# A Reduction Semantics for Java

part 1: the object oriented kernel

Stelvio Cimato Cosimo Laneve Paolo Ciancarini
*Dipartimento di Scienze dell'Informazione, Bologna, Italy*

October 10, 1998

**Abstract**

We define the operational semantics of the object-oriented kernel of Java. In particular we describe classes, class loading, inheritance, hiding and overriding of attributes, objects, method invocations.

The operational style we have chosen is the so-called "reduction semantics". This because, we think, it is more intelligible with respect to other styles, such as structured operational semantics, and because it offers flexibility for dealing with the issues of concurrency and distribution, which will be the contents of the forthcoming part 2.

Our semantics can be used both as a formal specification of a Java run-time system and for static and dynamic verification of Java programs.

## 1 Introduction

Since the first release of Java, a number of flaws have been discovered compromising the security of the Java programming environment. These flaws are due to implementation errors, to misleading interpretations of Java specification or even to undervalued issues and weaknesses introduced in the design of both the Java language and the byte-code. A taxonomy of bugs and errors in Java has been presented in [7]. Even if many of the detected flaws have been patched in the successive releases of the language, new bugs are found while the use of Java spreads among software developers. The Sun site (`http://java.sun.com`) maintains a continuously updated list of the previous, currently known and still unresolved errors signaled by Java users. (Visitors of the site can even vote for the bug they would like to be first resolved in the next release of the Java Development Kit!).

A source of errors must be found in the lack of any formal methodology both in the design and implementation of Java framework. The reference manuals, which are the only official description of Java language, are ambiguous or inconsistent, if submitted to a rigorous and detailed formal analysis [3]. This is not so singular, since Java incorporates many novel features (class loading, dynamic linking, applets, etc.) which are not easy to understand and are indeed current subject of research in object oriented theory. Without a strong theoretical foundation, claims about security of the Java system cannot be definitive and can be continuously revised, as long as new issues or bugs are found. In fact, despite a large amount of work [8, 9, 17, 16, 18], debate on safety of Java type system continues both on newsgroups (see [11]) and related conferences.

We think that a first step for the analysis of Java programs is the definition of its formal semantics. It is well known that formal semantics dispense with ambiguities in the informal description or misinterpretations of the reader. Our aim is to model, in a comprehensible way, the basic mechanisms which underlie Java programs execution. To this purpose we commit to a simple technique for formalizing the semantics, the *reduction style*. This technique has been profitably used for defining the semantics both of functional languages (see Barendregt's book [1], for instance) and concurrent processes [5]. In particular, in concurrency theory, reduction semantics has been refined into *chemical semantics* [2], where features as parallelism and distribution are modelled in a simple way. These latter issues will be central in the companion paper (the part 2).

In this paper we focus on the object oriented kernel of Java. We define inheritance, subtyping, overriding and overloading, the dynamic class loading policy, object creation and method invocations. We voluntarily restrict here to the object oriented kernel because Java is already innovative in this subset. The detailed formalization of the interrelationships between the object oriented features is necessary to achieve a complete comprehension of Java programming environment, finding out its potential weaknesses.

The paper is organized as follows. In section 2 we present the syntax of the subset of Java. Basic notations, such as those to manage memories and environments are collected in section 3. In section 4 we define the semantics of standard features of object oriented programming, while in section 5 we discuss the Java type system and the class loading process. In section 6 we introduce a sample set of axioms which allow reasoning about Java programs. An overview of the results in the literature and a comparison with them can be found in section 7. Section 8 illustrates our next and long-term research.

## 2 The object oriented kernel of Java

The basic operations we allow in the restricted language are: object instance creation, method invocation and field selection and updating. Let $Id$ be a countable set of *identifiers* ($\mathtt{this}, \mathtt{super} \notin Id$) and $\mathsf{Val}$ be a set of values containing integer,reals and booleans. Statements, commands, expressions and qualified identifiers in Core-Java are generated by the following grammar:

$$
\begin{array}{rcl}
Stm & ::= & \varepsilon \mid \mathtt{return} \mid \mathtt{return}\,(Exp) \mid Com\ ;\ Stm \\
 & & \mid Type\ Id\ ;\ Stm \mid \{Stm\}; Stm \\
Com & ::= & \mathtt{new}\ Id() \mid QId\,(Exp) \mid QId\ =\ Exp \\
Exp & ::= & QId \mid Com \mid \mathsf{Val} \\
QId & ::= & Id \mid Id.Id \mid \mathtt{this}[.Id] \mid \mathtt{super}[.Id] \\
Id & ::= & string
\end{array}
$$

We briefly describe the above instructions; see [10] for a detailed description. The sequence of execution of a Java program is controlled by *statements* which are executed for their effect and do not have values. Statements can be collected in blocks represented by braces. The *empty statement* does nothing; then body of methods or blocks can be empty.

A `return` statement returns the control to the caller of a method or a constructor. Any variable declaration *Type Id* introduces one (local) variable name *Id*, whose type is *Type*. Some expressions, such as assignment, class instance creation and method invocation may be used as statements if followed by a semicolon; in this case, once executed, the returned value if any, is discarded.

The operation `new C()` is used to create new objects of class `C`. A method invocation expression with form `a.m(e)`, is used to invoke an instance method (we do not deal with class methods, i.e. methods declared `static` which are shared by all the instances of a class). Finally assignment expressions are used to change values of variables or fields provided that the type of the variable and the value of the expression on the right hand are compatible (see Section 3). The keyword `this` denotes a reference to the object for which the instance method was invoked or to the object being constructed; `super` acts as a reference to the current object as instance of its superclass. They can be used only in the body of a method.

## 2.1 Class declarations

CoreJava programs are defined by the grammar below:

$$
\begin{array}{rcl}
\mathit{Dec} & ::= & \varepsilon \\
 & & \mid\ \texttt{class}\ \mathit{Id}\ \{\mathit{Body}\}\ \mathit{Dec} \\
 & & \mid\ \texttt{class}\ \mathit{Id}\ \texttt{extends}\ \mathit{Id}\ \{\mathit{Body}\}\ \mathit{Dec} \\
\mathit{Body} & ::= & \varepsilon \\
 & & \mid\ \mathit{MType}\ \mathit{Id}\ (\mathit{Flist})\ \{\mathit{Stm}\}\ \mathit{Body} \\
 & & \mid\ \mathit{Type}\ \mathit{Id}\ ;\ \mathit{Body} \\
\mathit{Flist} & ::= & \varepsilon\ \mid\ (\mathit{Type}\ \mathit{Id}) \\
\mathit{MType} & ::= & \texttt{void}\ \mid\ \mathit{Type} \\
\mathit{Type} & ::= & \mathit{Primitive\text{-}Type}\ \mid\ \mathit{Reference\text{-}Type}
\end{array}
$$

Class declarations have two shapes, according to a class is a subclass of another one or not. The `extends` clause defines the direct superclass of the class being constructed; whenever the `extends` clause is missing, the class is assumed to implicitly extend the class `Object`.

The attributes of a class may be of two kinds: fields or methods. A field declarator introduces variables associated with the class and its objects, while a method contains executable code to change object's state. Each method is distinguished by its name, by the number and type of its parameters and the type of its return value which are specified in the method declaration. A `void` method is a method which does not return any value.

Primitive types are the Java built-in data types such as `int` or `float`; reference types are created after a class declaration and the values are references to objects of that class. We assume the presence of a special value $\texttt{null}_\texttt{T}$ for every type `T`. The meaning for this value is the default value for the type, as `0` for `int` type or `false` for `boolean` type or `null` for objects.

