A Programmer Friendly Modular Definition of the Semantics of Java

— Extended Abstract —

Egon Börger[1] and Wolfram Schulte[2]

[1] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
Email: boerger@di.unipi.it
[2] Universität Ulm, Fakultät für Informatik, D-89069 Ulm, Germany
Email: wolfram@informatik.uni-ulm.de

1 Introduction

Java is a general purpose, concurrent, class based, object-oriented language. It is described in the language reference manual [9] (LRM), which gives a description of the meaning of each programming construct of the language. This gives the programmer enough understanding for many purposes, but it is not adequate for use by an implementor or by someone who wants to design his program with mathematical rigor. Thus, a mathematical description of Java is needed.

Our goal is to develop a provably correct compiling specification – using the techniques developed by Börger, Durdanovic and Rosenzweig [5, 2] – for the compilation of Java programs on the Java virtual machine [13] (VM). The correctness proof of such a compiling specification is split into three tasks. First, we define a mathematical model for the semantics of Java, which closely follows the Java LRM, and captures the intuitive programmer's view of the meaning of a Java program. Similarly, we construct a mathematical model of the Java VM by formalizing the semantics of the Java VM instruction set. Finally, we link these two models by an equivalence proof of the semantics of an arbitrary Java program to the semantics of its compiled Java VM code. In this extended abstract, we sketch the first task, and give a mathematical definition of the *semantics* of Java. The full paper [6] appears as a chapter of the book *Formal Syntax and Semantics of Java (tm)*, edited by Jim Alves-Foss [1].

A Java program consists of a set of classes. In the use of a class there are three phases: parsing, elaboration, and execution. Parsing determines the grammatical form and yields an abstract syntax tree. Elaboration, the static phase, determines whether the class is well-typed and well-formed and records such information as annotations in the abstract syntax tree. Execution, the dynamic phase, loads, links, and executes the code of the class by evaluating expressions and modifying the memory. Corresponding to these phases, a full mathematical definition of Java needs a grammar, a static and a dynamic semantics. The grammar is well defined in [9]. Numerous authors have formalized the static semantics of sequential Java, in particular its type soundness, see [8, 14, 16, 15]. The dynamic semantics defined in these papers cover only a small structured sublanguage of sequential Java and do not consider the interaction of jump statements (like `break`), exception handling and concurrency, which we treat in full. We therefore concentrate in this paper on a complete but nevertheless transparent mathematical definition of the *dynamic* semantics of Java.

Our dynamic semantics is based on *Abstract State Machines* (ASMs), formerly called Evolving Algebras. They have been successfully used to model the dynamic semantics of many programming languages, including Prolog [4], Occam[3], C[11], C++[17] and Oberon[12]. An ASM describes a transition system with statics given by a first order signature and transition rules to transform structures. For a formal definition of ASMs we refer the reader to the Lipari Guide [10]. The key idea to use an ASM for the description of sequential Java can be sketched as follows: The initial state of the ASM is the given program, represented in the form of an annotated abstract syntax tree. The transition rules describe the program execution, namely how the task and the memory evolve. *Task* points to the part of the program currently in execution, i.e., a node in the abstract syntax tree, and may be seen as an abstract program counter. It is updated in each transition to point to the next instruction to execute, with reference to the control flow of the program. The memory represents the current values of the program's variables. The memory is updated according to the semantics of the currently executed program part. For example, if *task* denotes an assignment, then executing the assignment modifies the memory and sets *task* to the next instruction to be executed. The same idea also holds for concurrent Java. However, instead of a single ASM we use distributed ASMs, each working with its own task and local working memory. Additionally, we have a main memory, which is shared among the processes.

2 The Java Language Model

We factor the dynamic semantics of Java into *four ASMs*, each being a *conservative extension* of its predecessor. In the following subsections, we only discuss the particular conceptual view of each sublanguage. In ASMs the main concepts of the sublanguages are usually expressed by *dynamic functions*, which are updated during the execution of the transition rules.

2.1 The Imperative Core of Java

The basic ASM $\text{Java}_\mathcal{I}$ formalizes the sequential imperative core of Java, namely its statements and expressions over Java's primitive types.

Normal execution. Three dynamic concepts characterize the normal execution of $\text{Java}_\mathcal{I}$: an abstract program counter, called *task*, points to the current phrase to execute; an environment, called *env*, records the association between variables and their values;

the dynamic function *val* represents the (intermediate) values of expressions. These dynamic concepts are used to define the meaning of expression evaluation, assignments and conditional control transfer.

Abrupt completion. The normal control flow can be abrupted in Java$_{\mathcal{I}}$ by the restricted jump statements `break` and `continue`. Upon switch from normal to abrupted mode, the execution of one or more phrases may be terminated before all steps of their normal mode of execution have completed. For instance only part of the statements of a block are executed because a `break` or `continue` was encountered, which transfers control out of the block. In the later extensions abrupt completion can also be caused by returning from a method or by throwing exceptions. For a uniform formulation of these interrupts (which supports the process oriented view of interrupt description in the LRM) we introduce a dynamic function *mode* which records whether the current execution mode is normal or abrupted, due to a `break` or `continue` statement with a specific label. When execution is abrupted control transfers up the lexical nesting level to the nearest enclosing labeled statement that has a matching label. Execution then proceeds normally, either at the first phrase of the statement (in case of a `continue`) or at the next statement (in case of a `break`).

