Algebraic Semantics of Statements of Sequential Java

Kazem Lellahi [1], Alexandre Zamulin[2]*

[1]LIPN, URM 7030 C.N.R.S
Université Paris 13, Institut Galilée,
99, Av. J.B. Clément, 93430 Villetaneuse France
kl@lipn.univ-paris13.fr, fax +33 (0)1 4826 0712
[2]A.P. Ershov Institute of Informatics Systems
Siberian Division of Russian Academy of Sciences
Novosibirsk 630090, Russia
zam@iis.nsk.su, fax: +7 3832 323494

Abstract. Formal semantics of some representative expressions and statements of sequential Java in the context of the state-based algebraic model of the language proposed by the authors is given. The representative expressions are field accesses, method invocation, initialized object creation, and non initialized array creation. Both basic and composed statements are considered. Basic statements are the assignment statement, method invocation, break statement, continue statement, return statement, and throw statement. Composed statements considered are the block, if statement, while loop, and try block. Finally formal semantics of the method definition is given and a notion of program specification is introduced.

Keywords: object-oriented language, object modeling, formal semantics, Java, statements, algebraic model, ASM, state update.

1 Introduction

In this work we define formal semantics of all statements of sequential Java (this means that we do not consider synchronized statements and methods). The purpose of the whole work is to define the semantics of a programming language (Java in particular) in the style of algebraic specifications where first a signature is defined, then the set of models of the signature is defined, then it is defined in what way terms in this signature are constructed and in what way they are interpreted in a given model, and finally it is defined in what way a specification is constructed and what is the set of models of a given specification. So, we want to represent the text of a (Java) program as a specification and define the semantics of the program as a set of models satisfying the specification.

The part of the work concerning the elaboration of notions similar to the signature, its set of models, and term (expressions) construction and interpretation for a Java program is reported in [11] an updated version of the report in a condensed form is given in [12]). The work is based on the ideas of Abstract State Machines [4] and D-oids [2].

In this paper we mainly consider the construction and interpretation of another kind of terms, Java statements, and their use in program specification. The paper is organized as follows. A short summary of our previous work is given in Section 2. Some representative Java expressions are summarized in Section 3. Representative Java statements are defined in Section 4. Formal semantics of method definition and a notion of program specification are given in Section 5. Related work is reviewed in Section 6, and concluding remarks are given in Section 7.

2 Formal model of a Java program

The Java model presented in [11] includes a type system consisting of basic types (boolean, integer, float, etc.) and reference types (interfaces, classes and arrays). The type system is used to construct a *program schema* (a counterpart of the algebra signature) consisting of class and interface declarations and

* The work of this author is supported in part by Russian Foundation for Basic Research under Grant 01-01-00787.

inheritance and implementation relations that define a subtype relation between classes and/or interfaces. The closure of the class schema is then constructed by extending each class/interface declaration by inherited fields and methods. The following notation is used in the sequel:

$\mathcal{S}$ — a program schema

$c <_{isa} c'$ — subtype relation between c and c' (class c is a subtype of c');

$x : t$ — declaration of a constant or variable with name x and type t;

$m(r) : t[q]$ — declaration of a method with name m, string of parameter types r, result type t, and exception types q;

$r[q]$ — declaration of a constructor with string of parameter types r and exception types q.

A program state $\mathtt{A}$ is defined as a many-sorted algebra consisting of a static part (base algebra) $\mathtt{B}$, containing sets of values and functions implementing the operations of each basic type, and dynamic part associating:

1) a set of references $\mathtt{A_{ic}}$ with each class/interface ic so that the set of references of a supertype includes the set of references of each its subtype,

2) stable constant $\mathtt{y^A_{ct}}$ with each constant declaration $y : t$ in class c,

3) updatable constant $\mathtt{x^A_{ct}}$ with each class variable declaration $x : t$ in class c,

4) partial function $\mathtt{x^A_{ct}} : \mathtt{A_c} \to \mathtt{A_t}$ with each instance variable declaration $x : t$ in class c,

5) partial functions $\mathtt{elem^A_{ct}} : \mathtt{A_c} \times \mathtt{A_{int}} \to \mathtt{A_t}$ and $\mathtt{length^A_{ct}} : \mathtt{A_c} \to \mathtt{A_{int}}$ with respective operations *elem* and *length* in each array type $c = array(t)$.

According to the definition of the program state, an object with reference $\mathtt{o}$ may belong to several classes in the inheritance hierarchy, and each of these classes is the type of this object. Class c is called the *most specific type* of the object $\mathtt{o}$ if $\mathtt{o}$ does not belong to any of its subclasses (it is denoted by $mst(\mathtt{o})$ in the sequel).

A program state can be transformed into another state by means of a *state update*, which is either a *function update* or *sort update*. A function update $(\mathtt{f_{rt}}, \bar{\mathtt{o}}, \mathtt{v})$ causes the function $\mathtt{f_{rt}}$ being applied to the tuple $\bar{\mathtt{o}}$ (empty tuple if $\mathtt{f_{rt}}$ is a constant) to yield the value $\mathtt{v}$ in the new state. A sort update $(\mathtt{c}, \mathtt{o})$ causes insertion of the reference $\mathtt{o}$ into the set of the references of the class c.

An update set Γ may contain several state updates. The notation $\mathtt{A}\Gamma$ denotes the transformation of the state $\mathtt{A}$ into a new state by simultaneous firing all state updates in Γ. The closure $\bar{\Gamma}$ of an update set Γ extends Γ by all state updates needed to preserve the validity of the state algebra (e.g, a reference inserted into a sort must be inserted into all its supersorts).

A program $\mathtt{P(B)}$ of the schema $\mathcal{S}$ possesses a set of states $|\mathtt{P(B)}|$ with the same base algebra $\mathtt{B}$ and the following functions among the others:

1) partial function $\mathtt{m^A_{cr}}(\mathtt{o}, \bar{\mathtt{v}})$ associated with each instance method declaration $m(r) : t[q]$ in class c and $\mathtt{A} \in |\mathtt{P(B)}|$ producing a pair $\langle \mathtt{A}', \mathtt{v} \rangle$ (a new algebra and value either of type t or of one of the exception types in q) for the object $\mathtt{o} \in \mathtt{A_c}$ and the tuple of arguments $\bar{\mathtt{v}}$ of the corresponding types from r,

2) partial function $\mathtt{m^A_{cr}}(\bar{\mathtt{v}})$ associated with each class (static) method declaration $m(r) : t[q]$ in class c and $\mathtt{A} \in |\mathtt{P(B)}|$ also producing a pair $\langle \mathtt{A}', \mathtt{v} \rangle$ for the tuple of arguments $\bar{\mathtt{v}}$ of the corresponding types from r.

