A Formal Executable Semantics for Java

Isabelle Attali, Denis Caromel, Marjorie Russo
INRIA Sophia Antipolis,
CNRS - I3S - Univ. Nice Sophia Antipolis,
BP 93, 06902 Sophia Antipolis Cedex - France
tel: 33 4 92 38 79 10
fax: 33 4 92 38 76 33
First.Last@sophia.inria.fr
http://www.inria.fr/croap/java

Abstract

Some of the main features of the Java language are that it is object-oriented and multi-threaded. This article presents a formal semantics of a large subset of Java, including inheritance, dynamic linking and multi-threading.
To describe the object-oriented features, we use a *big-step* semantics. The semantics of the concurrency is defined in a *small-step* semantics, using a structural operational semantics. This semantics is directly executable using the Centaur system.
An interactive programming environment, which provides textual and graphical visualization tools during program execution, is derived from this semantics.

1 Introduction

Both object-oriented and concurrent, the Java model features interrelated aspects that a re critical for the understanding of an application: objects, static variables, threads, locks, etc. In this article we consider a large subset of Java including primitives types, classes, inheritance, instance variables and methods, class variables and methods, interfaces, overloading, shadowing, dynamic method binding, object creation, threads creation and concurrency.

Our semantics definition is based on the informal Java specification of Sun [12]. We adopt a *big-step* semantics to describe the object-oriented features and the inheritance. To specify the semantics of the multi-threading we use *Structural Operational Semantics* [17]. More specifically, we use the

Natural Semantics [14] within the Centaur system [6], and the Typol formalism [8] which provides us with executable specifications.
The outcome of such an approach is twofold: (i) providing a programming environment in order to formally study concurrent object-oriented programming and to understand Java programs behavior; (ii) having a formal specification of the language from which we will check its soundness with respect to the compiler and also verify a set of properties expressing a security policy.

The next section of this paper is a discussion of related work. Section 3 presents the Centaur system and the Typol formalism. Section 4 focuses on the Java semantics definition. From this definition, graphical and interactive visualization tools are derived (Section 5). Finally, Section 6 briefly discusses our contribution and outlines future work.

2 Related Work

Java semantics is an active research area. This section details the different followed approaches and their goals.

This first important research domain is the proof of the soundness of the Java type system. Indeed Drossopoulou and Eisenbach [9], [10], [11] (the most recent version), and Syme [19] are specifying the semantics of different Java subsets in order to prove the type soundness in these subsets. Drossopoulou and Eisenbach are working in the three cited papers on a large sequential subset of Java and prove that program execution preserves the types by means of a subject reduction theorem. Directly related to this work, Nipkow and Oheimb [16] define and prove properties of the Java Light subset in the theorem prover Isabelle/HOL.

These soundness results apply to the language semantics, but not to any particular implementation of Java, nor to the Java Virtual Machine (JVM). So another approach is to work at the byte-code level on the JVM. Qian [18] has specified a subset of the JVM instructions for objects, methods and subroutines. He describes the runtime behaviors of the instructions in relevant memory areas as state transitions and most structural and linking constraints on the instructions as a static typing system. Börger and Schulte [4] define the JVM in order to prove the correctness of Java compilation. Jensen, Le Metayer and Thorn in [13] formalize dynamic class loading mechanisms in the JVM and study some security properties of Java.

Another important goal is to specify Java semantics in order to formalize the language. In [11], Drossopoulou and Eisenbach define an operational semantics for a sequential subset of Java which includes primitive types, classes and inheritance, instance variables and instance methods, interfaces, shadowing of instance variables, dynamic method binding, object creation, arrays, exceptions. Börger and Schulte [5] also give a dynamic semantics via successive subsets of Java but do not treat class loading, Java packages, names visibility.

In this article, we are defining a dynamic semantics of the language; we are not concerned with typing (we assume our programs are correctly type checked). This specification is on one side executable and on the other side, it will be the basis for formal verification of Java programs.

