

# A Formally Verified Calculus for Full Java Card

Kurt Stenzel

Lehrstuhl für Softwaretechnik und Programmiersprachen  
Institut für Informatik, Universität Augsburg  
86135 Augsburg Germany

email: stenzel@informatik.uni-augsburg.de

**Abstract.** We present a calculus for the verification of sequential Java programs. It supports all Java language constructs and has additional support for Java Card. The calculus is formally proved correct with respect to a natural semantics. It is implemented in the KIV system and used for smart card applications.

## 1 Introduction

The Java language has received a formal treatment for a couple of years now, see e.g. [1][2][9][11][15][24]. While issues like type safety have been solved [22], most people perceive that the problem of proving Java programs correct is not yet solved satisfactorily. Java combines a number of features that pose difficulties for a verification calculus: objects and object updates, static initialization, a class hierarchy and inheritance, exceptions, and threads. Additionally, Java has a large number of language constructs. It is not clear how to integrate these features into a proof system that is fast and (relatively) simple to use.

However, that may depend on the problem domain, i.e. what kind of Java programs are verified. Consider smart cards: they are distributed in large numbers (in credit and health cards or mobile phones), and they are usually security critical (money or privacy is lost if the security is broken). And, they can be programmed in Java Card, but this is difficult and error prone. So smart cards are a very useful area for Java program verification, and, as it turns out, a tractable one.

In this paper we present a calculus for sequential Java that is formally proved correct and implemented in the KIV system [12][3][23]. Section 2 describes the intended problem domain, smart card programming. Section 3 introduces the semantics that is the basis for the calculus and the correctness proofs in section 4. All of them are only presented in examples due to the complexity of Java. Section 5 shows an example proof, section 6 compares with other Java proof systems, and section 7 summarizes.

## 2 Java Card

Java Card [18] is a variation of Java that is tailored for smart cards. A smart card is a plastic card containing a small processor. Smart cards are used in

mobile phones, as credit cards, for authentication etc. These processors have very limited speed and memory. Typically they are 8-bit processors running at 5 Mhz. The most advanced cards have 128 KBytes ROM, 64 KBytes EEPROM, and 5 KBytes RAM. Often a cryptographic co-processor is included (e.g. with 1024 Bit RSA and 3-DES). This means that the complete JVM and all applications must fit into some dozen kilobytes of memory.

Java Card has the same language constructs (i.e. expressions and statements) as Java, but omits all features that make the JVM big and slow. It does not support threads, garbage collection, streams, floating point arithmetic, characters and strings, and integers. Essentially, Java Card is sequential Java with fewer primitive data types and a much smaller API. However, Java Card programs often use expressions like postfix increment or compound assignments, and rely heavily on byte and short arithmetic. This means that these constructs must be supported and modeled precisely.

An interesting smart card application is the storage of electronic tickets (e.g. railway tickets). If the smart card contains only genuine tickets they can be inspected and invalidated offline, i.e. without access to a central data base. This goal (only genuine tickets) can be achieved by access control and cryptographic protocols. In this scenario the smart card owner must be considered hostile because he has an interest to load forged or copied tickets onto the card. So the owner may not modify or read the code or data (the code itself is not secret, but cryptographic keys are). This is achieved by loading the program in a secure environment and by setting suitable access rights. The Java Card program on the smart card manages the tickets, and performs the steps of the cryptographic protocols. A typical program (including loading of tickets via internet and offline transfer of tickets) has about 1200 lines of code and about 40 methods. This does not include cryptographic operations like RSA encryption, digital signatures etc. that come from a predefined library (and are defined in the Java Card API).

### 3 A Natural Semantics for Sequential Java

Several different Java semantics exist, e.g. [5][8][26]. They differ in the used formalism, the number of supported Java constructs, in the assumptions made, and in the level of detail for auxiliary operations.<sup>1</sup> However, the basis for every formal semantics is the Java Language Specification [19]. Different formalisms can be used for the semantics: denotational semantics, abstract state machines (ASMs), Structural Operational Semantics (SOS), natural semantics, and others. (There is also the further notion of small-step and big-step semantics.) Maybe the “best” semantics is one that is closest to the informal descriptions in the Java Language Specification, so that people who are not experts in formal methods have a possibility to understand it. For this reason we choose a natural (big-step) semantics.

---

<sup>1</sup> Perhaps one should distinguish between a *formal* semantics – one written in a logic with a precise semantics and syntax, parsed by a computer, and a *mathematical* semantics that uses mathematical notation, but does not adhere to a precise syntax.

The semantics of a Java statement is a ternary relation: a statement  $\alpha$  transforms an initial state  $st = v \times h$  consisting of a variable mapping  $v$  (as in classical predicate logic) and a heap  $h$  into a final state  $st' = v' \times h'$  consisting of a new variable mapping  $v'$  and a new heap  $h'$ ,  $st \llbracket \alpha \rrbracket_{tds} st'$ . In SOS this is often written as  $\langle \alpha, st \rangle \longrightarrow st'$ . If  $\alpha$  does not terminate there is no final state  $st'$ . If  $\alpha$  terminates there is exactly one new state  $st'$ , because Java Card is deterministic. (The semantics can also be viewed as a transition system that (stepwise) transforms states.) The relation  $\llbracket \cdot \rrbracket_{tds}$  is annotated with the list of class and interface (type) declarations  $tds$  that comprise the Java program. The semantics of a Java expression  $e$  additionally includes a result value,  $st \llbracket e \rrbracket_{tds} st' \times val$  (or written as  $\langle e, st \rangle \longrightarrow \langle val, st' \rangle$ ). The variable mapping  $v$  contains values for the local variables of the program; they are identified with logical variables. The heap  $h$  contains the object fields (indexed by a reference and a field name) and a ‘reserved’ reference with ‘reserved’ fields for static fields, the initialization state of classes, and whether currently a jump (*abrupt transfer of control*, e.g. when an exception is thrown) happens or whether evaluation proceeds in the normal manner. No other ingredients are needed to describe the state. The heap is specified algebraically, so that the semantics of Java is defined relative to all models of this specification. Evaluation of statements and expressions is described inductively by reduction rules, the semantics is then the smallest set of triples  $\llbracket \cdot \rrbracket_{tds}$  closed under the rules for the relations. The construction guarantees that the smallest set is well defined (no negative occurrences of the relations).