3) partial function $\mathtt{c^A_{cr}}(\bar{\mathtt{v}})$ associated with each constructor declaration $r[q]$ in class c and $\mathtt{A} \in |\mathtt{P(B)}|$ also producing a pair $\langle \mathtt{A}', \mathtt{v} \rangle$ for the tuple of arguments $\bar{\mathtt{v}}$ of the corresponding types from r (a value of one of the types in q is produced in case an exception is thrown in the constructor, a value $\perp$ of type `void` in the normal case).

3 Expressions

In this section, some expressions typical of object-oriented languages are defined. For brevity, it is considered that an expression cannot throw an exception. We assume in this section that Σ_c is the internal class signature of some class c, which includes all the names than can be used in expressions and statements constructed within c (see [11] for the formal definition of Σ_c).

Syntax : Expressions involving field and method names are defined recursively by rules. They extend the usual rules of expression (term) construction, which are part of the base of recursion. To save space, we ignore many details including access restrictions, uses of *this* and *super*, etc. (these issues are elaborated in [11, 12]). In these rules the notation $e : t$ means that expression e has type t.

For any type t, sequence of types $r = t_1, \ldots, t_n$, class name s, interface/class name ic, constructor declaration $r[q]$), instance variable $x : t$ declaration, and instance method declaration $(m(r) : t[q]$ accessible in Σ_c,

$$\textbf{(subtyping)} \quad \frac{ic' <_{isa} ic \quad e : ic'}{e : ic}$$

$$\textbf{(field access)} \quad \frac{x : t \quad e : s}{e.x : t}$$

$$\textbf{(assignments)} \quad \frac{x : t \quad e_l : s \quad e_r : t}{e_l.x = e_r : t} \qquad \frac{x : ic \quad e_l : s}{e_l.x = null : ic}$$

$$\textbf{(method invocation)} \quad \frac{m : t_1 \ldots t_n \rightarrow t \quad e : s \quad e_1 : t_1 \ \ldots \ e_n : t_n}{e.m(e_1 \ldots e_n) : t}$$

$$\textbf{(initialized object creation)} \quad \frac{e_1 : t_1 \ \ldots \ e_n : t_n}{new\ s(e_1 \ldots e_n) : s}$$

$$\textbf{(non initialized array creation)} \quad \frac{e : int}{new\ t[e] : array(t)}$$

Semantics: The semantics of expression e of type t in algebra $\mathtt{A}$ is defined by a partial function, denoted by $[\![e]\!]^{\mathtt{A}}$, producing a pair, $\langle \mathtt{A}', \mathtt{v} \rangle$ (a new algebra and value). Rules of Java state that each next argument in a list of function arguments is evaluated after the previous one and the evaluation of both the argument and the function call may change the state. In the sequel, we denote the interpretation of a sequence of argument expressions $[\![(e_1, \ldots, e_n)]\!]^{\mathtt{A}}$ by a pair $\langle \mathtt{A_{arg}}, \bar{\mathtt{v}} \rangle$ where $\mathtt{A_{arg}}$ is the state produced by interpretation of e_n and $\bar{\mathtt{v}}$ the sequence $\mathtt{v_1}, \ldots, \mathtt{v_n}$ of the argument values produced. We consider in the sequel that a program `P(B)` is fixed and denote by `A` with an upper or lower index a state of this program. Given a state `A`, the semantics of expressions is defined recursively by the following rules:

$$\textbf{(subtyping)} \quad \frac{ic' <_{isa} ic}{[\![e : ic]\!]^{\mathtt{A}} = [\![e : ic']\!]^{\mathtt{A}}}.$$

$$\textbf{(field access)} \quad \frac{[\![e]\!]^{\mathtt{A}} = \langle \mathtt{A}', \mathtt{o} \rangle}{[\![e.x]\!]^{\mathtt{A}} = \langle \mathtt{A}', \mathtt{x}_{\mathtt{st}}^{\mathtt{A}'}(\mathtt{o}) \rangle} \qquad [\![e.x]\!]^{\mathtt{A}} \text{ is undefined if so is } \mathtt{x}_{\mathtt{st}}^{\mathtt{A}'}(\mathtt{o}) \ \mathtt{or}\ [\![e]\!]^{\mathtt{A}}.$$

$$\textbf{(assignments)} \quad \frac{[\![e_l]\!]^{\mathtt{A}} = \langle \mathtt{A_1}, \mathtt{v_1} \rangle \quad [\![e_r]\!]^{\mathtt{A_1}} = \langle \mathtt{A_r}, \mathtt{v_r} \rangle \quad \Gamma_{\mathtt{a}} = \overline{\{(\mathtt{x_s}, \mathtt{v_1}, \mathtt{v_r})\}} \quad \mathtt{A_a} = \mathtt{A_r}\Gamma_{\mathtt{a}}}{[\![e_l.x = e_r]\!]^{\mathtt{A}} = \langle \mathtt{A_a}, \mathtt{v_r} \rangle}$$

$$\frac{[\![e_l]\!]^{\mathtt{A}} = \langle \mathtt{A_1}, \mathtt{v_1} \rangle \quad \Gamma_{\mathtt{a}} = \overline{\{(\mathtt{x_s}, \mathtt{v_1}, \mathtt{null})\}} \quad \mathtt{A_a} = \mathtt{A_1}\Gamma_{\mathtt{a}}}{[\![e_l.x = null]\!]^{\mathtt{A}} = \langle \mathtt{A_a}, \mathtt{null} \rangle}$$

$[\![e_l.x = e_r]\!]^{\mathtt{A}}$ and $[\![e_l.x = null]\!]^{\mathtt{A}}$ are not defined if so is $\mathtt{v_r}$ `or` $\mathtt{v_1}$ *is* `null`.

$$\textbf{(instance method invocation)} \quad \frac{[\![e]\!]^{\mathtt{A}} = \langle \mathtt{A_e}, \mathtt{o} \rangle \quad [\![(e_1, \ldots, e_n)]\!]^{\mathtt{A_e}} = \langle \mathtt{A_{arg}}, \bar{\mathtt{v}} \rangle \quad c' = mst(\mathtt{o})}{[\![e.m(e_1, \ldots, e_n)]\!]^{\mathtt{A}} = \mathtt{m}_{\mathtt{c'r}}^{\mathtt{A_{arg}}}(\mathtt{o}, \bar{\mathtt{v}})}$$

$[\![e.m(e_1, \ldots, e_n)]\!]^{\mathtt{A}}$ `is undefined if` $[\![\mathtt{e}]\!]^{\mathtt{A}}$ `or` $[\![\mathtt{e_i}]\!]^{\mathtt{A_i}}$ `or` $\mathtt{m}_{\mathtt{c'r}}^{\mathtt{A_m}}(\mathtt{o}, \bar{\mathtt{v}})$ `is undefined.`

$$\textbf{(class method invocation)} \frac{[\![(e_1, \ldots, e_n)]\!]^{\mathtt{A_e}} = \langle \mathtt{A_{arg}}, \bar{\mathtt{v}} \rangle}{[\![c.m(e_1, \ldots, e_n)]\!]^{\mathtt{A}} = \mathtt{m}_{\mathtt{cr}}^{\mathtt{A_{arg}}}(\bar{\mathtt{v}})}$$

