\ =\ Ftype\ Ide([\texttt{final}]FType\ p)\{BStms\} \\ \mathcal{E}[\![FType_1]\!]_{\rho,\tau}\ \ Ide(\mathcal{E}\ [\![FType_2\ ]\!]_{\rho,\tau}\ p)\ \{\mathcal{E}[\![\ \texttt{shared}\ B\ par\$p = p;\ BStms'\ ]\!]_{\rho,\tau'''}\} \\ \qquad\qquad\qquad\qquad \text{with} \quad MDecl\ =\ Ftype\ Ide([\texttt{shared}]FType\ p)\{BStms\} \end{cases}$$

$$\mathcal{E}[\![BStms]\!]_{\rho,\tau} = \begin{cases} \mathcal{E}[\![Stm]\!]_{\rho,\tau}; \mathcal{E}[\![BStms]\!]_{\rho,\tau''} & \text{with } BStms = Stm;BStms \\ \mathcal{E}[\![LDecl]\!]_{\rho,\tau}; \mathcal{E}[\![BStms]\!]_{\rho,\tau'} & \text{with } BStms = LDecl;BStms \\ & \quad LDecl = \texttt{shared}\ FType\ Ide\ [= Exp] \end{cases}$$

where: $BStms' \equiv BStms$
$\tau^{iv} \equiv \tau''' \equiv \tau'' \equiv \tau$ and $\tau' \equiv \mathcal{S}[\![Ide]\!]_\tau$
$\mathcal{S}[\![y]\!]_\tau(x) = x.\texttt{value}$ if $y = x$ $\quad$ $\mathcal{L}$ is injective into new Class identifiers
$\mathcal{S}[\![y]\!]_\tau(x) = \tau(x)$ if $y \neq x$ $\quad$ $\mathcal{N}$ is injective into new Interface identifiers

a: Transformation $\mathcal{E}[\![]\!]_{\rho,\tau}$ for closures and non local variables

$\mathcal{E}[\![Stm]\!]_{\rho,\tau} \equiv \tau(Ide) = \mathcal{E}[\![Exp]\!]_{\rho,\tau''}$ with $Stm \equiv Ide = Exp$ and $Exp \equiv \{t_1x_1, ..., t_nx_n{:}t_0 \Rightarrow B\}$
where: $BStms' \equiv BStms$
$\tau^{iv} \equiv \tau''' \equiv \tau$ and $\tau'' \equiv \mathcal{S}[\![(Ide, \texttt{this})]\!]_\tau$, $\tau' \equiv \mathcal{S}[\![(Ide, Ide.value)]\!]_\tau$
$\mathcal{S}[\![(y, Val)]\!]_\tau(x) = Val$ if $y = x$
$\mathcal{S}[\![(y, Val)]\!]_\tau(x) = \tau(x)$ if $y \neq x$

b: Variant for recursive defined closures

$\mathcal{E}[\![Exp]\!]_{\rho,\tau} = self.value$ with $Exp = \texttt{this}$, $\tau(this) = (self, on)$
where: $BStms' \equiv \texttt{shared}\ A\ self = \texttt{this}; BStms$
$\tau''' \equiv \mathcal{S}[\![(this, (self, off))]\!]_\tau$, $\tau^{iv} \equiv \mathcal{S}[\![(this, (self, on))]\!]_\tau$ with $\tau(this) = (self, \_)$

c: Variant for closures that use `this`

Fig. 1. Transformation $\mathcal{E}[\![]\!]_{\rho,\tau}$ for closures and non local variables

