

Towards an Algebraic Formalization of Sequential Java-like Programs with Implicit States

Kazem Lellahi¹, Alexandre Zamulin^{2*}

¹LIPN, 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

²A.P. Ershov Institute of Informatics Systems
Siberian Division of Russian Academy of Sciences
Novosibirsk 630090, Russia
zam@iis.nsk.su, fax: +7 3832 323494

The purpose of this work is to develop a formal framework that could be easily used for giving the formal semantics of typical features of a sequential Java-like program such as primitive types, classes and interfaces, inheritance and implementation relations, subtyping, instance and class variables and methods, overloading and overriding, late binding, object and array creation and initialization. Our approach “marries” conventional many-sorted algebras with Abstract State Machines (initially proposed in [2] as “evolving algebras”) so that a program state is represented as a many-sorted (state) algebra, each program construction has a counterpart in the state, and a state may be transformed into another state as a result of execution of certain state updating functions.

We develop a formal model of a Java-like program in several steps. We first define a Java-oriented *object data model* involving a *type system* and a *program schema*. Our type system includes basic (primitive) types, object types (classes and interfaces), array types, and the type `void`. A program schema extends the type system by declaring a collection of classes and interfaces and several relations such as:

- a binary acyclic inheritance relation on the set of interfaces,
- a binary inheritance relation on the set of classes forming a tree with root `Object`,
- a binary implementation relation on the sets of classes and interfaces

and four partial functions *var*, *const*, *con*, and *meth* respectively defining the sets of variables, constants, constructors and method signatures in the program schema. The relations and functions are defined in such a way that the model supports single class inheritance, multiple interface inheritance, multiple interface implementation and method overloading within a class.

Using inheritance and implementation relations, each schema generates in a natural way a partial order over types called the *subtyping relation* and denoted by \leq_{isa} . Basing on the subtyping relation, *hidden fields* and *overridden methods* are formalized and a *schema closure* is produced in such a way that each class and each interface is supplied with any inherited field or method that is not hidden or overridden.

Next, we define a *basic algebra* in the manner an algebra of a given signature is defined. Thus, a set of elements is associated with each basic type and a function is associated with every basic data type operation. A special set `Objd` is reserved for *object identities* (references in Java [1]). Having a program schema, a basic algebra B is extended to a state algebra A in the following way:

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

- a set of elements $|A|_c \subset \text{Oid}$ is associated with each object type ic so that if $ic' \leq_{isa} ic$, then $|A|_{ic'} \subseteq |A|_{ic}$;
- a *field function* $x_{ct}^A : A_c \rightarrow A_t$ is associated with each instance variable $x : t$ in class c , and a constant $y_{ct}^A : A_t$ is associated with each class (static) variable $y : t$ in class c ;
- a partial function $elem_{ct}^A : A_c, A_{int} \rightarrow A_t$ and total function $length_c^A : A_c \rightarrow A_{int}$ are associated with the respective operations *elem* and *length* in the data type $c = array(t)$.

Thus, the set of object identities of a superclass includes object identities of its subclasses. Moreover, the set of object identities of an interface includes the object identities of all classes that implement this interface. If an object identity o is in $|A|_c$, then c is a type of o . Furthermore, if there is no $c' <_{isa} c$ such that $o \in |A|_{c'}$, then c is the *most specific type* of o .

One state can be transformed into another by a *state modifier*, which is either a *function update* or *carrier update*. A function update is applicable to a *dynamic function*, which is generally either a variable name or the operation name *elem* in an array type. A function update either inserts an element in the domain of definition of a dynamic function or modifies the value of such a function at one point in the domain or removes an element from the domain. A carrier update transforms a state A into a new state by extending the set of object identities of a certain class or array type by a new element different from any object identity existing in A . The set of all possible update sets is used as the semantics of the data type **void**.

A *program* P includes:

- a set of state algebras called the *carrier* of P ;
- for each constructor in class c , a state-dependant function producing an update set;
- for each method in class c , a state-dependant function producing an update set if the method's result type is **void** and a pair (*update set, value of type t*) if the method's result type is t .

Finally, several rules for creating and interpreting expressions involving field and method names are introduced. Thus, if $x : t$ is a variable in class c and e is an expression of type c , then $e.x$ is an expression of type t called a *field access*. The expression is interpreted by the corresponding field function. If $m : r \rightarrow t$ is a method in class c , where $r = t_1 \dots t_n$, and e, e_1, \dots, e_n are expressions of types c, t_1, \dots, t_n , respectively, then $e.m(e_1, \dots, e_n)$ is a *transition expression* of type t called an *object method call*. The interpretation of a method call in a state A causes the invocation of the method associated with the most specific type of the object $\llbracket e \rrbracket^A$ (i.e., *dynamic (late) binding* is provided). If in class c there is a constructor with arity (t_1, \dots, t_n) and e_1, \dots, e_n are expressions of types t_1, \dots, t_n , respectively, then $new\ c(e_1, \dots, e_n)$ is a transition expression called *initialized object creation*. This expression is interpreted by an update set consisting of a sort update followed by a set of field access updates; which leads to the creation of an initialized object. Similar rules exist for initialized and noninitialized array creation.

References

1. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The JavaTM Language Specification (Second Edition)*. Addison-Wesley, 2000.
2. Y. Gurevich, *Evolving Algebras 1993: Lipary Guide, Specification and Validation Methods*, Oxford University Press, 1994.