3 Natural Semantics Specifications

We use the Centaur system [6] as a formal tool to model and implement the dynamic semantics of the Java language and namely the Natural Semantics [14]. This section describes briefly the Centaur system and the Typol formalism [8].
The Centaur system is a generic programming environment: from the specifications of the syntax and the semantics of a given language, one can automatically produce a syntactic editor and semantics tools (for example type checkers, interpreters) for this language. This system has already been used to specify the semantics of the following languages: Sisal [3], Eiffel [1], Eiffel// [2], etc.
The specification of syntactic aspects includes the concrete and abstract syntax of the language. From this specification (written in Metal [15]), one can derive a parser that transforms the textual form of a program (a source file) into a structural representation (an abstract syntax tree that belongs to the formalism so defined). Every structured object is represented within the system as an abstract syntax tree.
Semantic aspects in the Centaur system are handled by the Typol formalism, which is an implementation of the Natural Semantics approach. The Typol formalism is based on a logical framework, as advocated by Plotkin [17], which makes it highly declarative and expressive.
A Typol specification is represented by an unordered collection of inference rules. Each inference rule is composed of a finite set of premises (which is empty for an axiom) and a conclusion. Figure 1 presents a Typol rule which specifies the last step of the assignment in Java. The premises (above the dash line in Figure 1), and the conclusion of a rule (below the dash line), are relations represented by sequents in the Gentzen natural deduction style.

```
AssignValueToASimpleVariable:
    --RC: Assigns a value to a variable.
    --
    --CS: TVIdent=TVValue;
    assign(ObjL1, ClVarL1, Env1, ObjId, TVIdent, TVValue
            ->
            ObjL1_1, ClVarL1_1, Env1_1)
    ----------------
    ObjL1, ClVarL1, Env1, _, ObjId, _
        |- binaryAssign(TVIdent, assignment(), TVValue) ->
            ObjL1_1, ClVarL1_1, Env1_1, TVValue, inst_l[],
            runnable() ;
```

Figure 1: A Typol Rule for the Assignment.

The object languages are manipulated via their abstract syntax. A sequent expresses the fact that some hypothesis (the term list in the left hand side of the sequent symbol) is needed to prove a particular property, about an abstract syntax term called the subject. In Figure 1, the subject of the rule is the abstract syntax term *binaryAssign(TVIdent, assignment(), TVValue)*. Sequents are typed, according to the syntactic nature of their subject; this type is defined with a judgment as shown in Figure 2. This Figure shows

```
--JC: Object list, Class variable list, Execution environment
--JC: (parameters and local variables), Current executing
--JC: thread reference, Current object reference, Execution
--JC: mode |- Expression to evaluate -> updated object and
--JC: class variable lists, updated environment, new expression
--JC: (which will be evaluated later), instruction list to
--JC: execute at the next execution step of the current
--JC: executing thread, new status of the current executing thread.
judgement
    Object_L, C_Var_L, Env, O_Id, O_Id, Identifier
        |- Comp_Exp ->
            Object_L, C_Var_L, Env, Comp_Exp, Inst_L, T_Status;
```

Figure 2: Example of a Typol Judgment.

the Typol judgment associated to the previous rule as shown in Figure 1.
Typol rules indicate how a sequent may be deduced from other sequents. Typol rules may be structured into sets that deal with the same object (for example the evaluation of an expression of the considered language). Within a set, a premise sequent of a rule refers to the same set unless another set is explicitly indicated by a named sequent (as in Figure 1 with the *assign* premise).

4 Java Semantics

This section presents our transition system. Our semantic definition is based on a Java abstract syntax and uses semantic structures which describe the manipulated objects and threads. The Typol rules presented in this section are the real ones (no simplification). They are commented in order to be easily understandable.