4.1 The formal definition of $\mathcal{E}[\![]\!]_{\rho,\tau}$

The formal definition of $\mathcal{E}[\![]\!]_{\rho,\tau}$, given in Fig 1, consists of a main definition, Fig 1.a, and two additional variants, in Fig 1.b-c. The main definition is constitued of a set of source-to-source translation rules on the syntax of Java programs that is affected by adding the mechanism of closures. $\mathcal{E}[\![]\!]_{\rho,\tau}$ applies to each method (of each class) of the source program, separately, producing a corresponding method and auxiliary definitions (for anonymous class interfaces and non local variables) in Java 1.4. $\mathcal{E}[\![]\!]_{\rho,\tau}$ is compositional, descends on the syntactic structure of the class, and is indexed by parameters ρ and τ. ρ is an environment introduced in [BO08a] to deal with Java extended with m_ and mc_parameter and is not specifically involved in transforming closures. However, ρ is maintained to underline that the two extensions are albeit orthogonal and can be straightforwardly integrated. As a consequence, the present transformation is seen as a further extension of the transformation presented in [BO08a]. Hence, $\mathcal{E}[\![]\!]_{\rho,\tau}$ allows closures to coexist with m_ and mc_parameters. The main definition contains the translation rules for closures in the basic structure of an anonymous non recursive function. In this case, τ is just the list of variables and parameters that are declared shared in the scope in which the closure is defined, and hence, the closure can access. τ is updated according to τ' in the rules for $BStms$: these rules apply also in case of shared parameters since the rules for $MDecl$ add a special shared variable when a shared parameter is encountered. τ is unchanged in the other rules, hence $\tau^{iv} \equiv \tau^{iii} \equiv \tau^{ii} \equiv \tau$, as well as the code $BStm'$, in the second rule for $Mdecl$, that is transformed to form the code of the anonymous function, hence $BStm' \equiv BStm$. The two variants in Fig 1 show how the main definition can be modified to support recursively defined closures, variant (b), and closures using `this`, i.e. the self-reference to the object on which the method that defines the closure is invoked, variant (c).

4.2 A variant of $\mathcal{E}[\![]\!]_{\rho,\tau}$ for recursively defined closures

Variant (b) supports recursively defined closures by transforming each reference, in the closure, to the identifier to which the closure is assigned, into a self reference `this` to the anonymous function (inner class) that the transformation produces. This can be obtained by using τ as an environment, instead of a list, that contains bindings for the identifiers. For variable or parameter identifiers in the main definition, we have that τ' is extended with a binding $(Ide, Ide.value)$, instead of a single identifier Ide. For the identifier on the left of a closure declaration or assignment, we have that τ'' is extended with binding $(Ide, \texttt{this})$ and the assignment of a closure is transformed with the rule in the variant (b), that uses τ'', instead of τ, in transforming the closure on the right of the assignment. The transformation is correct provided that the closure does not contain occurrences of `this`.

4.3 A variant of $\mathcal{E}[\![]\!]_{\rho,\tau}$ for closures referencing `this`.

Variant (c) supports closures with the self reference `this`. First of all, note the meaning of `this` in a closure: it is the object on which the method that defines the closure is invoked. Hence, variation (c) adds statement `shared` A `self` = `this` at the beginning of method body $BStms'$, for all $BStms$ of object methods, where `self` is a fresh variable of the same type A of the class the method belongs to. The environment τ is consequently

extended to support a on-off mechanism which says when $\mathcal{E}[\![]\!]_{\rho,\tau}$ has to replace the self reference `this` with the reference to the object bound to variable `self`. In particular, in entering a method, τ''' is extended with a binding of a fictitious identifier `this` with the pair (`self`,*off*) where `self` is the name of 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 $\mathcal{E}[\![]\!]_{\rho,\tau}$ traverses a closure updating τ^{iv} to set the flag to *on*, thus making the binding for `this` active in the rule for *exp* that is added from variant (c).

5 Examples of closures.

We see how the transformation works through an example Fig. 2 taken from [Tro08]. In the program four closures different for behaviour and use are defined. The first and the second closure from top show the interaction, through the shared variable `x`, between the execution of the code in which the closure is invoked, and the execution of the code in which it is declared. In the third case the closure is passed as parameter to a method of the right type. Last closure shows the access to a field in the scope in which the closure is declared.

The preprocessor result is shown in Fig. 3.

```
public class ProgramA {
  static int z = 13;
  static void doTwice({:void} block){
    block.invoke(); // print 7
    int y =20;
    block.invoke(); // print 8
    System.out.printl(y); // print 20
    }
  public static void main(String[] args){
    shared int x = 4;
    {:void} printX = {:void ⇒ System.out.println(x);};
    x++;
    printX.invoke(); // stampa 5
    x = 100;
    {int:int} addX = {int y :int ⇒x += y; return ++x;};
    System.out.println(addX.invoke(50)); // print 151
    System.out.println(addX.invoke(0)); // print 152
    x = 200;
    System.out.println(addX.invoke(A.z)); // print 214
    // a different use: the block is packed with non local y and passed as a parameter
    shared int y = 7;
    doTwice({:void ⇒ System.out.println(y++);});
    {:void ⇒ z++; System.out.println(z);}.invoke(); // print 14
  }}
```