## 2.2 Semantics of declarations

Every class declaration is represented by a tuple, where the name of the class, its direct superclass and its body are stored. A variable declared in a field is identified by its name and is associated with its value. For simplicity, we disallow contemporary declaration and initialization of field variables. Therefore the initial value for a variable will be the `null` value for the declared type. The effect of a declaration clause is to bind the identifier to values of the declared type. For declarations involving reference types, the object creation is delayed until an explicit call to the constructor of the class is made.

Finally each method declaration is stored as a tuple composed by its name, number and type of identifiers and of return value; the tuple so obtained is associated with its body. The rules in table 1 can be extended to deal with an arbitrary number of parameters.

EXAMPLE 2.1 *We examine a simple class in Java which defines point objects and the resulting model after the application of the semantic rules:*

$$
\begin{array}{rcl}
\texttt{class}\ C\{B\} & \rightarrow & C : (\texttt{Object}, B) \\
\texttt{class}\ C\{B\}\ \texttt{extends}\ C' & \rightarrow & C : (C', B) \\
D\ D' & \rightarrow & D, D' \\
\mathit{type}\ I;\ B & \rightarrow & \{I : \texttt{null}_{type}\} \cup B \\
\mathit{mtype}\ I\ (\mathit{type}\ x)\ \{\mathit{Stm}\}\ B & \rightarrow & \{(I, x, \texttt{null}_{type}, \texttt{null}_{mtype}) : \mathit{Stm}\} \cup B
\end{array}
$$

Table 1: Shape of classes

```
class Point { int x; int y;
    void move (int dx, int dy) { this.x=dx; this.y=dy; return}}
{ Point:  ( Object, { x:  null_int, y:  null_int,
    (move, dx, null_int, dy, null_int, null_void):  { this.x=dx; this.y=dy; return}}}
```

# 3 Basic notation

Let $\mathsf{Ref}$ be the set of *reference values*, namely pointers to objects. References are tuples which store the function $\xi$ to retrieve the value of the associated fields, the declared type $C$, the runtime type $C'$ of the object and the loader $L$ used to get those classes. Indeed Java, as every object-oriented language, distinguishes between compile and runtime types in order to retrieve method codes. Namely,

$$\mathsf{Ref} \subseteq ((\mathit{QId} \rightharpoonup_f \mathcal{N}) \times \mathit{Id} \times \mathit{Id} \times \mathit{Id})$$

where $\rightharpoonup_f$ denotes a partial function and $\mathbb{N}$ are natural numbers. Therefore the shape of a reference value will be $(\xi, C, C', L)$. We also denote the domain of a function $f$ with $\mathsf{dom}(f)$. For a reference value $v$ we use the notation $v.fun$, $v.dtype$, $v.rtype$ and $v.loader$ to access the referenced object, its declared and runtime type and its loader, respectively.

Let $<:$ be the *subclass* relationship which is the transitive closure of the *direct subclass* relationship: a class is the *direct subclass* of the class it extends. Two primitive values are *type compatible* if they have the same type. The reference $(\xi, C, C', L)$ is type compatible with $(\zeta, D, D', L')$ if $C <: D$ and $L = L'$. The auxiliary function $\mathbb{T}$ returns the type of its argument.

DEFINITION 3.1 *Memories are elements of* $\mathbb{N} \rightharpoonup_f \mathsf{Val} \cup \mathsf{Ref}$ *and are ranged over by* $\mathcal{M}, \mathcal{M}', \ldots$ *The operations of new, access, increment and update are defined in the table below.*

**Auxiliary function:** *Memory*

- **new**: $\mathsf{fresh}(\mathcal{M})$ gives a fresh location;
- **access**: $\mathcal{M}(\ell)$ gives the value of the location $\ell$
- **increment**: $\mathcal{M}{+}\ell \mapsto v$ is defined provided that $\ell \notin \mathsf{dom}(\mathcal{M})$. In this case $\mathcal{M}{+}\ell \mapsto v$ is the function:

$$(\mathcal{M} + \ell \mapsto v)(\ell') \quad = \quad \begin{cases} v & \text{if } \ell = \ell' \\ \mathcal{M}(\ell') & \text{otherwise} \end{cases}$$

- **update:** $\mathcal{M}[\ell \mapsto v]$ is defined, provided that $\ell \in \mathsf{dom}(\mathcal{M})$ and $\mathcal{M}(\ell)$ and $v$ have the same (primitive) type. In this case $\mathcal{M}[\ell \mapsto v]$ is the function defined as follows:

$$\mathcal{M}[\ell \mapsto v](\ell') \quad = \quad \begin{cases} v & \text{if } \ell = \ell' \\ \mathcal{M}(\ell') & \text{otherwise} \end{cases}$$

While the memory is the repository for values, the values associated with variables are retrieved by looking at the current environment where the computation takes place. Environments are structured in frames to reflect the nesting of blocks where identifiers are declared. Each frame is indexed with a $\diamond$ or with an identifier; the identifier corresponds to the name of the method which the block of statements being executed belongs to, while a $\diamond$ index represents an internal block of a method body. The reason for separating the two types of frames is the following: when an access to a global variable is found, it is possible to look for it by going back in the previous frames, provided all these frames are labelled with a $\diamond$. It is not possible to go beyond method frames (those labelled with an identifier) because methods haven't global variables .

DEFINITION 3.2 *An environment* $\Gamma$ *is a not-empty sequence of frames* $\gamma_{k_1} \circ \cdots \circ \gamma_{k_n}$*, that are elements of* $Id \rightharpoonup_f \mathbb{N}$ *The operations of access and increment of environments are defined in the following table.*

**Auxiliary function:** Environment

- **access**: $\Gamma(x)$ gives the value – the location – of the identifier $x$, relatively to the last definition retrieved in the frame;

$$\gamma_{k_1} \circ \cdots \circ \gamma_{k_n}(x) \quad = \quad \begin{cases} \gamma_{k_n}(x) & \text{if } x \in dom(\gamma_{k_n}) \\ \gamma_{k_1} \circ \cdots \circ \gamma_{k_{n-1}}(x) & \text{if } k_n = \diamond \\ undefined & \text{otherwise} \end{cases}$$

- **increment**: $\Gamma + x \mapsto \ell$ is defined provided that $x \notin \mathsf{dom}(\Gamma)$. In this case $\Gamma + x \mapsto \ell$ is the function:

$$(\Gamma + x \mapsto \ell)(y) \quad = \quad \begin{cases} \ell & \text{if } x = y \\ \Gamma(y) & \text{otherwise} \end{cases}$$

We assume that the domain of each frame contains two special identifiers, `curr`, which points to the current object, and `rval`, which contains the return type of the method in execution.

# 4 Semantics

The execution of Java programs is performed by a Java Virtual Machine (JVM) which executes the byte code corresponding to a Java source program. A JVM is typically started by a shell command like: `java myprogram` whose effect is to invoke the method `main` of the class `myprogram`. Since the JVM loads classes according to a call-by-need strategy, the above call results in discovering, at least at the very beginning, that `myprogram` is not loaded, or, better, that the virtual machine does not contain an internal binary representation for `myprogram`. To effectively retrieve `myprogram`, the machine invokes then a default class-loader that looks for the binary representation of a class. We defer a detailed treatment of classloading to section 5.2. For the while, let us assume that the JVM has these binary representations and let us analyze the initial class environment needed by the virtual machine to execute any Java program.

Starting from the binary representation of the class `myprogram` — the sequence of bytes contained in the `.class` file — the JVM builds a model for defining objects of the retrieved class. This model is constructed as an instance of the system class `Class` and gives a runtime representation of the type. This runtime representation consists of a loader, which is the loader used to retrieve the code, and the store representation of the class definition. To keep track of the classes which have been already loaded we store them into an internal symbol table, whose entries are triples: a class identifier, its type representation and its class loader.