$[\![e.m(e_1, \ldots, e_n)]\!]^{\mathtt{A}}$ `is undefined if` $[\![\mathtt{e}]\!]^{\mathtt{A}}$ `or` $[\![\mathtt{e_i}]\!]^{\mathtt{A_i}}$ `or` $\mathtt{m}_{\mathtt{c'r}}^{\mathtt{A_m}}(\mathtt{o}, \bar{\mathtt{v}})$ `is undefined.`

(initialized object creation)

$$\frac{\mathtt{o} \in \mathtt{Ref} \text{ and } \mathtt{o} \notin |\mathtt{A}| \quad \delta = (\mathtt{s}, \mathtt{o}) \quad \Gamma_\delta = \overline{\{\delta\}} \quad \mathtt{A}_\delta = \mathtt{A}\Gamma_\delta \quad [\![(e_1, \ldots, e_n)]\!]^{\mathtt{A}_\delta} = \langle \mathtt{A_{arg}}, \bar{\mathtt{v}} \rangle \quad \mathtt{s}_{\mathtt{sr}}^{\mathtt{A_{arg}}}(\mathtt{o}, \bar{\mathtt{v}})) = \langle \mathtt{A_{con}}, \perp \rangle}{[\![new\ s(e_1, \ldots, e_n)]\!]^{\mathtt{A}} = \langle \mathtt{A_{con}}, \mathtt{o} \rangle}$$

provided $[\![e_1]\!], \cdots, [\![e_n]\!]$, and $\mathtt{s}_{\mathtt{sr}}^{\mathtt{A_{con}}}(\mathtt{o}, \bar{\mathtt{v}})$ are defined.

(non initialized array creation)

$$\frac{[\![\mathtt{e}]\!]^{\mathtt{A}} = \langle \mathtt{A_e}, \mathtt{n} \rangle \quad \mathtt{o} \in \mathtt{Ref} \text{ and } \mathtt{o} \notin |\mathtt{A_e}| \quad \delta = (\mathtt{c}, \mathtt{o}) \quad \Gamma_\delta = \overline{\{\delta\}} \quad \mathtt{A}_\delta = \mathtt{A_e}\Gamma_\delta \quad \Gamma_1 = \{(\mathtt{length}_{\mathtt{array(t)}}, \mathtt{o}, \mathtt{n})\} \quad \mathtt{A_1} = \mathtt{A}_\delta\Gamma_1}{[\![new\ t[e]]\!]^{\mathtt{A}} = \langle \mathtt{A_1}, \mathtt{o} \rangle}$$

$[\![new\ t[e]]\!]^{\mathtt{A}}$ is not defined if $\mathtt{n} < 0$

In what follows we give a comment for each of the above rule.

1. A subtype expression can be used in a context where an expression of its supertype is needed.

2. A field value is produced by the corresponding field function, and a new algebra is produced in addition if the interpretation of e changes the state.

3. In the assignment expression first the left hand is evaluated and then the right hand is evaluated in the new state. The assignment (field function update) is made then producing an update set. Note that this update set is closed to ensure that the function is updated in all classes supplying or inheriting this field.

4. The interpretation of a method invocation in state $\mathtt{A}$ causes the invocation (in the state obtained by the evaluation of the method arguments) of the method associated with the most specific type of object $\mathtt{o}$. In this way, the method name is bound exactly to the method associated with the object independent of its current type, i.e., *dynamic (late) binding* is provided.

5. The interpretation of the expression $new\ s(t_1, \cdots, t_n)$ leads to the creation of an initialized object. Note that the carrier update δ is closed to ensure the insertion of the new reference into all superclasses/superinterfaces of s.

6. A non-initialized array creation computes the length of a new array, expands the set of references of a given array type by a new reference and sets the value of the attribute *length* for this array object.

The names declared in class c may be used within the class without qualifying them by anything. The variables *this* and *super* also can be used in expressions. This is explained by the following rules. For any type t, sequence of types $r = t_1, \ldots, t_n$, constructor declaration $r[q] \in con(c)$, instance variable declaration $(x : t) \in \overline{ivar}(c)$, instance method declaration $(m(r) : t[q]) \in \overline{imeth}(c)$ and class method declaration $(m(r) : t[q]) \in \overline{cmeth}(c)$,

(this) $\dfrac{}{this : c}$

(super) $\dfrac{c <_{isa} c' \quad \nexists c'' : c <_{isa} c'' <_{isa} c'}{super : c'}$ (variable "super" gets the type of the direct superclass of c)

(field access) $\dfrac{}{x : t}$

(assignments) $\dfrac{x : t \quad e_r : t}{x = e_r : t}$ $\qquad \dfrac{x : ic}{x = null : ic}$

(instance method invocation) $\dfrac{e_1 : t_1 \ \ldots \ e_n : t_n}{m(e_1 \ldots e_n) : t}$

(class method invocation) $\dfrac{e_1 : t_1 \ \ldots \ e_n : t_n}{m(e_1 \ldots e_n) : t}$

Interpretations of these expressions are very simple and do not need special comments:

(this) $\dfrac{\mathtt{o} \in \mathtt{A_c} \quad this \mapsto \mathtt{o}}{[\![this]\!]^{\mathtt{A}} = \langle \mathtt{A}, \mathtt{o} \rangle}$

(super) $\dfrac{\mathtt{o} \in \mathtt{A_c} \quad super \mapsto \mathtt{o}}{[\![super]\!]^{\mathtt{A}} = \langle \mathtt{A}, \mathtt{o} \rangle}$ (the subclass object with the type of its direct superclass)

(field access) $\dfrac{}{[\![x]\!]^{\mathtt{A}} = [\![this.x]\!]^{\mathtt{A}}}$

(assignments) $\dfrac{}{[\![x = e_r]\!]^{\mathtt{A}} = [\![this.x = e_r]\!]^{\mathtt{A}}}$ $\qquad \dfrac{}{[\![x = null]\!]^{\mathtt{A}} = [\![this.x = null]\!]^{\mathtt{A}}}$

(instance method invocation) $\dfrac{}{[\![m(e_1, \ldots, e_n)]\!]^{\mathtt{A}} = [\![this.m(e_1, \ldots, e_n)]\!]^{\mathtt{A}}}$

(class method invocation) $\dfrac{}{[\![m(e_1, \ldots, e_n)]\!]^{\mathtt{A}} = [\![c.m(e_1, \ldots, e_n)]\!]^{\mathtt{A}}}$

4 Statements

In this section we give semantics of some representative Java statements. One can find more in [12].