Fig. 2. ProgramA: Closures, with non locals, assigned to variables, passed as parameter, and invoked [Tro08]

```
public class A {
  static int z = 13;
  static void doTwice(Apply$Void block){
    block.invoke(); // print 7
    int y =20;
    block.invoke(); // print 8
    System.out.printl(y); // print 20
    }
  public static void main(String[] args){
    final Shared$Int x = new Shared$Int(4);
    Apply$Void printX = new Apply$Void(){public void invoke(){
        System.out.println(x.value);}};
    x.value++;
    printX.invoke(); // print 5
    x.value = 100;
    Apply$Int$Int addX = new Apply$Int$Int(){public int invoke(int y){
        x.value +=y; return ++x.value;}};
    System.out.println(addX.invoke(50)); // print 151
    System.out.println(addX.invoke(0)); // print 152
    x.value = 200;
    System.out.println(addX.invoke(A.z)); // stampa 214
    // a different use: the block is packed with non local y and passed as a parameter
    final Shared$Int y = new Shared$Int(7);
    doTwice(new Apply$Void(){public void invoke(){System.out.println(y.value++);}});
    new Apply$Void(){public void invoke(){z++; System.out.println(z);}}.invoke();
  }}// print 14
  // interfaces and classes generated from the transformation
  class Shared$Int{int value; Shared$Int(int n){value=n;};}
    // it allocates a variable on the heap
  interface Apply$Void{public void invoke();}
  interface Apply$Int$Int{public int invoke(int x);}
```

Fig. 3. $\mathcal{E}[\![\text{ProgramA}]\!]_{\rho,\tau}$: The result of preprocessing ProgramA

6 Use of `this` and other special cases.

Fig. 4 contains a definition of method `mksum` taken from [Tro08]. The method declares two local variables, n and s, and returns a closure as the computed value. The computed closure uses variables `n` and `s` hence, when the closure is invoked, as it happens in the three last statements of the main method, the activation frame [LY96] of the execution of `mksum` is no more active, as well as it should be for its local variables `n` and `s` if they were not declared `Shared`. Fig. 5 shows the Java code that is generated applying $\mathcal{E}[\![]\!]_{\rho,\tau}$ to program in Fig. 4.

```
public class C{
  static {:int} mksum(){ // it computes a closure
    shared int n = 1, s = 0;
    return {:int ⇒ s += n; n++; s;} // the activation record of
    } // mksum is no more active
  public static void main(String[] args){
    { :int} sum = mksum();
    System.out.println(sum.invoke()); // print 1
    System.out.println(sum.invoke()); // print 3
    System.out.println(sum.invoke()); // print 6
    }}
```

Fig. 4. `mksum`: Closure with locals of a code whose activation record is no more active

```
public class C{
  static {:int} mksum(){ // it computes a closure
    final shared$Int n = new Shared$Int(1), s = new Shared$Int(0);
    return new Apply$Int(){public int invoke(){s.value += n.value;
             n.value++; return s.value;}}
    } // the activation record of mksum is no more active
  public static void main(String[] args){
    Apply$Int sum = mksum();
    System.out.println(sum.invoke()); // print 1
    System.out.println(sum.invoke()); // print 3
    System.out.println(sum.invoke()); // print 6
    }}// interfaces and classes generated
  class Shared$Int{int value; Shared$Int(int n){value=n;};}
    // it allocates a variable on the heap
  interface Apply$Int{public int invoke();}
```

Fig. 5. $\mathcal{E}[\![\texttt{mksum}]\!]_{\rho,\tau}$: The result of preprocessing `mksum`

A completely different situation arises from the program in Fig. 6. In this case we would like to use a local variable, namely `fact`, defined in the scope in which the closure is defined, namely the scope of the `main` method, to support a mechanism for recursively

defined functions, i.e. the factorial function. Variable `fact` is declared `final` in the the program, see Fig. 6, since it is used in the closure. The Java code produced according to $\mathcal{E}[\![]\!]_{\rho,\tau}$ is in Fig. 7-a. and it generates a static error: The Java compiler rejects the program and signals "variable `fact` might not have been initialized". The compiler forbids the use in the expression initializing a final variable, of the variable itself. In effect, this use makes sense only in presence of functional abstractions and closures as it is when we extend Java with closures. However, the self reference of OO language already supports a mechanism for recursive definitions. This is accomplished by variant (b) of Fig. 1, which produces the program in Fig. 7-b.

```
public class Fact{
  public static void main(String[] args){
    final {int:int} fact = {int n :int ⇒
                  if (n==0) return 1;
                  return (n * fact.invoke(n-1); }
    }}
```