## 4.1 The abstract model

Our abstract model of Java is a transition system. Transition systems are pairs consisting of a set of *configurations* and a relation over them, the *transition relation*.

Configurations are tuples consisting of a *class environment* $\mathbb{C}$ modelling the symbol table of the JVM, a *memory* $\mathcal{M}$, an *environment* $\Gamma$ and the statements to be evaluated. In notation:

$$\mathbb{C}, \mathcal{M}, \Gamma \vdash \Sigma\ .$$

The class environment $\mathbb{C}$ contains three kinds of elements:

1. all the *visible classes* — the class files which are in the local file system (or even in remote file systems). For instance, in the initial state, $\mathbb{C}$ has all the system classes (those in the standard library in the Java development kit). We encode the binary representation of a class using the rules we give in section 2.1. Moreover, we assume that every class name is indexed by the actual location of the file in the file system.

2. the *type representations* of classes. We denote the type representation by indexing the class name with the keyword *obj* and we model it as a pair consisting of the store representation of a class definition, as described in table 1, and a loader, which is the loader used to retrieve the code. A default loader is the *system class-loader*, $L_{sys}$ in the following, which retrieves the code and loads it.

3. the *entries of the symbol table* needed to resolve the symbolic references found during the execution of a Java program. For every entry, we define a tuple associating the identifier of the class, its class loader and the type representation.

Example 4.1 *The following are the class model and the class object for a class* `Point` *of example 2.1 located in the directory* `/user`*, whose entry in the system class-loader table is* $(Point, Point_{obj}, L_{sys})$*:*

$$\begin{aligned} Point_{/user} : &\quad \Big(Object,\ \{x : \mathtt{null}_{int},\ y : \mathtt{null}_{int},\ (move, \cdots) : \{\cdots\}\}\Big) \\ Point_{obj} = &\quad \Big((Object, \{x : \mathtt{null}_{int}, \cdots\}),\ Loader\Big)\ . \end{aligned}$$

Statements $\Sigma$ are structured in blocks, each one denoted by square brackets. Whenever a new block $\{\cdots\}$ is found, the sequence of square brackets is lenghtened with a new item (corrispondingly the frame sequence of the environment is lenghtened with a new empty frame). Every block has the shape $[\![P,\ S]\!]$, where the $P$-component is a store that keeps expressions which need further evaluation, while $S$ is a sequence of valid Java statements. Whenever an expression to be evaluated is encountered, we defer the application of the appropriate rule till a value is reached. In this case the statement execution is suspended, and a $\bullet$ is substituted for the expression in the store part of the rule. The computation proceeds in another block till a value is returned and the application of a transition rule becomes possible. The remaining part of the section deals with the definition of the transition relation. We shall introduce a set of rule schemata and the transition relation is defined as the least relation satisfying the rules.

## 4.2 Objects

There are two ways to create objects in Java: by `new C()` or by `T.newInstance()`, where `T` is a class object of a type. We will examine `newInstance` in section 5.

A new object is created by instantiating a block of fresh memory locations to store the object's fields. The operation `new` returns the reference to this block and completes by invoking the constructor method of the argument class. In turn, the execution of the constructor implicitly — managed by the compiler — invokes the constructor of the super-class as a first statement; till the constructor of the class `Object` is invoked, which terminates without effects. To uniform constructor calls with method calls, we assume that the first statement of every constructor is the call of the constructor of the super class.

Two auxiliary functions will be used to define objects. The first one, called $\mathsf{Inst}$, defines the tuple which must belong to the class environment $\mathbb{C}$; $C.a$ refers to the field $a$ in the class $C$. The second function, denoted $\mathsf{Mkfun}$, allocates memory for the value of the fields of the object.

**Auxiliary function:** Inst, Mkfun

Let $C_{obj} = (C : (C', B), L)$ in $\mathbb{C}$ :

$$\begin{array}{rcl} \mathsf{Inst}_{\mathbb{C}}(\varepsilon, L) & = & \emptyset \\ \mathsf{Inst}_{\mathbb{C}}(C, L) & = & \{C.a : val \mid a : val \in B_C\} \cup \mathsf{Inst}_{\mathbb{C}}(C', L) \\ \\ \mathsf{Mkfun}(\emptyset, f, \mathcal{M}) & = & (f, \mathcal{M}) \\ \mathsf{Mkfun}(\{a : val\} \cup A, f, \mathcal{M})) & = & Mkfun(A, f + a \mapsto (\ell)\, \mathcal{M} + \ell \mapsto val) \end{array}$$

where $\ell = \mathsf{fresh}(\mathcal{M})$.

The transition rule for the object creation is defined in Table 2. According to this rule, the computation progresses by invoking the constructor of the class in a new frame where the `curr` identifier is set to the new object. The reference to the new object is represented by a tuple containing the object function returned by $\mathsf{Mkfun}$, the declared and runtime type (which are set to the class for which the creation was invoked) and a reference to the current class loader. In the rule we assume that the class associated with the current class loader is already in $\mathbb{C}$. If this is not the case an invocation to the class-loader is needed to retrieve the class and to create the corresponding molecule in the $\mathbb{C}$, as we show in more detail in section 5.2.

$$\{(C, C_{obj}, L)\} \cup \mathbb{C}, \Gamma, \mathcal{M} \vdash [\![\ \emptyset,\ \mathtt{new}\ C()\ ;\ S\ ]\!]\Sigma \rightarrow$$
$$\{(C, C_{obj}, L)\} \cup \mathbb{C}, \Gamma \circ \gamma, \mathcal{M}'' \vdash [\![\ \emptyset,\ C()\ ;\ (\zeta, C, C, L)\ ]\!]\,[\![\ \emptyset,\ S\ ]\!]$$

where:
- $(\zeta, \mathcal{M}') = \mathsf{Mkfun}(\mathsf{Inst}_{\mathbb{C}}(C, L), \emptyset, \mathcal{M})$,
- $L = \mathcal{M}(\Gamma(\mathtt{curr})).loader$,
- $\mathcal{M}'' = \mathcal{M}' + [\gamma(\mathtt{curr}) \mapsto (\zeta, C, C, L)]$.

Table 2: Semantics of new

## 4.3 Declaration of local variables

Table 3 defines the semantics for the declarations of local variables. The scope of a local variable is set to the block within it is declared. Within the scope no re-definition is admitted.

| |
|---|
| $\mathbb{C}, \Gamma, \mathcal{M} \vdash [\![ \emptyset,\ ptype\ x\ ; S ]\!]\ \Sigma \to \mathbb{C}, \Gamma + x \mapsto \ell, \mathcal{M} + (\ell \mapsto \mathtt{null}_{ptype}) \vdash [\![ \emptyset, S ]\!]\ \Sigma$ |
| where - $\ell = \mathsf{fresh}\ (\mathcal{M})$, |
| $\{(C, C_{obj}, L)\} \cup \mathbb{C}, \Gamma, \mathcal{M} \vdash [\![ \emptyset,\ rtype\ x\ ; S ]\!]\ \Sigma \to$ $\{(C, C_{obj}, L)\} \cup \mathbb{C}, \Gamma + x \mapsto \ell, \mathcal{M}' \vdash [\![ \emptyset, S ]\!]\ \Sigma$ |
| where - $\mathcal{M}' = \mathcal{M} + (\ell \mapsto (\zeta,\ rtype,\ rtype,\ L))$, $L = \mathcal{M}(\Gamma(\mathtt{curr})).loader$ |

Table 3: Semantics of local variable declarations

To avoid confusion in retrieving the meaning of names, we distinguish between variables and fields, by making explicit field access through their qualified names. So a reference to $x$ returns the value of the variable $x$ (if any); to access the field $x$ of the current class, we must provide its qualified name, i.e. $C.x$ or $\mathtt{this}.x$. Moreover we do not allow contemporary declaration and initialization of local variables.