4.1 Syntactic Features

Our Java abstract syntax definition is composed of 140 operators and 65 types. As an illustration, we give in Figure 3 the abstract syntax tree corresponding to the expression: `Obj.m_name(Expr1, Expr2)`. Operator names are given in lower-case, while type names start with a capital letter. This syntactic definition is used in the semantic specification.

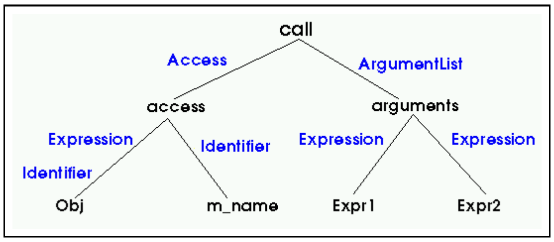

Figure 3: Abstract Syntax for Method Call.

4.2 Semantic Structures

During execution, a Java program creates, uses, and updates objects and threads. The result of the semantic evaluation of a Java program is a list of objects and threads, which denotes the behavior (the meaning) of the program. The chosen semantic structure is therefore a list of objects and threads (see Figure 4). In case of a simple object (not a thread), the only difference is that the activity is *nil*. The activity is composed of a status and a continuation, which is made of:

- a thread identifier,
- the name of the current method,
- an instruction list (language statements as well as closures for method calls);

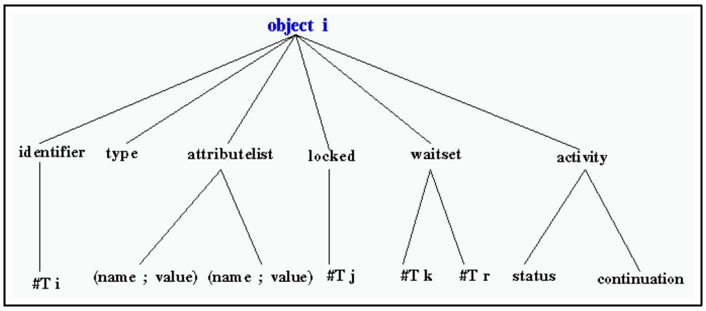

Figure 4: Abstract Syntax for Objects and Threads.

- an execution environment made of parameters (name-value pairs) and local variables (name-value pairs).

The next paragraph shows the module organization of our semantics.

4.3 Semantics Modules

The semantic specification is composed of 400 inference rules describing an operational semantics of Java. These rules are both highly declarative and executable. They are organized in modules as shown in Figure 5, which enhances design, readability, and ease of debugging.

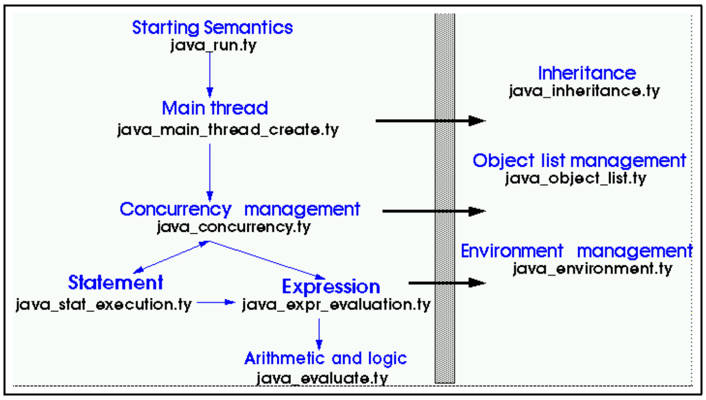

Figure 5: Semantic Modules.

The semantics of inheritance and dynamic binding (e.g. `java_inheritance.ty` and `java_object_list.ty`) is expressed in Natural Semantics. Although,

the modules describing the actual execution of statements (loops, method calls, assignment, etc) are expressed in Structural Operational Semantics style (SOS) [17] (especially concurrent features, e.g. `java_stat_execution.ty`, `java_expr_evaluation.ty`).
Natural Semantics (*big-step* semantics) is opposed to SOS (*small-step* or *transitional* semantics) in the sense that intermediate steps of a program execution are hidden in a big-step semantics. These two styles of semantic description cohabit well in the logical framework of the Typol formalism. This enables us to mix large-step and small-step semantics in our specification of a formal executable semantics for Java.