Fig. 6. Fact: Closure using recursive definitions

```
public class Fact{
  public static void main(String[] args){
    final Apply$Int$Int fact = new Apply$Int$Int(){public int invoke(int n){
              if (n==0) return 1;
              return (n * fact.invoke(n); } // reference to fact is replaced by this
    }}// interfaces and classes generated
  interface Apply$Int$Int{public int invoke(int x);}
```

a: The code resulting from preprocessing Fact with $\mathcal{E}[\![]\!]_{\rho,\tau}$

```
public class Fact{
  public static void main(String[] args){
    final Apply$Int$Int fact = new Apply$Int$Int(){public int invoke(int n){
              if (n==0) return 1;
              return (n * this.invoke(n); } // reference to fact is replaced by this
    }}// interfaces and classes generated
  interface Apply$Int$Int{public int invoke(int x);}
```

b: The resulting of preprocessing Fact with variation (b) of $\mathcal{E}[\![]\!]_{\rho,\tau}$

Fig. 7. $\mathcal{E}[\![\texttt{Fact}]\!]_{\rho,\tau}$: Use of `this` in the preprocessing of Fact

A last situation is exemplified by program in Fig. 8. Again the example is taken from [Tro08]. In this case we have a class `D` with an object member variable `s` of type `String` and an object method `test`. The method declares a local variable `s` of type `String` and invokes a closure which prints the string bound to the object member variable `s`. The Java code produced according to the variant (c) of $\mathcal{E}[\![]\!]_{\rho,\tau}$ is in Fig. 9. As we can see the

replacement of `this`, in the closure, with `self.value` allows the access to the object on which `test` is invoked.

```
public class D{
  String s = "hello";
  void test(){
    String s = "hi";
    { :void ⇒ System.out.printl(this.s);}.invoke();
    } // print "hello"
  public static void main(String[] args){
    new D().test();
    }}
```

Fig. 8. Test: A closure which uses `this`

```
public class D{
  String s = "hello";
  void test(){
    final shared$D self = new shared$D(this);
    String s = "hi";
    new Apply$Void(){public void invoke()
                      {System.out.println(self.value.s);}}.invoke();
    } // print "hello"
  public static void main(String[] args){
    new D().test();
    }}// interfaces and classes generated
  class Shared$D{D value; Shared$D(D n){value=n;};}
  interface Apply$Void{public void invoke();}
```

Fig. 9. $\mathcal{E}[\![\texttt{Test}]\!]_{\rho,\tau}$: The result of preprocessing Test

7 $\mathcal{E}[\![]\!]_{\rho,\tau}$: From Java 1.4 to Java 1.6

The transformation $\mathcal{E}[\![]\!]_{\rho,\tau}$, as defined so far, applies only to non generic Java programs. In fact, its definition relies on the language type structure as we can see noting that in Fig. 1, $\mathcal{E}[\![]\!]_{\rho,\tau}$ requires an extension of the structure of types, namely $FType$, and, more relevant, the use of two injective mappings $\mathcal{L}$ for class identifiers and $\mathcal{N}$ for interface identifiers. In particular, the mappings collect the types of the shared variables and the types of the closures and recognize distinct types. For mapping $\mathcal{N}$ a closure type is a tuple of types. In the non generic type system of Java two tuples of types are the same type if and only if they have the same syntax (see section 4.3.4 [GJSB00]). On the contrary, when types variables are considered, the problem of type subsumption must taken into account. That is, a type t_1 can be obtained by another type t_2 through a