Each statement is described in this paper in three parts: *definition*, *typing*, and *dynamic semantics*. The definition generally states what a well-formed statement is. The typing indicates the kind of the statement and the type of a possible result. The dynamic semantics indicates the algebra and its element that may be produced by the execution of the statement.

A statement may be provided with a label that can be used by some statements for control transfer. We consider that the label has the string type, and it is the empty string in an unlabeled statement.

Once again, we assume that Σ_c is the internal signature of some class c, which includes all the names than can be used in expressions and statements constructed within c. Note that any Σ_c-algebra is a state algebra since it contains only the elements associated with the components of the program schema. For this reason, a Σ_c-algebra is also a Σ_s-algebra for any other class s. This fact will be used in the definition of the semantics of the method definition in Section 6.

The term "Σ-statement" used in the sequel means that the statement is constructed in the signature Σ, which is either the internal class signature or its extension in a block statement (see Section 4.2). In accordance with this, a Σ-algebra $\mathtt{A}$ is either a program state or some its extension (the term "state" used in the sequel refers to either of them).

Taking into account that some statements do not transfer control and some others do, we distinguish five kinds of statement: ordinary, break, continue, return and throw statements. Some of these statements do not produce any value as result, some others do. To define the typing of a complex statement, one has to analyze its behavior in different program states, and result is not always defined. Therefore, typing of a statement in state $\mathtt{A}$ is a partial function, denoted by $[]^{\mathtt{A}}$, which produces a pair $\langle \mathtt{t}, \mathtt{k}\rangle$, where $\mathtt{t}$ is the type of the value that may be produced by the statement and $\mathtt{k}$ is one of the symbols $\mathbf{O}$, $\mathbf{B}$, $\mathbf{C}$, $\mathbf{R}$, or $\mathbf{T}$ denoting, respectively, the *ordinary*, *break*, *continue*, *return*, or *throw* statement.

The dynamic semantics of a statement in state $\mathtt{A}$ is a partial function, denoted by $[\![]\!]^{\mathtt{A}}$, which produces a pair, $\langle \mathtt{A}', \mathtt{v}\rangle \in |\mathtt{P(B)}|_{\mathtt{t}}$, where $\mathtt{A}'$ is a new algebra and $\mathtt{v}$ a value of the type produced by typing. This value may be either the label of the statement to be terminated by the break statement or continued by the continue statement (empty string if no label is indicated in the corresponding break or continue statement), or the value produced by a return or throw statement (the value $\bot$ if the result type is `void`).

4.1 Basic statements

Assignment statements. If $e_l = e_r$ is an assignment expression, then $e_l = e_r;$ is a Σ-statement called the *assignment statement* (i.e., the assignment expression in the context where, according to the Java grammar, a statement is needed is a statement).

Semantics. $[e_l = e_r;]^{\mathtt{A}} = \langle \mathtt{void}, \mathbf{O}\rangle$ and $[\![e_l = e_r;]\!]^{\mathtt{A}} = \langle \mathtt{A}', \bot\rangle$,
where $\mathtt{A}'$ is the algebra produced by evaluations of e_l, e_r and the assignment itself (see details in the expression section). Thus, the assignment statement is an ordinary statement not transferring control. Note that an ordinary statement always produces the result $\bot$ (a single value associated with type `void`), which means that no result is produced at all.

Method invocations. If $(m(r) : \mathtt{void}[q]$, where $r = \langle t_1 \dots t_n\rangle$, is an instance method from Σ_c, and $e, e_1, \dots, e_n$ are Σ-expressions of respective types $s, t_1, \dots, t_n$, then $e.m(e_1, \dots, e_n)$ is a Σ-statement called the *instance method invocation*.

Typing: $[e.m(e_1, \dots, e_n)]^{\mathtt{A}} = \langle \mathtt{t}, \mathtt{k}\rangle$.
In this tuple, $\mathtt{k}$ may be either $\mathbf{O}$ (normal termination) or $\mathbf{T}$ (exception is thrown); type $\mathtt{t}$ is **void** when $\mathtt{k}$ is $\mathbf{O}$, and $\mathtt{t}$ is one of the types in q or a subtype of it, otherwise.

Dynamic semantics. As it was defined in Section 2.5, the invocation of a method on a given object is interpreted by the execution of the function associated with the method in the most specific type of the object (dynamic binding). In case of method invocation as statement, this produces a pair $\langle \mathtt{A}', \mathtt{v}\rangle$, where $\mathtt{A}'$ is a new algebra and $\mathtt{v}$ a value of type t in case an exception of type t is thrown and the value $\bot$) in case of normal method termination.

The semantics of the class method invocation is similar to the semantics of the instance method invocation.

Break statements. If $\mathtt{M}$ is a label, them **break** and **break** $\mathtt{M}$ are Σ-statements called the *break statements.*

Semantics.
$[\mathbf{break}]^{\mathtt{A}} = \langle \mathtt{string}, \mathbf{B}\rangle$ and $[\![\mathbf{break}]\!]^{\mathtt{A}} = \langle \mathtt{A}, \langle\rangle\rangle$
$[\mathbf{break}\ \mathtt{M}]^{\mathtt{A}} = \langle \mathtt{string}, \mathbf{B}\rangle$ and $[\![\mathbf{break}\ \mathtt{M}]\!]^{\mathtt{A}} = \langle \mathtt{A}, \mathtt{M}\rangle$.
Thus, the typing of the statement indicates that control transfer takes place and the type of an optional label is the result type. The dynamic semantics indicates that the statement causes no state update, but it produces the label of the statement to be terminated (empty string if no label is indicated in the break

statement).

Continue statements. If M is a label, then **continue** and **continue** M are Σ-statements called the *continue statements.*

Semantics.
$[\textbf{continue}]^{\mathtt{A}} = \langle \mathtt{string}, \mathbf{C}\rangle$ and $[\![\textbf{continue}]\!]^{\mathtt{A}} = \langle \mathtt{A}, \langle\rangle\rangle$
$[\textbf{continue } \mathtt{M}]^{\mathtt{A}} = \langle \mathtt{string}, \mathbf{C}\rangle$ and $[\![\textbf{continue } \mathtt{M}]\!]^{\mathtt{A}} = \langle \mathtt{A}, \mathtt{M}\rangle$.
The semantics of this statement is similar to the previous one with the exception that another kind of statement is indicated.

Return statements. If e is a Σ-expression of type t, then **return** and **return** e are Σ-statements called the *return statements.*

Semantics.
$[\textbf{return}]^{\mathtt{A}} = \langle \mathtt{void}, \mathbf{R}\rangle$ and $[\![\textbf{return}]\!]^{\mathtt{A}} = \langle \mathtt{A}, \perp\rangle$
$[\textbf{return } e]^{\mathtt{A}} = \langle \mathtt{t}, \mathbf{R}\rangle$ and $[\![\textbf{return } e]\!]^{\mathtt{A}} = [\![e]\!]^{\mathtt{A}}$.
The semantics indicates that the statement neither updates the state nor produces a value in the first case and produces a value (which becomes the result of the method containing this statement) with a possible update of the state.