4.4 Object-Oriented Features

The object-oriented features such as object creation, subclasses and inheritance are specified in big-step semantics. As an example Figure 6 shows how the attributes list of a given type can be obtained. The first premise of the

```
Main_rule:
    --C : Gets the attribute list of a class.
    --C : "class" : gets the class declaration.
    --C : "make_local_attribute_list" : determines the
    --C : local attribute list.
    --C : "make_inherited_attribute_list" : determines
    --C : the inherited attribute list.
    class(|- ClName -> ClDec)  &  make_local_attribute_list(|- ClDec : LocAttrL)
    &  make_inherited_attribute_list(ClName, LocAttrL |- ClDec -> InhAttrL)  &
    appendtree(LocAttrL, InhAttrL, AttrL)
    ----------------
    |- ClName -> AttrL ;
```

Figure 6: Formal Definition of Attribute List.

rule gets the attribute list of the current class and the second one gets the inherited attribute list. The result list is the concatenation of these two lists.

4.5 The Transition System

Our semantic specification of concurrency aspects can be described as a transition system which, for a given program P, maps configurations to new configurations. A configuration is composed of the current object list ('ObjL1' in the example rules: Figure 7, 8, and 9), the current class variable list ('ClVarL1'). The initial configuration is composed of an object list made of only one thread: the main thread which will execute the *main* method, and of the static variable list obtained by the class loading. Figure 7 shows the

```
initial_configuration_creation:
   --C : "notify_exec" : waits a signal from the programmer.
   --C : "class_loader" : loads all the program classes.
   --C : "class_name" : obtains the name of the class
   --C : which contains the "main" method.
   --C : "create_main_thread" : creates the semantic
   --C : structure corresponding to the main thread.
   --C : "store_execthread" : saves the current executing thread.
   --C : "execute_one_step" : starts the program interpretation.
   --C : "store_comp_unit" : saves the program tree.
   notify_exec()  &  class_loader(typeDeclarations[Cl1.ClL1] -> ClVarL1)  &
   class_name(Cl1 -> Cl_Name)  &
   create_main_thread(Cl_Name, object_l[] -> ObjL1)  &
   store_execthread(o_id(true(), o_ident 0))  &
   execute_one_step(ObjL1, ClVarL1 -> ObjL1_1, ClVarL1_1)
   ----------------
   |- compilationUnit(A, B, typeDeclarations[Cl1.ClL1]) -> ObjL1_1, ClVarL1_1 ;
            provided
               store_comp_unit(compilationUnit(A, B, typeDeclarations[Cl1.ClL1]));
```

Figure 7: Initial Configuration.

corresponding Typol rule. We simulate concurrency by interleaving between program threads. The transitions between configurations are specified with rules which describe one step of execution of a given thread. These rules are of the form:

$$< ObjL1, ClVarL1 > \rightarrow < ObjL1\_1, ClVarL1\_1 >$$

which is interpreted as follows: *A system in a configuration* $<$*ObjL1, ClVarL1*$>$ *performs an execution step and changes its configuration into* $<$*ObjL1_1, ClVarL1_1*$>$. Execution is therefore a sequence of transitions as shown in Figure 8. The rule on top of this figure is the general transition rule. This rule determines the thread which is going to execute itself and then performs

```
Transition_Rule:
   determine_executing_thread(ObjL1, ClVarL1 -> ObjL1_1, ClVarL1_1, OThId1)
   &  exec_thread(ObjL1_1, ClVarL1_1, OThId1 -> ObjL1_2, ClVarL1_2)  &
   (ObjL1_2, ClVarL1_2 -> ObjL1_3, ClVarL1_3)
   ----------------
   (ObjL1, ClVarL1 -> ObjL1_3, ClVarL1_3) ;

Last_Transition_Rule:
   --C : All threads dead : end of the program interpretation.
   all_threads_dead(ObjL1 -> true())
   ----------------
   (ObjL1, ClVarL1 -> ObjL1, ClVarL1) ;
```