2.2 Adding Object-oriented Concepts

Java$_{\mathcal{O}}$, the ASM for the object-oriented sublanguage of Java, extends Java$_{\mathcal{I}}$ with classes, interfaces and objects, instance fields and instance methods, instance creation and method overriding, type casts and null pointers. We abstract from precompilable features by assuming the following: a) the static overloading of methods and fields is resolved, b) the subtype relation and the information where to locate methods in the inheritance hierarchy is precomputed.

To model the dynamic state of instances, we introduce a dynamic function *store*, which returns the value stored under a field name of an object, and a dynamic function *class*, which records the class to which an object belongs. The class of the object is set when a new instance is created, instance creation, instance field access and assignement operations initialize, read and write the store, respectively.

The introduction of methods has a fundamental effect on the definition of Java$_{\mathcal{I}}$'s dynamic functions. During execution of a Java$_{\mathcal{I}}$ program the environment and the value of an expression is computed and both may then be used in future computation steps. However, if in Java$_{\mathcal{O}}$ a method invokes itself recursively, the same statements and expressions are executed several times, thus 'overwriting' the environment as well as values of expressions. To handle this problem properly, we refine the two functions *val* and *env* by indexing each value of an expression and each variable in the environment with the calling depth of the code in execution. This corresponds to the usual technique of introducing stacks of environments and stacks of intermediate values, to handle recursion. We push the current task on the method call stack *ret* in order to resume execution at this point after the invoked method has finished. We extend the universe of the dynamic function *mode* by a new reason for abruption, since `return` statements abrupt a method's body completely. Execution continues normally, when control leaves the current method.

2.3 Adding Exceptions

Java$_{\mathcal{E}}$ adds exceptions to Java$_{\mathcal{O}}$. As is usual for dynamic semantics, we abstract from failures detected by the executing machine and consider only run-time and user-defined exceptions.

User-defined exceptions are thrown explicitly, using `throw` statements. Run-time exceptions are thrown, if certain semantic constraints on Java's operators do not hold. The introduction of exception handling provides a meaning for these up to now undefined phrases. In any case, when an exception is thrown processing completes abruptly. To model this we extend the universe of the dynamic function *mode* with exceptions. Exceptions first propagate through the lexical block structure of a Java method, and then up the method call stack to the nearest (dynamically) `catch` clause of a `try-catch` statement that handles the exception. A `catch` clause handles an exception, if the exception object is compatible with the declared type. When an exception is handled, the execution mode is set to normal mode again.

In situations where it is desirable to ensure that after one block of code another one is always executed, Java provides the `try-finally` statement. The `finally` clause is generally used to clean-up after the `try` clause. It is executed if any portion of the `try` block – regardless how it completes – is executed. In the normal case control reaches the end of the `try` block and then proceeds to the `finally` block. If control leaves the `try` block abruptly, the code of the `finally` block is executed before control transfers to the 'intended' interrupt destination. To support this kind of suspended interrupt, we introduce a mode stack, called *modes*. When entering a `finally` clause the current mode is pushed, on leave the mode is restored, provided the `finally` clause was not abrupted.

2.4 Adding Threads

Java$_{\mathcal{T}}$, the model for *multithreaded* Java, extends Java$_{\mathcal{E}}$ by supporting many different tasks working concurrently on shared main and local working memory. We consider Java's thread creation and destruction, its mechanisms for synchronizing the concurrent activity of threads using locks, and Java's waiting and notification mechanism for efficient transfer of control between threads.

The LRM specifies a memory model for shared memory multiprocessors that allows objects to reside in main and local working memory and presents rules (formalized as a particular event-structure by Cenciarelli et al. [7]) when a thread is permitted or required to transfer the contents of its working copy of an instance variable into the master copy in main memory or vice versa.

Shared Memory implementation. In order to separate Java's memory model from the semantics of Java's facilities for thread handling, we first build a model which uses only the main memory for storing objects (the latter is already modelled by the *store* function).

Distributed ASM's fit perfectly the concurrency of Java threads. Threads correspond exactly to code executing *agents* in distributed ASMs [10]. To model the autonomous activity of threads, we parameterize the dynamic functions *task* and its execution *mode*, the environment *env* and the intermediate values representing function *val* as well as the method invocation stack *ret* with its agent.

To synchronize threads Java uses locks uniquely associated with objects. When a `synchronized` statement is processed, the executing thread must grab the lock associated with the target reference before it can continue. Upon completion of the block the mechanism releases that very same lock. We use a dynamic function *grabbed_locks* to keep track of the dynamic nesting of synchronized statements by a particular thread. Since a single thread can hold a lock more than once, we have to define dynamic *lock_counter*s.