Throw statements. If e is a Σ-expression of an exception type t, then **throw** e is a Σ-statement called the *throw statement.*

Semantics. $[\textbf{throw } e]^{\mathtt{A}} = \langle \mathtt{t}, \mathbf{T}\rangle$ and $[\![\textbf{throw } e]\!]^{\mathtt{A}} = [\![e]\!]^{\mathtt{A}}$. Thus, the throw statement produces a result of some exception type (which will be used by the corresponding exception handler), optionally updates the state and transfers control.

4.2 Statement constructors

Complex statements are constructed recursively from basic statements by means of several statement constructors. In this section, we will define some of them. For convenience, we consider separately sequences of statements not containing local declarations and block statements containing declarations of local variables.

Sequences of statements. If $St_1, St_2, \ldots, St_n$ are Σ-statements, then $\{\ St_1\ St_2\ \ldots\ St_n\ \}$ is a *sequence of Σ-statements.*

Semantics. The semantics of this statement is defined recursively.

Base case. $[\{\}]^{\mathtt{A}} = \langle \mathtt{void}, \mathbf{O}\rangle$ and $[\![\{\}]\!]^{\mathtt{A}} = \langle \mathtt{A}, \perp\rangle$.

General case. Let $[St_1]^{\mathtt{A}} = \langle \mathtt{t}, \mathtt{k}\rangle$ and $[\![St_1]\!]^{\mathtt{A}} = \langle \mathtt{A}', \mathtt{v}\rangle$. If $\mathtt{k} = \mathbf{O}$ (an ordinary statement has been executed), then
$[\{\ St_1\ St_2\ \ldots\ St_n\ \}]^{\mathtt{A}} = [\{\ St_2 \ldots St_n\ \}]^{\mathtt{A}'}$ and $[\![\{\ St_1\ St_2\ \ldots\ St_n\ \}]\!]^{\mathtt{A}} = [\![\{\ St_2\ \ldots\ St_n\ \}]\!]^{\mathtt{A}'}$.
Thus, in this case the typing of the sequence is the typing of the tail of the sequence in the new state, and the dynamic semantics depends on the execution of the tail of the sequence in the new state

If k is one of **T**, **R**, **B**, or **C** (throw, return, break or continue statement has been executed), then
$[\{\ St_1\ St_2\ \ldots\ St_n\ \}]^{\mathtt{A}} = \langle \mathtt{t}, \mathtt{k}\rangle$ and $[\![\{\ St_1\ St_2\ \ldots\ St_n\ \}]\!]^{\mathtt{A}} = \langle \mathtt{A}', \mathtt{v}\rangle$.
In this case, the typing of the sequence is the typing of the first control transfer statement in the sequence, and the execution of the sequence is terminated with producing a new algebra and value.

Thus, to execute a sequence of statements starting with a state A, execute successively each next statement until either the end of the sequence is reached or a control transfer statement is encountered.

$[\![\{\ St_1\ St_2\ \ldots\ St_n\ \}]\!]^{\mathtt{A}}$ is not defined if $[\![St_1]\!]^{\mathtt{A}}$ or $[\![\{\ St_2 \ldots St_n\ \}]\!]^{\mathtt{A}'}$ is not defined.

Blocks. For brevity, we consider that only non-initialized local variables can be declared in a block, any array type used in local declarations belongs to AR, and declarations precede statements.

First of all, we introduce a notion of signature increment simulating declarations of local variables. Let $t_1, ..., t_n$ be types and $x_1, ..., x_n$ different identifiers, then the set
$\Delta = \{x_1 : t_1, ..., x_n : t_n\}$
is a *signature increment.* Two signature increments
$\Delta_1 = \{x_{11} : t_{11}, ..., x_{1n} : t_{1n}\}$ and $\Delta_2 = \{x_{21} : t_{21}, ..., x_{2m} : t_{2m}\}$
are *union-compatible* if there is no $i = 1, ..., n$ and $j = 1, ..., m$ such that $x_{1i} = x_{2j}$. Thus, if a block is

declared within a block, no identifier conflict may take place (Java does not permit to redeclare in a inner block the identifiers of an outer block).

Second, we introduce a notion of block signature containing the names that can be used in the block. Let Σ_c be the internal signature of the class c and Δ_1 a signature increment, then $\Sigma_1 = (\Sigma_c, \Delta_1)$ is a *block signature*. Now, if $\Sigma_{n-1} = (\Sigma_{n-2}, \Delta_{n-1})$ is a block signature and Δ_n is a signature increment union-compatible with $\Delta_1 \cup ... \cup \Delta_{n-1}$, then $\Sigma_n = (\Sigma_{n-1}, \Delta_n)$ is a *block signature*. Thus, a block signature generally looks as follows:

$$((((\Sigma_c, \Delta_1), ...), \Delta_{n-1}), \Delta_n).$$

In the sequel, we denote the block signature by $\Sigma' = (\Sigma, \Delta_n)$ where $\Sigma = (((\Sigma_c, \Delta_1), ...), \Delta_{n-1})$. We also denote $\Delta_1 \cup ... \cup \Delta_n$ by Δ (i.e., Δ is the signature increment with respect to Σ_c).

We should recall that names declared in the class c may be used without qualification when Σ_c-expressions are constructed. Unfortunately, this is not always possible when expressions of signature Σ', which is an extension of Σ_c, are constructed. The rules of Java state that if a local variable in a block has the same name as a field in the class, then the field name is shadowed by the local variable name. Therefore, we have to modify slightly the rules of construction of expressions using field names of the class c when they are constructed in a block and define rules for construction of expressions using local variables. For example:

- if $x : t$ is an instance variable from $\overline{ivar}(c)$ and $(x : t') \notin \Delta$ for some type t', then x is a Σ'-expression of type t.
- if $x : t$ is a local variable from Δ, then x is a Σ'-expression of type t.

Now, if Δ_n is a signature increment, $\Sigma' = (\Sigma, \Delta_n)$, and $St_1, ..., St_m$ a sequence of Σ'-statements, then

$$\{\ \Delta_n\ St_1\ St_2 \ldots St_m\ \}$$

is a Σ-statement called the *block*.

Now, if $\mathtt{A}$ is a Σ-algebra, then the block algebra $\mathtt{A}'$ is constructed by extending $\mathtt{A}$ with a constant $\mathtt{x}^{\mathtt{A}'}$ of the corresponding type for each $(x : t) \in \Delta_n$, and this constant interprets the name x. Respectively, each function associated with method declaration or constructor declaration is extended so that being invoked in a block algebra, it produces another block algebra as part of the result (see [12] for details).