Declarations are resolved differently according to the type of the variable. If the variable has a primitive type, then a fresh memory location is allocated and the current environment is updated with the binding between the identifier and the location. The location will contain the default value for the primitive type. If the variable has a reference type — it contains a reference to an object — the associated location will contain the tuple which represents the object. In this case, the reference is resolved with respect to the current class-loader.

As for the object creation, we assume that the definition of the reference type, i.e. the class object associated with the current identifier, is already in the class environment, otherwise the class-loader in invoked to load the class (section 5.2)

## 4.4 Commands and Expressions

To define the semantics of statements we use an auxiliary function, called Fret, which takes the label of the field and retrieves it starting from the declared class of the object. The function Fret associates the referred label with the qualified label making the reverse work of the Inst function.

**Auxiliary function:** Fret

Let $C : (C', B) \in \mathbb{C}$. $\quad \mathsf{Fret}(C, x) \quad = \begin{cases} C.x & \text{if } x \in B \\ \mathsf{Fret}(C', x) & \text{otherwise} \end{cases}$

The semantics of statements and expressions is defined in Tables 4 and 5, respectively.

In the first rule of Table 4 dealing with field updating, we use the auxiliary function Fret for retrieving the field, starting from the declared class of the object. Assignment for local variables, is simply resolved by updating the location associated to the named identifier by the current environment $\Gamma$. Note that, for primitive types, a type checking is performed, since memory updating for values with different types is not defined.

Things are a little bit more complicated for assignment involving variables of reference types. Objects in memory are stored as tuple recording the function which maps fields to values as well as its declared and runtime types (with the class-loader used to retrieve the associated classes). Changing the link between an identifier and an object amounts to change the tuple in the referenced memory location. The object function is substituted with the one of the assigned object and the runtime type is updated. Remark that at runtime an object may have a type different from the declared one, due to the polymorphism implemented by Java. However it is

$$
\begin{array}{rcl}
\mathbb{C},\Gamma,\mathcal{M} \vdash [\ (\xi,C,C',L).x = k, S\ ]\ \Sigma & \rightarrow & \mathbb{C},\Gamma,\mathcal{M}[\xi(\mathsf{Fret}(C,x)) \mapsto k] \vdash [\ \emptyset,\ S\ ]\ \Sigma \\
\mathbb{C},\Gamma,\mathcal{M} \vdash [\ x = k,\ S]\ \Sigma & \rightarrow & \mathbb{C},\Gamma,\mathcal{M}[\Gamma(x) \mapsto k] \vdash [\ \emptyset,\ S]\ \Sigma \\
\mathbb{C},\Gamma,\mathcal{M} \vdash [\ \emptyset,\ \mu.x = e\ ;\ S\ ]\ \Sigma & \rightarrow & \mathbb{C},\Gamma \circ \{\}_\diamond,\mathcal{M} \vdash [\ \emptyset,\ e\ ]\ [\ \mu.x = \bullet, S\ ]\ \Sigma \\
\text{where } \text{-}\ \mu = (\xi,C,C',L) & & \\
\mathbb{C},\Gamma,\mathcal{M} \vdash [\ \emptyset,\ x = e\ ;\ S]\ \Sigma & \rightarrow & \mathbb{C},\Gamma \circ \{\}_\diamond,\mathcal{M} \vdash [\ \emptyset,\ e\ ]\ [\ x = \bullet\ , S]\ \Sigma \\
\mathbb{C},\Gamma,\mathcal{M} \vdash [\ x = (\xi,C,C',L),\ S]\ \Sigma & \rightarrow & \mathbb{C},\Gamma,\mathcal{M}[\Gamma(x) \mapsto o] \vdash [\ \emptyset,\ S]\ \Sigma \\
\multicolumn{3}{l}{\text{where } C <: \mathcal{M}(\Gamma(x)).\mathit{dtype},\quad o = (\xi,\mathcal{M}(\Gamma(x)).\mathit{dtype},C',\mathcal{M}(\Gamma(x)).\mathit{loader})} \\
\\
\mathbb{C},\Gamma \circ \gamma_m,\mathcal{M}+\mu \vdash [\emptyset,\mathtt{return};]\ \Sigma & \rightarrow & \mathbb{C},\Gamma,\mathcal{M}+\mu \vdash \Sigma \\
\text{where } \text{-}\ \mu = \gamma_m(\mathtt{rval}) \mapsto \mathtt{null}_{\mathtt{void}} & & \\
\mathbb{C},\Gamma \circ \gamma_m,\mathcal{M} \vdash [\ \mathtt{return}(k),\ S\ ]\ \Sigma & \rightarrow & \mathbb{C},\Gamma,\mathcal{M}[\gamma_m(\mathtt{rval}) \mapsto k] \vdash k\ \Sigma \\
\mathbb{C},\Gamma,\mathcal{M} \vdash [\ \emptyset,\mathtt{return}(e);S]\ \Sigma & \rightarrow & \mathbb{C},\Gamma \circ \{\}_\diamond,\mathcal{M} \vdash [\emptyset,e]\ [\mathtt{return}(\bullet),\mathtt{S}]\ \Sigma \\
\mathbb{C},\Gamma,\mathcal{M} \vdash [\emptyset,\{S'\};S]\ \Sigma & \rightarrow & \mathbb{C},\Gamma \circ \{\}_\diamond,\mathcal{M} \vdash [\emptyset,S']\ [\emptyset,S]\ \Sigma
\end{array}
$$

Table 4: Semantics of sequential commands

required that the type of the object must be a subtype of the declared type of the reference.

Return statements mark the end of a method body execution and transfer the control to the invoker of the method. They discard the current frame and test for the correct typing of the return value. This is accomplished by looking at the special identifier `rval` in the current frame. In case an expression is returned, then the value of the expression becomes the value of the method invocation which must be compatible with the declared return type of the method. If the `return` has no expression, then the method containing it must have been declared as `void`. 5

Braces group statements in blocks. Blocks are evaluated in a new empty frame; in this way local variables of a block will be no more valid at the end of the block since the last frame is discarded.

Rules in table 5 are of easy interpretation. Whenever a constant value is encoun-