To aid communication between threads, each object also has an associated dynamic *wait_set*, a set of threads. Wait sets are used by the statements `wait` and `notify`. The `wait` method allows a thread to wait for a notification condition. Executing `wait` adds the current thread to the wait set for this object and releases the lock prior to waiting – which is reacquired to continue processing after the thread has been be notified by another thread.

Every thread can be in one of five states, modelled by the dynamic function *state*. The thread is either initialized, running, waiting, notified or exiting. The methods of classes `Thread` and `Object` set the thread's state appropriately.

The Java LRM – on purpose – leaves the scheduling strategy open by which threads are chosen for executing a lock. This is naturally reflected by the abstraction mechanisms of ASM's which allows us to have a not further specified function, which deals with threads that want to access a lock for the first time, and the ones who have been waken up and try to get their lock back.

Distributed Memory implementation. The former model is adequat for a Java implementation running on a single processor, where concurrent threads are implemented by time sharing. However, Java's memory model supports also shared memory multiprocessors. In these implementations each thread uses its own working memory as much as possible. Only when required, caused by the use of a `synchronized` statement or by a nonexistent value in the working memory, transfers between the main and working memory take place. The working memory is modeled by the dynamic function *cache* which returns for each thread the value stored under a field name of an object. We explicitly formalize the time delay between corresponding actions on the working and main memory. However, we do not model the transmission time as specified in Java' LRM, since it is irrelevant for the semantics.

3 Conclusion

The above abstract sketches the concepts needed for our mathematical model of the main parts of Java's dynamic semantics. We believe that our definition captures the dynamic semantics of Java precisely. It is easy to convince oneself by inspection of the transition rules, given in the full version of this paper [6], that they correctly formalize the corresponding verbal explanations of the language constructs in the Java LRM.

We proof several interesting theorems. We show, that the modells $\text{Java}_{\mathcal{O}}$, $\text{Java}_{\mathcal{E}}$ and $\text{Java}_{\mathcal{T}}$ are conservative extension of their respective predecessors. This means, that all facts which are valid for a predecessor sublanguage are mutatis mutandis also valid for the sublanguage. We show that the shared and the distributed memory implementations (with the same scheduling stragey) will produce the same states (modulo abstraction from the different memories) with the same best practice program. Best practice programs guarantee that access to shared variables is always synchronized. We proof, that exceptions are precise, that any thrown exception will be caught and that any `finally` clause will be executed, provided any portion of its `try` clause is executed – regardless how it completes.

Based on the ASM technology developed for the implementation for Prolog [5] and of Occam [2] we are currently working on refinements of our Java model to the level of abstraction of the Java VM. These refinements take advantage from the modular specification of orthogonal Java features as they appear in our models. The ASM we are developing for the Java VM will provide the basis for a rigorous mathematical analysis of general compilation schemes of Java programs into Java VM code including correctness proofs as developed in the work on Prolog and Occam.

References

[1] J. Alves-Foss, editor. *Formal Syntax and Semantics of Java(tm)*. Springer LNCS, to appear. 1998.

[2] E. Börger and I. Durdanovic. Correctness of compiling Occam to transputer code. *The Computer Journal*, 39, 1996.

[3] E. Börger, I. Durdanovic, and D.Rosenzweig. Occam: Specification and compiler correctness. Part I: Simple mathematical interpreters. In E.-R. Olderog, editor, *Proc. PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi)*, 1994.

[4] E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24, 1995.

[5] E. Börger and D. Rosenzweig. The WAM - definition and compiler correctness. In L.Plümer C.Beierle, editor, *Logic Programming: Formal Methods and Practical Applications*. Elsevier Science B.V./North-Holland, 1995.

[6] E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of java. In Alves-Foss [1].

[7] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. From sequential to multithreaded Java. In *AMAST'97: Algebraic methods and software technolgy*, To appear as Springer LNCS, 1997.

[8] S. Drossopoulou and S. Eisenbach. Java is type safe probably. In Alves-Foss [1].

[9] J. Gosling, B. Joy, and G. Steele. *The Java(tm) Language Specification*. Addison Wesley, 1996.

[10] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[11] Y. Gurevich and J. Huggins. The semantics of the C programming language. In *Computer Science Logic*, Springer LNCS 702 and 832, 1993.

[12] Ph. Kutter and A. Pierantonio. The formal specification of Oberon. *Journal of Universal Computer Science (J.UCS)*, 1997. available at the URL: http://hyperg.iicm.tu-graz.ac.at/jucs/.

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

[14] Th. Nipkow and D. von Oheimb. Java$_{light}$ is type-safe – definitely. In Alves-Foss [1].

[15] V. Saraswat. Java is not type-safe. manuscript, 1997.

[16] D. Syme. Proving Java type soundness. In Alves-Foss [1].

[17] Ch. Wallace. The semantics of the C++ programming language. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.