Semantics.

$$[\{\ \Delta_n\ St_1\ St_2 \ldots St_m\ \}]^{\mathtt{A}} = [\{St_1\ St_2 \ldots St_m\ \}]^{\mathtt{A}}.$$

The typing of the block is the typing of the enclosed sequence of statements. Now let

$$[\![\{\ St_1\ St_2 \ldots St_m\ \}]\!]^{\mathtt{A}'} = \langle \mathtt{A}'_1, \mathtt{v} \rangle,$$

then

$$[\![\{\ \Delta_n\ St_1\ St_2 \ldots St_m\ \}]\!]^{\mathtt{A}} = \langle \mathtt{A}_1, \mathtt{v} \rangle$$

where $\mathtt{A}_1 = \mathtt{A}'_1|_{\Sigma}$. Thus, by taking the reduct of the algebra obtained by the execution of the block to the original signature Σ, all local variables of the block are forgotten when the block is exited.

A block with intermixed variable declarations and statements may be considered as a number of nested blocks where each of them satisfies the condition stated at the beginning of the section. If Δ_n contains an initialized local variable declaration $t\ x = e$, then the algebra $\mathtt{A}'$ is updated by the evaluation of the statement $x = e$.

If statements. If g is a Σ-expression of type Boolean and St and St_e are Σ-statements, then

if (g) St and **if** (g) St **else** St_e

are Σ-statements called the *if statements*.

Semantics. Denote **if** (g) St **else** St_e by ISt and let $\mathtt{A}$ be a Σ-algebra and $[\![g]\!]^{\mathtt{A}} = \langle \mathtt{A_g}, \mathtt{g} \rangle$, then

if `g = true`,
then $[ISt]^{\mathtt{A}} = [St]^{\mathtt{A_g}}$ and $[\![ISt]\!]^{\mathtt{A}} = [\![St]\!]^{\mathtt{A_g}}$,
else $[ISt]^{\mathtt{A}} = [St_e]^{\mathtt{A_g}}$ and $[\![ISt]\!]^{\mathtt{A}} = [\![St_e]\!]^{\mathtt{A_g}}$.

If `g = true`, then the semantics of the statement is not defined if $[\![St]\!]^{\mathtt{A_g}}$ is not defined; if `g = false`, then the semantics of the statement is not defined if $[\![St_e]\!]^{\mathtt{A_g}}$ is not defined.

The semantics of **if** (g) St is self-evident.

While loops. If St is a Σ-statement and g a Σ-expression of type Boolean, then M : **while** (g) St is a Σ-statements. The label M may be the empty string.

Semantics. Denote **while** (g) St by WSt and let $g^{\mathtt{A}} = \langle \mathtt{A_g}, \mathtt{g} \rangle$. Now,
if $\mathtt{g} = \mathtt{false}$ (the execution of the statement is terminated)

then $[M:\ WSt]^{\mathtt{A}} = \langle \mathtt{void}, \mathbf{O}\rangle$ and $[\![M:\ WSt]\!]^{\mathtt{A}} = \langle \mathtt{A_g}, \bot\rangle$
else let $[St]^{\mathtt{A_g}} = \langle \mathtt{t_{st}}, \mathtt{k_{st}}\rangle$ and $[\![St]\!]^{\mathtt{A_g}} = \langle \mathtt{A_{st}}, \mathtt{v_{st}}\rangle$ in
if $\mathtt{k_{st}} = \mathbf{O}$ (St completes normally)
then $[M:\ WSt]^{\mathtt{A}} = [M:\ WSt]^{\mathtt{A_{st}}}$ and $[\![M:\ WSt]\!]^{\mathtt{A}} = [\![M:\ WSt]\!]^{\mathtt{A_{st}}}$
else (St completes abnormally)
if $\mathtt{k_{st}} = \mathbf{B}$ (break statement has been executed)
then if $(\mathtt{v_{st}} = \mathtt{M}) \vee (\mathtt{v_{st}} = \langle\rangle)$ (this loop should be terminated)
then $[M:\ WSt]^{\mathtt{A}} = \langle \mathtt{void}, \mathbf{O}\rangle$ and $[\![M:\ WSt]\!]^{\mathtt{A}} = \langle \mathtt{A_{st}}, \bot\rangle$ (finish normally)
else $[M:\ WSt]^{\mathtt{A}} = \langle \mathtt{string}, \mathbf{B}\rangle$ and $[\![M:\ WSt]\!]^{\mathtt{A}} = \langle \mathtt{A_{st}}, \mathtt{v_{st}}\rangle$ (finish abnormally)
else if $\mathtt{k} = \mathbf{C}$ (continue statement has been executed)
then if $(\mathtt{v_{st}} = \mathtt{M}) \vee (\mathtt{v_{st}} = \langle\rangle)$ (this loop should be continued)
then $[M:\ WSt]^{\mathtt{A}} = [M:\ WSt]^{\mathtt{A_{st}}}$ and $[\![M:\ WSt]\!]^{\mathtt{A}} = [\![M:\ WSt]\!]^{\mathtt{A_{st}}}$
else $[M:\ WSt]^{\mathtt{A}} = \langle \mathtt{string}, \mathbf{C}\rangle$ and $[\![M:\ WSt]\!]^{\mathtt{A}} = \langle \mathtt{A_{st}}, \mathtt{v_{st}}\rangle$ (finish abnormally)
else (return or throw statement has been executed)
$[M:\ WSt]^{\mathtt{A}} = \langle \mathtt{t_{st}}, \mathtt{k_{st}}\rangle$ and $[\![M:\ WSt]\!]^{\mathtt{A}} = \langle \mathtt{A_{st}}, \mathtt{v_{st}}\rangle$ (finish abnormally).
$[\![M:\ WSt]\!]^{\mathtt{A}}$ is not defined if $[\![St]\!]^{\mathtt{A_g}}$ is not defined or no definite values of typing and dynamic semantics are produced (the loop does not terminate).

The semantics of the do-while loop and for loop are defined in a similar way.

Try statements. To save space, we will consider only the general case of the try statement. Let $St, FSt, St_1,\ ...,\ St_n$ be Σ-blocks, $p_1, ..., p_n$ identifiers, and $t_1, ..., t_n$ types, then

try St
catch $(t_1\ p_1)\ St_1$
...
catch $(t_n\ p_n)\ St_n$
finally FSt

is a Σ-statement called the *try statement*.