$$
\begin{array}{rcl}
\mathbb{C},\Gamma \circ \gamma_\diamond,\mathcal{M} \vdash [\ \emptyset,\ k\ ]\ [\ e,S]\ \Sigma & \rightarrow & \mathbb{C},\Gamma,\mathcal{M} \vdash [\ e[k/\bullet],\ S\ ]\ \Sigma \\
\mathbb{C},\Gamma,\mathcal{M} \vdash [\ \emptyset,\ x\ ]\ \Sigma & \rightarrow & \mathbb{C},\Gamma,\mathcal{M} \vdash [\ \emptyset,\ \mathcal{M}(\Gamma(x))\ ]\ \Sigma \\
\mathbb{C},\Gamma,\mathcal{M} \vdash [\ k,\ \emptyset\ ]\ [\ e,S]\ \Sigma & \rightarrow & \mathbb{C},\Gamma,\mathcal{M} \vdash [\ e[k/\bullet],\ S\ ]\ \Sigma \\
\mathbb{C},\Gamma,\mathcal{M} \vdash [\ \emptyset,\ e.x\ ]\ \Sigma & \rightarrow & \mathbb{C},\Gamma \circ \{\}_\diamond,\mathcal{M} \vdash [\ \emptyset,\ e\ ]\ [\ \bullet.x,\ \emptyset\ ]\ \Sigma \\
\mathbb{C},\Gamma,\mathcal{M} \vdash [\ \emptyset,\ \mathtt{this}]\ \Sigma & \rightarrow & \mathbb{C},\Gamma,\mathcal{M} \vdash \mathcal{M}(\Gamma(\mathtt{curr}))\ \Sigma \\
\mathbb{C},\Gamma,\mathcal{M} \vdash [\ \emptyset,\ \mathtt{super}]\ \Sigma & \rightarrow & \mathbb{C},\Gamma,\mathcal{M} \vdash [\ \emptyset,\ (o.\mathit{fun},D,o.\mathit{rtype},o.\mathit{loader})]\ \Sigma \\
\multicolumn{3}{l}{\text{where } \text{-}\ o = \mathcal{M}(\Gamma(\mathtt{curr})),\quad (o.\mathit{dtype})_{obj} = ((D,B),\ o.\mathit{loader}) \in \mathbb{C},\quad o.\mathit{dtype} \neq \mathtt{Object}} \\
\multicolumn{3}{l}{\mathbb{C},\Gamma,\mathcal{M} \vdash [\ (\xi,C,C',L).x,\ \emptyset\ ]\ \Sigma \rightarrow \mathbb{C},\Gamma,\mathcal{M} \vdash [\ \mathcal{M}(\xi(\mathsf{Fret}(C,x))),\ \emptyset\ ]\ \Sigma}
\end{array}
$$

Table 5: Semantics of expressions

tered at the end of a block or returned in the stack part as result of an expression evaluation, it becomes available to the caller — the previous frame — and the frame used for its computation is discarded.

Variable access is resolved returning the content of the memory location associated with the named identifier in the current frame. When a reference to `this` is found, the value of the `curr` identifier is returned; when a reference to `super` is met, the returned value is a reference to the current object considered as an instance of its direct superclass. In this way fields overridden by another declaration in the current class, can be accessed.

Basic operations between primitive values are resolved by applying the operator and returning the resulting value. The evaluation of complex expressions is performed by evaluating arguments in left-to-right order till a simpler expression between primitive values is reached.

Field accesses are evaluated by retrieving the value in the memory location referenced by the object function. We use the function Fret to find the right location. As before, we require that fields are accessed through qualified identifier, i.e. explicitly naming the owner object or using the keyword `this` for fields belonging to the current class.

## 4.5 Method invocation

A method is invoked by providing its name and the target reference of the object to which the method belongs. The general form of the expression for method invocation is $\mathtt{e.m(k}_1\mathtt{,\ ...,\ k}_n\mathtt{)}$ where the expression `e` returns a valid object reference and $\mathtt{k}_i$ are the parameters. Each method has a *signature*, namely a fixed number of parameters of a specified compile type and a return type.

Resolving a method invocation expression is not easy for the possibility of method *overloading* and *overriding*. Methods belonging to a class may have the same name but different signatures; in this case we say that the method is *overloaded*. The declaration of a method in a class is said to *override* any other method with the same signature in the superclasses of that class. After the method is invoked, the signature is used to determine the method that will be actually invoked. At runtime, the method to be executed is determined using dynamic method lookup — *late binding* —, i.e. retrieving the method body starting from the class corresponding to the runtime type of the object from which the method was invoked.

We model method resolution by defining three auxiliary functions: Mreach, Spec and Retrieve. The first one retrieves all those methods in a class $C$ which are *applicable* to the invocation, namely those methods whose signature is compatible with the invocation. Spec considers method definitions pairwise and selects the *most specific* one, (§15.11.2.2 [10]). Finally, the Retrieve function starts the search of the selected method body in the class corresponding to the runtime type of the target reference, going upward the class hierarchy till a method with the same name and signature is found.

**Auxiliary function:** Mreach, Spec, Retrieve

Let $C_{obj} = ((C', B)), L) \in \mathbb{C}$ and $t$ be the type of $k$. Then

$$\begin{aligned}\mathsf{Mreach}(C, L, m, k) \;=\; & \mathsf{Mreach}(C', L, m, k) \;\bigcup \\ & \{(C, (m, x, \mathtt{null}_T, \mathtt{null}_R)) \mid (m, x, \mathtt{null}_T, \mathtt{null}_{Mt}) \in B \text{ and } t <: T\}\end{aligned}$$

Let $M_i = (m, x, \mathtt{null}_{T_i}, \mathtt{null}_{R_i})$. Then

$$\mathsf{Spec}(\{(C_i, M_i) \mid i \in \mathbb{N}\}) \;=\; M_j \quad \text{if } \forall i.\;\; C_j <: C_i,\; T_j <: Ti$$

Let $\mathbb{C}(C, L) = (D, B)$. Then:

$$\text{Retrieve}(M, C, L) = \begin{cases} (C, S) & \text{if } M : S \in B \\ \text{Retrieve}(M, D, L) & \text{otherwise} \end{cases}$$

We are now ready to define the method invocation. To explain the rules in table 6, suppose we invoke method `m` of an object of declared class type `C` and `C'` as runtime type. So the method invocation expression is $(\xi, C, C', L)$`.m(k)`. The right method body is returned from Mreach $\circ$ Spec. Once the appropriate method is found, a new frame $\gamma_m$ is created and the memory $\mathcal{M}$ is modified adding the locations which contain the return value, the parameters of the method (if any), and the values for the identifiers `curr` and `rval` . Remark that the value for the declared type of `curr` is the class which the method in execution belongs to (ref. §15.7.2). The created frame will be discarded at the end of the execution of the method body (we require each method ends with a `return` statement).

The special method invocation in which the keyword `super` is used, must be treated in an *ad hoc* rule, since the corresponding method is not choosen looking to the runtime type of the reference. Rather, the implementation of the method is searched in the superclass of the class whose declaration contains the method declaration.

$$\mathbb{C}, \Gamma, \mathcal{M} \vdash [\ (\zeta, C, C', L).m(k),\ S\ ]\ \Sigma \ \rightarrow\ \mathbb{C}, \Gamma \circ \gamma_m, \mathcal{M}' \vdash [\ \emptyset,\ S'\ ]\ [\ \emptyset,\ S\ ]\ \Sigma$$

where
- $\ell, \ell', \ell'' = \mathsf{fresh}\ (\mathcal{M})$,
- $\gamma_m = x \mapsto \ell + \mathtt{rval} \mapsto \ell' + \mathtt{curr} \mapsto \ell''$,
- $\mathcal{M}' = (\mathcal{M} + (\ell \mapsto \mathtt{null}_T) + (\ell' \mapsto \mathtt{null}_R) + \ell'' \mapsto (\zeta, C, C', L))[\ell \mapsto k]$,
- $M = \mathsf{Spec}(\mathsf{Mreach}(C, L, m, x))$,
- $(D, S') = \mathsf{Retrieve}(M, C', L)$,
- $\mathcal{M}'(\Gamma(\mathtt{curr})).dtype = D$

$$\mathbb{C}, \Gamma, \mathcal{M} \vdash [\ \emptyset, \mathtt{super}.m(k); S\ ]\ \Sigma \ \rightarrow\ \mathbb{C}, \Gamma \circ \gamma_m, \mathcal{M}' \vdash [\ \emptyset,\ S'\ ]\ [\ \emptyset,\ S\ ]\ \Sigma$$

where
- $\ell, \ell', \ell'' = \mathsf{fresh}\ (\mathcal{M})$,
- $(\xi, C, C', L) = \mathcal{M}(\Gamma(\mathtt{curr}))$,
- $C_{obj} = ((F, B), L) \in \mathbb{C},\ C \neq \mathtt{Object}$,
- $\gamma_m = x \mapsto \ell + \mathtt{rval} \mapsto \ell' + \mathtt{curr} \mapsto \ell''$,
- $\mathcal{M}' = (\mathcal{M} + (\ell \mapsto \mathtt{null}_T) + (\ell' \mapsto \mathtt{null}_R) + \ell'' \mapsto (\zeta, C, C', L))[\ell \mapsto k]$,
- $M = \mathsf{Spec}(\mathsf{Mreach}(F, L, m, x))$,
- $(D, S') = \mathsf{Retrieve}(M, F, L)$,
- $\mathcal{M}'(\Gamma(\mathtt{curr})).dtype = D$ .