Figure 8: Transition System Rules.

an execution step of the given thread. The bottom rule of Figure 8 is applied when all threads are dead. It has the form:

$$< ObjL1, ClVarL1 > \rightarrow < ObjL1, ClVarL1 >$$

Naturally, if neither of these two rules (Figure 8) applies itself, a deadlock is detected and the program is stopped with an error message. An example of

```
Assignment_end:
   --CS : Ident1=Value1;
   --C : "assign" : makes the assignment.
   assign(ObjL1, ClVarL1, Env1, ObjId1, Ident1, Value1
            ->
            ObjL1_1, ClVarL1_1, Env1_1)
   ----------------
   ObjL1, ClVarL1, Env1, OThId1, ObjId1, Mode
      |- assign_comp(void(), Ident1, Value1) ->
           ObjL1_1, ClVarL1_1, Env1_1, Value1, inst_l[], runnable() ;

General_rule_of_assignment:
   --CS : ObjId2.Ident1 = Expr1;
   --C : "eval_expr" : makes one step of evaluation
   --C : of the expression 'Expr1'. Returns a more
   --C : simple expression 'Expr1_1' and an instruction list.
   eval_expr(ObjL1, ClVarL1, Env1, OThId1, ObjId1, Mode
                |- Expr1 -> ObjL1_1, ClVarL1_1, Env1_1, Expr1_1, InstL1, ThStatus)
   &  appendtree(InstL1, inst_l[assign_comp(ObjId2, Ident1, Expr1_1)], InstL1_1)
   ----------------
   ObjL1, ClVarL1, Env1, OThId1, ObjId1, Mode
      |- assign_comp(ObjId2, Ident1, Expr1) ->
           ObjL1_1, ClVarL1_1, Env1_1, Expr1_1, InstL1_1, ThStatus ;

Assignment_begin:
   --CS : Ident1 = Expr1;
   --C : "eval_expr" : makes one step of evaluation
   --C : of the expression 'Expr1'. Returns a more
   --C : simple expression 'Expr1_1' and an instruction list.
   eval_expr(ObjL1, ClVarL1, Env1, OThId1, ObjId1, Mode
                |- Expr1 -> ObjL1_1, ClVarL1_1, Env1_1, Expr1_1, InstL1, ThStatus)
   &  appendtree(InstL1, inst_l[assign_comp(void(), Ident1, Expr1_1)], InstL1_1)
   ----------------
   ObjL1, ClVarL1, Env1, OThId1, ObjId1, Mode
      |- binaryAssign(Ident1, assignment(), Expr1) ->
           ObjL1_1, ClVarL1_1, Env1_1, Expr1_1, InstL1_1, ThStatus ;
```

Figure 9: Assignment Rules.

interleaving treatment is given with the three rules shown in Figure 9 which describe the semantics of a simple assignment of the form Ident1=Expr1.

5 From Semantics to Visualization

The Centaur system permits, from a set of formal specifications (both syntactic and semantic), the derivation of a dedicated minimal programming environment. From the dynamic semantics specification presented in the latter section, we derive an interpreter which takes as input a syntactically correct and well typed Java program (in fact, an abstract syntax tree).