binding of the type variables occurring in t_2. As a matter of fact, consider $A < T >$ and $A < Long >$ that are two Java types for a parametric class A. To adapt $\mathcal{E}[\![]\!]_{\rho,\tau}$ to work with generic Java programs, we need to extend $\mathcal{N}$ into a function that, given a (tuple of) type(s), checks the list of the generated interfaces for one whose type is an instance. In the Fibonacci serie example, proposed in [Dup08], the new function $\mathcal{N}$ must recognizes that the type $Long\$Long\$Long$ is an instance of the type $T\$T\$T < T >$. Provided such modification, the application of $\mathcal{E}[\![]\!]_{\rho,\tau}$ to the transformation of the code LinearFib in Fig. 10, follows $\mathcal{E}[\![]\!]_{\rho,\tau}$ provided for non generic programs.

```
class Stream<T>{ /* this class should form an imported API */
  private T head;
  private {:Stream<T>} lazyTail;
  public Stream(T head, {:Stream<T>} lazyTail){
    this.head = head;
    this.lazyTail = lazyTail;
    }
  public static <T> Stream<T> mkStream(shared T x1, shared T x2, shared {T,T:T} opt){
    return new Stream<T>(x2,{:Stream<T> ⇒ mkStream(x2,opt.invoke(x1,x2), opt)});
    }
  public void show(){
    System.out.print(head+" ");
    try{Thread.sleep(100);} catch(Exception e){/* ignore */}
    lazyTail.invoke().show();
    }
  }
public class S{ /* linear fibonacci serie */
  public static void main(String[] args){
    Long zero = Long.valueOf(0);
    Long one = Long.valueOf(1);
    {Long,Long:Long} plus = {Long x, Long y :Long ⇒ System.out.print(". ");
                             return x+y;};
    Stream.<Long>mkStream(zero,one,plus).show();}
  }
```

Fig. 10. LinearFib: A generic program for linear lazy Fibonacci numbers using closures

8 Conclusions

In the direction of extending Java with higher order mechanisms [OW97,Mei01,McC01] [Set03,Cor04,Mic04,BO05,Bri05,GHJV05,SS07], the paper discussed an extension of Java with *closures*. It revisited aims and characteristics of current similar proposals [BGGvdA06], [BLB06,CS06b]. All proposals are equipped with a (more or less complete) running prototype but a clear semantics is lacking. Using techniques we exploited in extending Java with methods as parameters [BO08c,BO08a], the paper introduced a formal definition and a prototype of Java with closures. The formal definition, $\mathcal{E}[\![]\!]_{\rho,\tau}$, consists in a set of source to source translation rules that state the meaning of (1) *closure types*, (2) *closures*, (3) *closure invocations* and (4) *shared variables*, in terms of

```
class Stream<T>{ /* this class should form an imported API */
  private T head;
  private StreamT <T> lazyTail;
  public Stream(T head, StreamT<T> lazyTail){
    this.head = head;
    this.lazyTail = lazyTail;
    }
  public static <T> Stream<T> mkStream(T x1, T x2, T$T$T<T> opt){
    final SharedT<T> par$x1 = new SharedT<T>(x1);
    final SharedT<T> par$x2 = new SharedT<T>(x2);
    final SharedT$T$T<T> par$opt = new SharedT$T$T<T>(opt);
    return new Stream<T>(par$x2.value,
        new StreamT<T> () {public Stream<T> invoke(){
                            return mkStream(par$x2.value,
                                 par$opt.value.invoke(par$x1.value,par$x2.value),
                                 par$opt.value)
                            }});
    }
  public void show(){
    System.out.print(head+" ");
    try{Thread.sleep(100);} catch(Exception e){/* ignore */}
    lazyTail.invoke().show();
    }
  }
public class S{ /* linear fibonacci serie */
  public static void main(String[] args){
    Long zero = Long.valueOf(0);
    Long one = Long.valueOf(1);
    T$T$T<Long> plus = new T$T$T<Long>(){public Long invoke(Long x, Long y){
                            System.out.print(". ");
                            return x+y;
                            }}
    Stream.<Long>mkStream(zero,one,plus).show();}
  }
  // interfaces and classes generated
  interface StreamT<T>{public Stream<T> invoke();}
  interface T$T$T<T>{public T invoke(T x1, T x2);}
  class SharedT<T>{T value; SharedT(T n){value=n;};}
  class SharedT$T$T<T>{T$T$T<T> value; SharedT$T$T(T$T$T<T> n){value=n;};}
```

Fig. 11. $\mathcal{E}[\![\texttt{LinearFib}]\!]_{\rho,\tau}$: the result of preprocessing LinearFib

compositions of well known ordinary Java structures. In particular, non-local variables in a closure, are syntactically `shared` variables and semantically bindings that are allocated in the heap. Hence, non-locals variables are transformed in references to objects of classes, generated at preprocessing time, having only one field for the current value, and one argument constructor for initialization. Based on $\mathcal{E}[\![]\!]_{\rho,\tau}$ parametric structure, two variants are defined to give semantics and define the corresponding transformation of recursively defined closures and closures accessing, through the identifier `this`, the object on which the method that defines the closure is invoked. Furthermore, we discussed how the parametric structure may allow to apply the transformation to Java 1.6. Examples of increasing difficulty showed on one hand closures expressivity and on the other hand the semantics associated through the code generated by $\mathcal{E}[\![]\!]_{\rho,\tau}$.