Table 6: Semantics of method invocation

# 5 Advanced Features

There are two basic features which are still out of our semantics: accessing to the type system and reprogramming the class loading. We devote this section to cover these remaining aspects.

## 5.1 Access to the type system

Java types are internally represented by special objects in `Class` which record all the informations about the type of the object. `Class` objects are sometimes called *meta-classes* since they model classes (and interfaces) in a way that a running program can manipulate. More precisely, objects in `Class` can be used for queries like to ask for the number of methods the class contains, or to create new instances of a class. In this way the type system effectively becomes part of the program environment and can be navigated or exploited for debugging purposes. In addition `Class` objects play a crucial role in the mechanism for loading classes, as we will see in the next subsection.

To access to the information in class objects, Java makes available three methods of class `Class`: `Class getSuperclass()` which given a Class object, gives back the Class representing the superclass of that class; `Object newInstance()` which creates an object instance exactly as an object creation expression was used; `ClassLoader getClassLoader()` which gives back a reference to the loader used to load the class. In our formalism, `Class` objects belong to the class environment $\mathbb{C}$. Each `Class` object is represented by a pair carrying the body associated with the class identifier, and the classloader. Since `Class` cannot be extended (it is final), we can model directly the semantics of the above methods. Class objects may also be accessed through the final `getClass` pf class Object.

$$\mathbb{C}, \Gamma, \mathcal{M} \vdash [\ ((D, B), Loader).\mathtt{getSuperClass}(), S]\ \Sigma \ \rightarrow \ \mathbb{C}, \Gamma, \mathcal{M} \vdash [\ ((E, B_D), Loader),\ S\ ]\ \Sigma$$

where $-\ D \neq \mathtt{Object}$

$$\mathbb{C}, \Gamma, \mathcal{M} \vdash [\ ((D, B), Loader).\mathtt{newInstance}(), S]\ \Sigma \ \rightarrow \ \mathbb{C}, \Gamma, \mathcal{M} \vdash [\ \mathtt{new}C(), S\ ]\ \Sigma$$

$$\mathbb{C}, \Gamma, \mathcal{M} \vdash [\ ((D, B), Loader).\mathtt{getClassLoader}(), S]\ \Sigma \ \rightarrow \ \mathbb{C}, \Gamma, \mathcal{M} \vdash [\ ((E, B_{Loader}), Loader), S]\Sigma$$

$$\mathbb{C}, \Gamma, \mathcal{M} \vdash [\ (\xi, C, C', L).\mathtt{getClass}(), S\ ]\ \Sigma \ \rightarrow \ \mathbb{C}, \Gamma, \mathcal{M} \vdash [\ C'_{obj}, S\ ]\ \Sigma$$

Table 7: Rules for final methods in class `Class` and `Object`

## 5.2 Dynamic Class Loading

Java provides a powerful mechanism for downloading of code, the *dynamic loading of the classes.* To retrieve the byte-code associated to the class name the runtime system maintains associations between class objects and the classloader in a table — the *class table* —. Looking up the table gives back the class object associated to the reference and to the current class loader. If there is no available association for the named class, then the `loadClass` method of the current Classloader is invoked with the name of the searched class as argument. The default `loadClass` method just searches the file system according to an order defined by a "class path" file. Whenever the byte-code is found, it is loaded and a new entry of the class table is created.

Java allows to change this default mechanism by providing a `ClassLoader` object that look for byte-codes in some *ad-hoc* way and then loads it into the runtime. `ClassLoader` objects must be obtained as subclasses of the base abstract class provided by the standard Java library and must provide an implementation of the abstract `loadClass` method.

In our formalism classes, class objects, the class table and the class-path file, denoted *ClassPath*, are modelled in $\mathbb{C}$ such that the class loading mechanism is closely reflected by our semantics. In table 5.2 we define object creation and local variable semantics the invoked class object does not belong to the $\mathbb{C}$.

$\mathbb{C}, \Gamma, \mathcal{M} \vdash [\, \emptyset,\ \texttt{new}\ C()\ ;\ S\,]\Sigma \rightarrow \mathbb{C}, \Gamma, \mathcal{M} \vdash [\, \emptyset,\ S'] \,[\, \emptyset,\ \texttt{new}\ C()\ ;\ S\,]\Sigma$
where - $(C, C_{obj}, \mathcal{M}(\Gamma(\texttt{curr})).loader) \notin \mathbb{C}$;
- $S' = (\mathcal{M}(\Gamma(\texttt{curr})).loader).\texttt{loadClass}(C, \texttt{true})$.

$\mathbb{C}, \Gamma, \mathcal{M} \vdash [\, \emptyset,\ rtype\ x\ ; S]\ \Sigma \rightarrow \mathbb{C}, \Gamma, \mathcal{M} \vdash [\, \emptyset,\ S']\ [\, \emptyset,\ rtype\ x\ ; S]\ \Sigma$
where - $(C, C_{obj}, \mathcal{M}(\Gamma(\texttt{curr})).loader) \notin \mathbb{C}$;
- $S' = (\mathcal{M}(\Gamma(\texttt{curr})).loader).\texttt{loadClass}(C, \texttt{true})$.

$\mathbb{C}, \Gamma, \mathcal{M} \vdash [\, \emptyset,\ L_{sys}.\texttt{loadClass}(C, \texttt{true})] \ \rightarrow \mathbb{C} \cup (C, C_{obj}, L_{sys}), \Gamma, \mathcal{M} \vdash [\, \emptyset,\ C_{obj}]$
where - $(C, C_{obj}, L_{sys}) \notin \mathbb{C}$, - $ClassPath = t_1 \cdot \ldots \cdot t_n$
- $C_{t_i} \in \mathbb{C}\ \wedge C_{t_j} \notin \mathbb{C}\ \forall\ j < i$, - $C_{obj} = (C_{t_i}, L_{sys})$

$\mathbb{C}, \Gamma, \mathcal{M} \vdash [\, (\xi, Dl, Dl, L).\texttt{defineClass}(\texttt{name}, \texttt{byte}, \texttt{a}, \texttt{b}), S]\ \Sigma\ \rightarrow$
$\mathbb{C} \cup \{(name_{\texttt{byte}}, Dl)\}, \Gamma, \mathcal{M} \vdash [\, \emptyset, name_{obj};\ S\,]\ \Sigma$
where - $Dl <: \texttt{ClassLoader}$

$\mathbb{C}, \Gamma, \mathcal{M} \vdash [\, (\xi, Dl, Dl, L).\texttt{findSystemClass}(\texttt{name}), S]\ \Sigma\ \rightarrow$
$\mathbb{C}, \Gamma, \mathcal{M} \vdash [\, \emptyset, Lsys.\texttt{loadClass}(\texttt{name}, \texttt{true}); S\,]\ \Sigma$
where - $Dl <: \texttt{ClassLoader}$

Table 8: Dynamic class loading

Observe that the class loading process is triggered as soon as a class object is not found in $\mathbb{C}$. In this case the reference to a nonexistent class object is preceded by an invocation of the `loadClass` method of the current class-loader. The default implementation of this abstract method (described in table 5.2) retrieves the class following the *ClassPath* file. Afterwards, the standard evaluation of `new` (see table 2) is resumed. A user-defined `loadClass` method typically invokes one of the two other methods, `findSystemClass` or `defineClass` in the `ClassLoader` class. The method `findSystemClass` takes a name of class and gives a class object with that name; the method `defineClass` takes an array of bytes — a *bytecode* — and a name and creates a class object with that name whose body is the bytecode. The rules defining `defineClass` and `findSystemClass` are in table 5.2.