This section presents a global view of our environment and then evokes some aspects of the programming animation. Figure 10 presents a global view of our environment during execution of the Producer-Consumer program [7]. Besides the program itself (top left window), there are two synthetic views: object and thread status (top middle), thread stacks and object activations (top right). The object list is first presented in a textual form

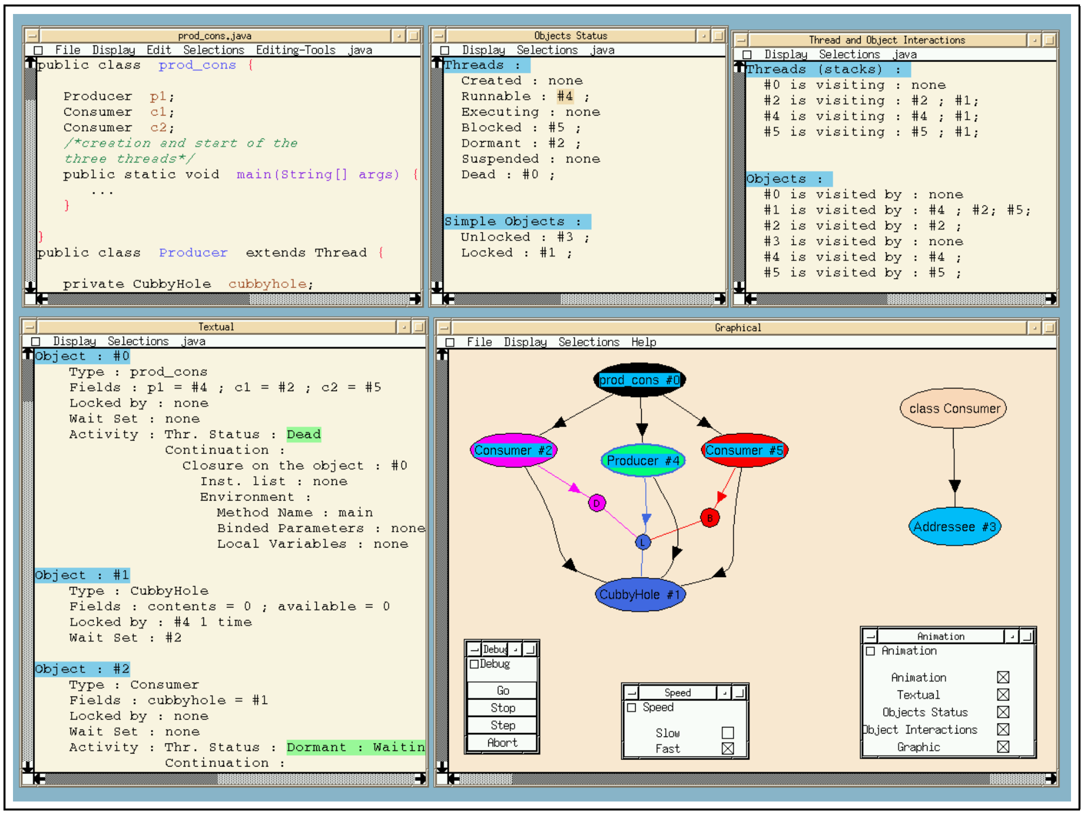

Figure 10: Global view of our graphical environment.

(bottom left) where a detailed view of objects and threads is given, including the activity of each thread (stack or continuations). The graphical view (bottom right) features the topology of the object graph, the threads status (from top, and left to right: dead, dormant, executing, blocked), together with the visualization of locks (object 1 is locked by object 4). A control panel (inside the graphical window) provides for a step-by-step execution.

Our environment provides animation to visualize objects *during* program execution, and so have a better understanding of the behavior of the program. For that purpose, the semantics is equipped with notifications. On some appropriate semantic rules, when successfully applied (proved), the notification (if it exists) is triggered and the visualization engines become aware of some modification in the semantic structures. Altogether, less than 10 semantic rules needed to be equipped with such notifications. In the case of the graphical server, the rules where we need to send notifications are the following: object and thread creation, thread status change (runnable, executing, locked, etc.), object status change (locked, unlocked), method calls and returns, assignments. Another critical aspect is incrementality: in order to have efficient and quality visualizations (avoiding flashing) the changes are done in an incremental manner in both views.