Semantics. Denote the try statement by TSt and let $[St]^{\mathtt{A}} = \langle \mathtt{t}, \mathtt{k}\rangle$ and $[\![St]\!]^{\mathtt{A}} = \langle \mathtt{A_{st}}, \mathtt{v}\rangle$. Now,
if $\mathtt{k} = \mathbf{T}$ (an exception is thrown in St)
then if t_i is the first type in the sequence $t_1, ..., t_n$ such that $t \leq_{isa} t_i$
then let $\Sigma' = (\Sigma, \{p_i : t_i\})$, $\mathtt{A}'$ a Σ'-algebra extending $\mathtt{A_{st}}$ by the constant $\mathtt{p_i^{A'}} = \mathtt{v}$,
$[St_i]^{\mathtt{A}'} = \langle \mathtt{t_i}, \mathtt{k_i}\rangle$, $[\![St_i]\!]^{\mathtt{A}'} = \langle \mathtt{A_i}, \mathtt{v_i}\rangle$, $[FSt]^{\mathtt{A_i}} = \langle \mathtt{t_f}, \mathtt{k_f}\rangle$ and $[\![FSt]\!]^{\mathtt{A_i}} = \langle \mathtt{A_f}, \mathtt{v_f}\rangle$ in:
if $\mathtt{k_f} = \mathbf{O}$ (finally block completes normally)
then $[TSt]^{\mathtt{A}} = \langle \mathtt{t_i}, \mathtt{k_i}\rangle$ and $[\![TSt]\!]^{\mathtt{A}} = \langle \mathtt{A_f}, \mathtt{v_i}\rangle$ ($\mathtt{v_i}, \mathtt{t_i}, \mathtt{k_i}$ from St_i are used)
else $[TSt]^{\mathtt{A}} = \langle \mathtt{t_f}, \mathtt{k_f}\rangle$ and $[\![TSt]\!]^{\mathtt{A}} = \langle \mathtt{A_f}, \mathtt{v_f}\rangle$ ($\mathtt{v_f}, \mathtt{t_f}, \mathtt{k_f}$ from FSt are used)
else (no catch clause fits t)
let $[FSt]^{\mathtt{A_{st}}} = \langle \mathtt{t_f}, \mathtt{k_f}\rangle$ and $[\![FSt]\!]^{\mathtt{A_{st}}} = \langle \mathtt{A_f}, \mathtt{v_f}\rangle$ in
if $\mathtt{k_f} = \mathbf{O}$ (FSt completes normally)
then $[TSt]^{\mathtt{A}} = \langle \mathtt{t}, \mathtt{k}\rangle$ and $[\![TSt]\!]^{\mathtt{A}} = \langle \mathtt{A_f}, \mathtt{v}\rangle$ ($\mathtt{v}, \mathtt{t}, \mathtt{k}$ from St are used)
else $[TSt]^{\mathtt{A}} = \langle \mathtt{t_f}, \mathtt{k_f}\rangle$ and $[\![TSt]\!]^{\mathtt{A}} = \langle \mathtt{A_f}, \mathtt{v_f}\rangle$ ($\mathtt{v_f}, \mathtt{t_f}, \mathtt{k_f}$ from FSt are used)
else (no exception thrown)
let $[FSt]^{\mathtt{A_{st}}} = \langle \mathtt{t_f}, \mathtt{k_f}\rangle$ and $[\![FSt]\!]^{\mathtt{A_{st}}} = \langle \mathtt{A_f}, \mathtt{v_f}\rangle$ in
if $\mathtt{k_f} = \mathbf{O}$ (FSt completes normally)
then $[TSt]^{\mathtt{A}} = \langle \mathtt{t}, \mathtt{k}\rangle$ and $[\![TSt]\!]^{\mathtt{A}} = \langle \mathtt{A_f}, \mathtt{v}\rangle$ ($\mathtt{v}, \mathtt{t}, \mathtt{k}$ from St are used)
else $[TSt]^{\mathtt{A}} = \langle \mathtt{t_f}, \mathtt{k_f}\rangle$ and $[\![TSt]\!]^{\mathtt{A}} = \langle \mathtt{A_f}, \mathtt{v_f}\rangle$ ($\mathtt{v_f}, \mathtt{t_f}, \mathtt{k_f}$ from FSt are used).
The semantics of the try block TSt is not defined if either the semantic of the internal statement St or exception handler St_i (if an exception of type t_i is thrown) or final statement FSt is not defined.

5 Method definition and program specification

If $m(r) : t[q]$, where $r = t_1, ..., t_n$, is the declaration of a method in class c, Σ_c the internal signature of c, $\Delta = \{p_1 : t_1, ..., p_n : t_n\}$ a signature increment, St a Σ_c-block with the signature increment Δ, and t a type, then

$t\ m(t_1\ p_1, ..., t_n\ p_n)$ **throws** $q\ St$

is a *method definition* in class c. Putting in terms of algebraic specifications, the method definition can be considered as a combination of the method declaration $t\ m(t_1, ..., t_n)$ **throws** q and the equation

$m(p_1, ..., p_n) \ = \ St.$

A *program specification* $\mathcal{PS}$ for a program schema $\mathcal{S}$ is the set of definitions of the methods of $\mathcal{S}$. The program specification $\mathcal{PS}$ is *complete* if each concrete method of $\mathcal{S}$ is defined[1].

Semantics. Let Σ be the signature of the program fragment where the method m is invoked, $\mathtt{A}$ a Σ-algebra, $e, e_1, ..., e_n$ are Σ-expressions of respective types $c, t_1, ..., t_n$,

$[\![e]\!]^{\mathtt{A}} = \langle \mathtt{A_0}, \mathtt{o} \rangle$, $[\![e_1]\!]^{\mathtt{A_0}} = \langle \mathtt{A_1}, \mathtt{v_1} \rangle$, . . . , $[\![e_n]\!]^{\mathtt{A_{n-1}}} = \langle \mathtt{A_n}, \mathtt{v_n} \rangle$,

$\mathtt{A_c} = \mathtt{A_n}|_{\Sigma_c}$ and $\mathtt{A}'$ a (Σ_c, Δ)-algebra extending $\mathtt{A_c}$ as follows: $\mathtt{p_1^{A'}} = \mathtt{v_1}$, ..., $\mathtt{p_n^{A'}} = \mathtt{v_n}$. Thus, using the reduct $\mathtt{A_n}|_{\Sigma_c}$, we have obtained the algebra of the signature where the block St is constructed (i.e., the algebra of the signature (Σ_c, Δ)). This algebra extends the state algebra existing at method invocation by the values of the method actual parameters[2]. Now, let $this \mapsto \mathtt{o}$ be a valuation of the variable *this* (it may be explicitly or implicitly used in St) and under this condition:

$[\![e.m(e_1, ..., e_n)]\!]^{\mathtt{A}} = \langle \mathtt{A_m}, \mathtt{v_m} \rangle$, $[St]^{\mathtt{A}'} = \langle \mathtt{t_{st}}, \mathtt{k_{st}} \rangle$ and $[\![St]\!]^{\mathtt{A}', \sigma} = \langle \mathtt{A_{st}}, \mathtt{v_{st}} \rangle$.

The method definition $t \ m(t_1 \ p_1, ..., t_n \ p_n)$ **throws** $q \ St$ is *well-formed* iff

$\mathtt{k_{st}}$ is either $\mathbf{O}$ or $\mathbf{R}$ and $\mathtt{t_{st}}$ is `void` if t is `void`,

or

$\mathtt{k_{st}}$ is $\mathbf{R}$ and $\mathtt{t_{st}} \leq_{isa} t$ if t is not `void`,

or

$\mathtt{k_{st}}$ is $\mathbf{T}$ and there exists $t_i \in q$ such that $\mathtt{t_{st}} \leq_{isa} t_i$.