References

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

[BGGvdA06] G. Bracha, N. Gafter, J. Gosling, and P. von der Ahe. Closures for java, 2006. //blogs.sun.com/ahe/resource/closures.pdf.

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

[BLB06] D. Lea B. Lee and J. Bloch. Concise instance creation expressions: Closure without complexity, 2006. crazybob.org/2006/10/java-closure-spectrum.html.

[Blo07] J. Bloch. The closure controversy. In *JavaPolis07*. //www.parleys.com-/display/PARLEYS, 2007.

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

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

[BO08a] 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.

[BO08b] M. Bellia and M.E. Occhiuto. *JavaΩ: The Structures and the Implementation of a Preprocessor for Java with m_parameters*. University of Pisa, Dipartimento Informatica, 2008.

[BO08c] M. Bellia and M.E. Occhiuto. Methods as parameters: A preprocessing approach to higher order in java. *Fundamenta Informaticae*, 85(1):35–50, 2008.

[Bri05] B. Bringert. HOJ - higher-order Java, 2005. cs.chalmers.se/bringert/hoj.

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

[CS06a] C.Donnely and R. Stallman. Bison: The yacc-compatible parser generator, 2006. www.gnu.org/software/bison/manual.

[CS06b] S. Colebourne and S. Shulz. First_class methods: Java style closures, 2006. //docs.google.com /view?docid=ddhp95vd_6hg3qhc.

[CSC07] S. Colebourne, S. Shulz, and R. Clarkson. Java control abstraction - position paper, 2007. //docs.google.com /view?docid=ddhp95vd_8f8zkn3.

[CSC08] S. Colebourne, S. Shulz, and R. Clarkson. Fcm+jca, 2008. //docs.google.com /View?docid=ddhp95vd_0f7mcns.

[Dup08] L. Duponcheel. Closures: Linear lazy fibonacci numbers, 2008. September 17, //lucdup.blogspot.com/2008/09/closures-linear-lazy-fibonacci-numbers.html.

[Gaf07] N.M. Gafter. Jsr proposal: Closures for java, 2007. JavaCommunity Process, www.javac.info/consensus-closure-jsr.html.

[Gaf08] N.M. Gafter. Java closures prototype feature-complete, 2008. //gafter.blogspot.com/2008/08/java-closures-prototype-feature.html.

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

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

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

[Goe07] 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.

[Jon87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Lan66] P.J. Landin. A $\lambda-$calculus approach. In *Andvances in Programming and Non-numerical Computation, Ed. L. Fox*, pages 97–141. Pergamon Press, 1966.

[Lee06] B. Lee. The java closure spectrum, 2006. crazybob.org/2006/10/java-closure-spectrum.html.

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

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series, Addison Wesley, 1996.

[MAD^{+}62] J. McCarthy, P.W. Abrahams, D.J.Edwards, T.P. Hart, and M.I. Levine. *LISP 1.5 Programming Manual*. M.I.T., 1962. also in, www.softwarepreservation.org/projects/LISP.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part 1. *Comm. A.C.M.*, 3:184–195, 1960.

[McC01] G. McCluskey. Using method pointers and abstract classes vs interfaces. *Electronic Notes TCS*, 2001. //java.sun.com/developer/JDCTechTips-/2001/tt1106.html.

[Mei01] E. Meijer. Lambada, Haskell as a better Java. *Electronic Notes TCS*, 41(1), 2001.

[Mic04] Sun Microsystems. About microsoft's delegates, 2004. java.sun.com-/docs/white/delegates.html.

[OW97] M. Odersky and P. Wadler. Pizza into Java: translating theory into practice. In *Proc. 24th Symposium on Principles of Programming Languages*, pages 146–159, 1997.

[Pie02] B.J. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PZ00] T.W. Pratt and M.V. Zelkowitz. *Programming Languages: Design and Implementation*. Prentice-Hall, 2000.

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

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

[Tro08] Z. Tronicek08. Java closures tutorial, 2008. //gafter.blogspot.com/2008/08/java-closures-prototype-feature.html.