6 Conclusion

In this paper we presented a general view of our semantic definition of a large subset of Java and briefly describe the programming environment we derive from this specification.

The semantic specification, using both a small-step and a big-step style (thanks to the Typol logical framework which enables to mix these two styles), includes primitives types, classes, inheritance, instance variables and methods, class variables and methods, interfaces, overloading, shadowing, dynamic method binding, object creation, threads creation and concurrency. From this specification we derive a graphical programming environment. This environment is animated and interactive, it includes visualization of the objects topology during program execution.

The semantic definition is still under progress. The exceptions specification is on going and future work is first to extend the covered subset of Java to arrays and packages. In the same time we will work on improving the environment visualization tools: we particularly want to develop a more synthetic graphical view in order to be able to scale our environment to larger applications. Our final goal is then to use this semantic specification in order to perform formal verification of Java programs.

References

[1] I. Attali, D. Caromel, and S. O. Ehmety. A Natural Semantics for Eiffel Dynamic Binding. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(5), Novembre 1996.

[2] I. Attali, D. Caromel, S. O. Ehmety, and S. Lippi. Semantic-based visualization for parallel object-oriented programming. In *Proc. OOPSLA'96 (Object-Oriented Programming: Systems, Languages, and Applications)*, volume 31, number 10. ACM Press, Sigplan Notices, Oct 1996.

[3] I. Attali, D. Caromel, and A. Wendelborn. A Formal Semantics and an Interactive Environment for Sisal. In *Tools and Environment for Parallel and Distributed Systems*. Kluwer Academic Publishers, 1996.

[4] E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In *23rd International Symposium on Mathematical Foundations of Computer Science*, LNCS. Springer-Verlag, 1998. to appear.

[5] E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In *Formal Syntax and Semantics of Java*. Springer-Verlag, LNCS, 1998. to appear.

[6] P. Borras and et al. Centaur: the System. In *SIGSOFT'88 Third Annual Symposium on Software Development Environments*, Boston, 1988.

[7] M. Campione and K. Walrath. *The Java Tutorial (Object-Oriented Programming for the Internet)*. Addison–Wesley, 1998.

[8] T. Despeyroux. Typol: A Formalism to Implement Natural Semantics. Research Report 94, INRIA, 1988.

[9] S. Drossopoulou and S. Eisenbach. Is the Java Type System Sound? In *4th Int. Workshop Foundations of Object-Oriented Languages*, 1997.

[10] S. Drossopoulou and S. Eisenbach. Java is Type Safe - Probably. In *ECOOP'97*, LNCS 1241, pages 389–418. Springer Verlag, January 1997.

[11] S. Drossopoulou and S. Eisenbach. Towards an Operational Semantics and Proof of Type Soundness for Java. In *Formal Syntax and Semantics of Java*, LNCS. Springer-Verlag, 1998. to appear.

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

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

[14] G. Kahn. Natural Semantics. In *Proc. of Symposium on Theoretical Aspects of Computer Science, Passau, Germany, LNCS 247*, 1987.

[15] G. Kahn, B. Lang, and B. Melese. Metal: a Formalism to Specify Formalisms. In *Science of Computer Programming, volume 3*, North-Holland, 1983.

[16] T. Nipkow and D. Von Oheimb. Java Light is Type Safe - Definitely. In *25st ACM Symp. Principles of Programming Languages*, 1998.

[17] G. D. Plotkin. A Structural Approach to Operational Semantics. Report, DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[18] Z. Qian. A Formal Specification of the Java Virtual Machine Instructions for Objects, Methods and Subroutines. In *Formal Syntax and Semantics of Java*. Springer-Verlag, LNCS, 1998. to appear.

[19] D. Syme. Proving Java Type Soundness. Technical report 427, University of Cambridge Computer Laboratory, 1997.