For a well-formed method definition, we define the typing of the method invocation in the following way: $[m(e_1, ..., e_n)]^{\mathtt{A}} = \langle \mathtt{t_{st}}, \mathtt{k} \rangle$, where $\mathtt{k} = \mathbf{O}$ if $\mathtt{k_{st}}$ is either $\mathbf{O}$ or $\mathbf{R}$ (normal termination of the method) and $\mathtt{k} = \mathbf{T}$ if $\mathtt{k_{st}}$ is $\mathbf{T}$ (exception is thrown in the method). Thus, one needs to execute the body of the method to determine its typing.

A program $\mathtt{P(B)}$ satisfies a method definition $t \ m(t_1 \ p_1, ..., t_n \ p_n)$ **throws** $q \ St_1$ in the specification $\mathcal{PS}$ iff for any Σ where the method is invoked, any Σ-algebra $\mathtt{A}$, and any Σ-expressions $e, e_1, ..., e_n$ of their respective types, it holds:

$$\mathtt{A_m}|_{\Sigma_c} = \mathtt{A_{st}}|_{\Sigma_c} \text{ and } \mathtt{v_m} = \mathtt{v_{st}}.$$

Note that by taking the reduct of resulting algebras to the internal class signature Σ_c we require the equivalence of program state algebras and ignore any possible block local variable. The class method and constructor definitions and their satisfaction conditions are defined in a similar way. The program $\mathtt{P(B)}$ is a *model* of the specification $\mathcal{PS}$ if it satisfies the definition of each method and each constructor in $\mathcal{PS}$. Thus, the set of valid programs of a given program schema consists of those ones that are the models of the specification associated with the schema. In practice, a program specification (together with its schema) is represented by a program text and the set of its models consists of all properly compiled object programs.

6 Related Work

A large amount of work concerning formal semantics of Java is done in the research project LOOP (*Logic of Object-Oriented Programming*). In the framework of this project a type-theoretic memory model of a sequential Java program is presented in [3]. A companion paper [8] reports of modeling the inheritance semantics in higher-order logic. And finally it is investigated in [10] how monads in addition to coalgebras can be used in denotational semantics of Java statements. The peculiarity of our work is definition of the program state as a many-sorted algebra and interpretation of expressions and statements in terms of state updates.

The denotational approach is also followed in [1] where actually the semantics of a Java interpreter is defined.

Operational semantics of almost complete Java is defined in several works [7, 5, 13]. Once again, an abstract interpreter of a Java program is defined in each of these works. Consider as an example the

[1] In fact, a program specification may also contain static and dynamic initializers and the initializers of variables and constants (i.e., anything that can be present in a real program text), but we do not discuss them in this paper for brevity.

[2] Recall that any Σ_c-algebra is also a Σ_s-algebra for any other class s. Therefore, if Σ is the signature of some block where method m is invoked, then the reduct permits us to "forget" the local variables of the block and correctly interpret the symbols in the method body St.

definition of Java by Abstract State Machines in [13]. The state of the interpreter is defined there by a large number of so called dynamic functions (such a function can be different in different states). Each time a particular program construction is parsed, the corresponding dynamic functions are updated, and control passes to the "next task" defined by the dynamic function "task" (a counterpart of object program instruction counter). In contrast to this, we define an abstract program model and use its components as denotations for Java constructions. Our approach helps to avoid a large quantity of auxiliary functions representing the interpreter's state and thus make the semantics more concise and clear.

Several small sublanguages of Java are designed to experiment with semantics suitable for a theorem prover like Isabelle/HOL, while our primary goal is to experiment in providing an abstract model naturally simulating the Java program.

7 Conclusion and future work

We have introduced a formal model of a sequential Java program with states as algebras and have shown how some Java statements can be constructed and interpreted in terms of this model. The novelty of the approach consists in defining a specialized high-level operational machine whose components naturally represent program components. This provides us with a clear way of defining the semantics of an imperative language: basing on the type system of the language, define a program schema and an operational machine for this schema, then give the interpretations of expressions and statements in terms of this machine and use them for method specifications. Java has been chosen in this work to illustrate that even such complex features of the language as interfaces in addition to classes, implementation in addition to inheritance, object identity, late binding, etc. can be easily modeled with the use of simple algebraic notions including sets and functions.

Because of lack of space, several interesting aspects of the model have not been presented in this paper, local classes are among them. Particularly interesting is the task of using the presented approach for defining a state-based model of a parallel Java-program. We are proceeding with research in these directions.

Finally, we thank the anonymous reviewers for the insightful comments on the submitted text of the paper.

References

1. J. Alves-Foss and F. S. Lam. Dynamic Denotational Semantics of Java. *Formal Syntax and Semantics of Java*, LNCS, vol. 1523, 1999, pp. 201-240.
2. E. Asteziano and E. Zucca, *D-oids: a Model for Dynamic Data Types*, Mathematical Structures in Computer Science, 5(2), June 1995, 257-282.
3. J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A Type-Theoretic Memory Model for Verification of Sequential Java Programs *14th International Workshop on Algebraic Development Techniques (WADT'99)*, LNCS, vol. 1827, 2000.
4. Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer-Verlag, 2003.
5. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An Event-Based Structural Operational Semantics of Multi-Threaded Java. *Formal Syntax and Semantics of Java*, LNCS, vol. 1523, 1999, pp. 157-200.
6. J. Cosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification (Second Edition).* Addison-Wesley, 2000.
7. S. Drossopoulou and S. Eisenbach. Describing the Semantics of Java and Providing Type Soundness. *Formal Syntax and Semantics of Java*, LNCS, vol. 1523, 1999, pp. 41-82.
8. M. Huisman and B. Jacobs. Inheritance in Higher Order Logic: Modeling and Reasoning. *Theorem Proving in Higher Order Logics (TPHOL)*, LNCS 1869, 2000, p.301-319.
9. B. Jacobs. Objects and Classes, Coalgebraically. *Object Orientation with Parallelism and Persistence*, Kluwer Academic Publishers, 1996, pp. 83-103.
10. B. Jacobs and E. Poll. Coalgebras and Monads in the Semantics of Java. *Theoretical Computer Science*, 2002, 291/3 p.329-349.
11. K. Lellahi, A. Zamulin. *Implicit State Approach for Formalization of Sequential Java-like Programs*. Research report of LIPN 2002-10, Univ. Paris 13, 2002 (http://www.iis.nsk.su/persons/zamulin/LZ_report.zip).
12. K. Lellahi, A. Zamulin. *State-based Algebraic Semantics of Statements of Sequential Java*. Research report of LIPN 2003-03, Univ. Paris 13, 2003
13. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001.