# 6 Reasoning about programs: a sample set of axioms

Having a formal definition of the semantics of Java, we can use powerful means to reason about program properties. In particular, we can exploit our semantics to establish the equivalence of program phrases. The direct way consists in proving that two programs, beginning in a same state terminate in a same state. An alternative indirect way is to single out a set of axioms and infer equivalences by means of them. This approach is surely more appealing since it allows to abstract away from the semantics. The unique role played by the semantics is to show the consistency of the axioms.

Let us present a sample set of axioms and let us show their consistency. Foremost we introduce the notion of equivalence.

Definition 6.1 *Let $S$ and $S'$ be two statements in CoreJava. $S$ and $S'$ are equivalent, in notation $S = S'$, if, for every $\mathbb{C}, \Gamma, \mathcal{M}$ defining free variables in $S$ and $S'$:*

$$\mathbb{C}, \Gamma, \mathcal{M} \vdash S \rightarrow^* \mathbb{C}', \Gamma', \mathcal{M}' \vdash \epsilon \quad \Longleftrightarrow \quad \mathbb{C}, \Gamma, \mathcal{M} \vdash S' \rightarrow^* \mathbb{C}', \Gamma', \mathcal{M}' \vdash \epsilon$$

We list few axioms in table 9; they can be used to infer the semantic equivalence between phrases of our language. We assume that $\mathtt{e}$ and $\mathtt{e}'$ are *simple* expressions, i.e. they do not contain method invocations or update fields. Furthermore, we restrict the usage of axioms to well typed programs.

$$\frac{\mathtt{x} \neq \mathtt{y} \qquad \mathtt{x} \notin \mathsf{fv}(\mathtt{e}') \qquad \mathtt{y} \notin \mathsf{fv}(\mathtt{e})}{(\mathtt{x=e;\ y=e'}) = (\mathtt{y=e';\ x=e})}\ \mathtt{e}, \mathtt{e}' \text{ simple} \qquad \frac{\mathtt{S}_1 = \mathtt{S}_1' \quad \mathtt{S}_2 = \mathtt{S}_2'}{\mathtt{S}_1; \mathtt{S}_2 = \mathtt{S}_1'; \mathtt{S}_2'}$$

$$\frac{\mathtt{S}\rho = \mathtt{S}'}{\mathtt{m}(\mathtt{x}_1, \ldots, \mathtt{x}_n)\{\mathtt{S}\} = \mathtt{m}'(\rho(\mathtt{x}_1), \ldots, \rho(\mathtt{x}_n))\{\mathtt{S}'\}} \quad \rho \text{ name bijection}$$

$$\frac{\mathtt{m}(\mathtt{x}_1, \ldots, \mathtt{x}_n)\{\mathtt{S}\} = \mathtt{m}'(\mathtt{x}_1', \ldots, \mathtt{x}_n')\{\mathtt{S}'\}}{\mathtt{x.m}(\mathtt{e}_1, \ldots, \mathtt{e}_k) = \mathtt{x.m}'(\mathtt{e}_1, \ldots, \mathtt{e}_k)}$$

$$\frac{\mathtt{m}_i(\widetilde{\mathtt{x}_i})\{\mathtt{S}_i\} = \mathtt{m}_i(\widetilde{\mathtt{x}_i}')\{\mathtt{S}_i'\}^{i \in 1..n}}{\mathtt{class\ C}\{\mathtt{f}_j^{j \in 1..k}, \mathtt{m}_i(\widetilde{\mathtt{x}_i})\{\mathtt{S}_i\}^{i \in 1..n}\} = \mathtt{class\ C}'\ \{\mathtt{f}_j^{j \in 1..k}, \mathtt{m}_i(\widetilde{\mathtt{x}_i}')\{\mathtt{S}_i'\}^{i \in 1..n}\}}$$

Table 9: Axioms for Java

We now demonstrate the first equivalence by examining the transitions resulting from the application of the semantic rules. Let us start from a configuration $\mathbb{C}, \Gamma, \mathcal{M}$ where all the free variables in $\mathtt{e}$ and $\mathtt{e}'$ are defined and $x, y \in \mathsf{dom}(\Gamma)$ are two distinct variables:

$$\begin{array}{lll}
\mathbb{C}, \Gamma, \mathcal{M} \vdash [\ \emptyset,\ \mathtt{x = e; y = e'}] & & \\
\quad \to & \mathbb{C}, \Gamma \circ \{\}_\diamond, \mathcal{M} \vdash [\ \emptyset,\ e]\ [\ x = \bullet,\ \mathtt{y} = \mathtt{e}'] & \\
\quad \to^* & \mathbb{C}, \Gamma \circ \gamma_\diamond, \mathcal{M} \vdash [\ \emptyset,\ k]\ [\ x = \bullet,\ \mathtt{y} = \mathtt{e}'] & (*) \\
\quad \to & \mathbb{C}, \Gamma, \mathcal{M} \vdash [\ x = k,\ \mathtt{y} = \mathtt{e}'] & \\
\quad \to & \mathbb{C}, \Gamma, \mathcal{M}[\Gamma(x) \to k] \vdash [\ \emptyset,\ \mathtt{y} = \mathtt{e}'] & \\
\quad \to & \mathbb{C}, \Gamma \circ \{\}_\diamond, \mathcal{M}[\Gamma(x) \to k] \vdash [\ \emptyset,\ e]\ [\ y = \bullet,\ \epsilon] & \\
\quad \to^* & \mathbb{C}, \Gamma \circ \psi_\diamond, \mathcal{M}[\Gamma(x) \to k] \vdash [\ \emptyset,\ k']\ [\ y = \bullet,\ \epsilon] & (*) \\
\quad \to & \mathbb{C}, \Gamma, \mathcal{M}[\Gamma(x) \to k] \vdash [\ y = k',\ \epsilon] & \\
\quad \to & \mathbb{C}, \Gamma, \mathcal{M}[\Gamma(x) \to k, \Gamma(y) \to k'] \vdash [\ \emptyset,\ ; \epsilon] &
\end{array}$$

Expressions are evaluated each time in a new empty frame which is discarded when a primitive value is returned ($k, k'$ in $(*)$). The assignment is performed by modifying the content of the locations pointed by the bindings in the current environment. Changing the order of the assignment does not affect the final configuration.

Starting from the second member of the first axiom we obtain:

$$\mathbb{C}, \Gamma, \mathcal{M} \vdash [\ \emptyset,\ \mathtt{y = e'; x = e}] \to^* \mathbb{C}, \Gamma, \mathcal{M}[\Gamma(y) \to k', \Gamma(x) \to k] \vdash [\ \emptyset,\ ; \epsilon]$$

The equivalence of the final configurations is established by observing the definition of $\mathsf{update}$ operation for $\mathcal{M}$ and remarking that $\Gamma(x) \neq \Gamma(y)$.

# 7 Related work

To overcome the problems and the ambiguities due to the informal specifications for Java language [10] and JVM bytecode[13], recently a number of works have proposed formalizations based on different modeling techniques.

Pioneer work on the semantics of Java has been carried on by the group of Drossopolou and Eisenbach at the Imperial College , whose research focuses on Java type system. They consider a subset of Java, Java$_s$, whose syntax resembles the original Java grammar and which is then furtherly expanded to take into account also compilation and runtime information needed to evaluate expressions. An operational semantics for $\text{Java}_s$ is provided in a sos style. The subject reduction and soundness theorem guarantee that rewriting preserves types and that erroneous situations will raise exceptions never corrupting the environment. The same subset of Java, $\text{Java}_s$, is the starting point for the work of D. Syme, [18], which provides a mechanical proof of its type system soundness by using the declarative proof system DECLARE. Exploiting the proof checker, a number of flaws and inconsistencies of the original semantics for $\text{Java}_s$, have been discovered. In the same spirit of the previous work, Nipkow and Oheimb in [14] provide a formal proof of type soundness for a subset of Java language, $Java_{light}$, by using an instantiation of the interactive theorem prover Isabelle based on Higher Order Logic.

Respect to these work, which share the same approach for the definition of Java semantics and aim to give a formal description of Java type system, we focus on the operational semantics and on the runtime environment in which Java programs are executed. We (voluntarily) cover a more restricted subset of the original language w.r.t. $\text{Java}_s$ and $Java_{light}$ still including key features of Java implementations as dynamic binding, and considering important issues as class-loading which have been leaved out from the treatment in the previous work. Instead to provide a hierarchy of languages to deal with, static, compile and runtime information, we provide a compact notation for objects which carry their compile and runtime types. The formalization of the memory and of the environment is standard and adheres to the Java implementation. Furthermore we keep us faithful to the original language, not mixing other information with the statements of the program in execution in the rewriting rules.

The work by Jensen et al. [12] provides a sufficiently clear formalization of visibility rules and dynamic class loading in Java, through a number of set and operators which provide an abstract model for class files and the class-loader. The level of abstraction at which they reason on class loading allows a rapid recognition of the interrelationships among classes, but makes it difficult to reason on the effective behavior of programs , since the abstract virtual machine behavior is modeled mixing Java source code and bytecode related information. In our approach the classloader is not an abstract entity, but it is modeled as a normal object, with its own methods, closer to actual Java implementation.

A more complete description of Java semantics is provided by Borger in [4], relying on the abstract state machine formalism and focusing on the dynamics of Java programs. The whole language is decomposed in modules of increasing complexity which cover different features of Java, starting from its imperative kernel to object oriented features, exception handling and concurrency. The work then provides a mathematical model of Java language, adhering to the reference manual, which can be used to reason on the behaviour of Java programs. But we think that our approach based on rewriting rules is more manageable than the approach with abstract state machines, where the number and the complexity of dynamic functions increase with the coverage of the features considered.

Moving the discussion to works on JVM semantics, Stata and Abadi in [17] give a type system for a small set of JVM instructions, focusing on subroutines. Dynamic semantics is provided through rules which define state transitions for the execution of each instruction while typing rules define the static semantics of a Java program. The soundness of type system with respect to the dynamic semantics is then proved, stating that any computation can stop only after the reaching of an halt instruction in a state where the operand stack is well typed. The same result but for a larger set of instructions of the JVM is stated by Qian in [15] while a complete formal specification of the JVM is provided by Bertelsen in [3]. The detailed formalization allowed the uncovering of a number of errors and ambiguities in the informal official specification provided by SUN in [13]. The low level of detail however makes it difficult to exploit such a notation for formal reasoning on Java program execution.

Even if our model for Java program execution relies on the underlying JVM mechanisms, we are not directly interested in the semantics of Java Virtual Machine. Furthermore, the previous works are based on a "closed world" assumption for classes, meaning that all classes have been already loaded by a classloader and are available during execution. Our approach to the modeling of Java semantics, could be easily extended to the formalization of bytecode execution. In any case, a formal semantics of Java bytecode combined with our semantics could be useful for the analysis and verification of Java to JVM bytecode compiling.

# 8 Conclusion and overview of the part II

In this paper we have defined the formal semantics of the sequential kernel of Java. Crucial aspects such as the notion of class, object, method invocation, hiding and overriding, class loading have been discussed in detail.

However important features, as parallelism and distribution are not covered by this article, though much of the success of Java is due to them. The second part of our research deals with these aspects detailing, in particular, all the facilities offered by Java to interface with the World Wide Web.

The model of the full language will be a refinement of the reduction semantics used here. This motivates our semantic choice. More precisely we will adopt a *chemical framework*, in the style of [2], where the semantics of a parallel or distributed system may be defined by focussing on the components which actually move. This simplifies the rules since we never care about the context as mach as we have done in the sequential case.

Up to now we have mainly used the semantics of Java to define a verification methodology for the implementation of simple Java components to be used in the building of larger applications [6]. We plan to study other formal verifications techniques for Java programs, in particular axiom systems for expressive sublanguages of Java and static analysis properties. The semantics of this paper and the forthcoming part II will become essential for pursuing on this job.

# References


[1] H.P. Barendregt. *The Lambda Calculus.* North Holland, 1985.

[2] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings $17^{th}$ ACM Symposium on Principles of Programmining Languages.* MIT Press, 1990.

[3] P. Bertelsen. Semantics of Java byte code. Technical report, Department of Information Technology, Technical University of Denmark, March 1997.

[4] E. Borger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Sintax and Semantics of Java.* Lecture Notes in Computer Science, 1998. to appear.

[5] G. Boudol. The $\pi$-calculus in direct style. In *Proceedings $24^{th}$ ACM Symposium on Principles of Programmining Languages*, pages 228 – 241. MIT Press, 1997.

[6] S. Cimato. *A Methodology for the Specification and Verification of Java Components and Architectures.* PhD thesis, Dept. of Computer Science, University of Bologna, Italy, November 1998. forthcoming.

[7] D. Dean, E. W. Felten, and D. S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the IEEE 1996 Symposium on Security and Privacy*, pages 190–200, Oakland,CA, May 1996.

[8] S. Drossopoulou and S. Eisenbach. Is the Java Type System Sound? In *Proceedins of the Fourth International Workshop on Foundations of Object-Oriented Languages*, 1997.

[9] S. Drossopoulou and S. Eisenbach. Java is Type-Safe–probably. In *11th European Conference on Object Oriented Programming*, June 1997.

[10] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[11] http://www.cs.indiana.edu/hyplan/pierce/common/prof/Types.html.

[12] T. Jensen, D. Le Metayer, and T. Thorn. Security and Dynamic Class Loading in Java: A Formalization. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*, pages 4–15, May 1998.

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

[14] T. Nipkow and D. von Oheimb. Java$_{light}$ is Type-Safe–Definitely. In *Proceedings of 25th ACM Symposium on Principles of Programming Languages*, San Diego, CA, January 1998.

[15] Z. Qian. A Formal Specification of Java Virtual Machine Instructions. Technical report, Universitat Bremen, 1997.

[16] V. Saraswat. Java is not Type Safe. Technical report, AT&T research, 1997. http://www.att.research.com/ vi.

[17] R. Stata and M. Abadi. A Type System for Java Bytecode Subroutines. Technical report, Digital Equipment Corporation, System Research Center, 1997.

[18] D. Syme. Proving Java Type Soundness. Technical Report 427, Cambridge University, 1997.