$$\frac{h[mode] \neq normal}{v \times h \llbracket e \rrbracket v \times h \times \perp} \quad (1) \qquad \frac{h[mode] = normal}{v \times h \llbracket l \rrbracket v \times h \times l_v} \quad (2)$$

$$\frac{v \times h \llbracket e \rrbracket v_0 \times h_0 \times r_0 \quad r_0 = null \vee h_0[mode] \neq normal}{v_0 \times h_0 \llbracket \mathbf{throw\ new\ NullPointerException}(); \rrbracket v_1 \times h_1} \quad (3)$$

$$\frac{v \times h \llbracket e.f \rrbracket v_1 \times h_1 \times \perp}{v \times h \llbracket e.f \rrbracket v_1 \times h_1 \times \perp} \quad (4)$$

**Fig. 1.** Semantics rules for jump (1), literal (2), and instance field access (3, 4).

Figure 1 shows four simple rules. The jump rule (1) states that in case of an abrupt transfer of control (when an exception is thrown, or a **return**, **break**, or **continue** statement is encountered) any expression is skipped. The heap  $h$  stores Java values that can be accessed with keys (normally a pair of an object reference and a field name, but there are some special keys and special values: one special key  $mode$  is used to record the evaluation mode, and  $h[mode]$  looks up the value in the heap. So  $h[mode] \neq normal$  means that no normal evaluation takes place. The expression is not evaluated, the variable mapping and heap

remain unchanged, and a dummy value  $\perp$  is returned. A literal (2) is evaluated if the mode is normal. The value is the literal evaluated under the current variable mapping  $l_v$  (in our case literals may contain algebraic expressions with variables), everything else remains unchanged. In case of an instance field access (3) a `NullPointerException` is thrown if the invoking expression is null (if initially the mode was not normal the evaluation of  $e$  is skipped due to the jump rule; if during evaluation of  $e$  an exception occurs the `throw` statement will be skipped due to the throw rule). Otherwise the value is looked up in the heap using the computed reference  $r_0$  together with the field name ( $h_0[r_0 \times f]$ ) (4).

We consider another example: the instance method invocation (see Fig. 2). It has three rules, the first in case the invoking expression is null, the second for exceptions during evaluation of the arguments or the body, or in case the body completes without a `return` (this is possible for `void` methods). Only the third rule is shown. First, the invoker and the arguments are evaluated, then the arguments are bound and the method body  $\alpha$  is evaluated. If the body completes with a `return` ( $is\_return\_mode(h_{n+1}[mode])$ ) the mode is set back to normal ( $h_{n+1}[mode, normal]$ ) and the value ( $h_{n+1}[mode].val$ ) returned. The main point is that the real problem of the rule, the dynamic method lookup, is completely hidden in the definition of *getMethod* (an algebraically specified function). Every formal Java semantics must specify precisely how this method lookup works (e.g. what happens in case of a cyclical class hierarchy). The same holds for all other definitions. So the semantics is much more (complicated, longer) than just the rules. This makes it very difficult to compare two semantics that are formally defined in different proof systems.

$$\begin{array}{c}
v \times h[e]v_0 \times h_0 \times r_0 \quad v_0 \times h_0[e_I]v_1 \times h_1 \times r_1 \dots v_{n-1} \times h_{n-1}[e_n]v_n \times h_n \times r_n \\
r_0 \neq null \wedge h_n[mode] = normal \\
\frac{(v_n)_{x_1, \dots, x_n, this}^{r_1, \dots, r_n, r_0} \times h_n[\alpha]v_{n+1} \times h_{n+1} \quad is\_return\_mode(h_{n+1}[mode])}{v \times h[e.m(e_1, \dots, e_n)]v_n \times h_{n+1}[mode, normal] \times h_{n+1}[mode].val}
\end{array}$$

$m(x_1, \dots, x_n) \{ \alpha \} = \text{getMethod}(\text{classOf}(r_0), m, tds),$

**Fig. 2.** The third rule for instance method invocation.

All in all 24 expressions and 23 statements are specified. In other semantics the count may differ due to decisions what are considered different constructs. Seven of the 23 statements are not part of Java. They are introduced because they are used in the calculus. `static` and `endstatic` are used for the initialization of super classes and to capture exceptions during initialization; `target` and `targetexpr` define a target for `return` statements, `finally` and `endfinally` are used to describe the beginning and end of a finally block, and `catches` holds a list of catch clauses. The full semantics of the language constructs is described in 123 rules. This is a large number, but every rule describes exactly one case that may occur during evaluation. About 50 nontrivial definitions with several

hundred axioms are specified algebraically, plus the heap, and the primitive types for bytes, shorts, and the bitwise integer operations. A pretty printing has more than 50 pages. Obviously it is a nontrivial task to make sure that the specification captures the intended Java semantics in every little detail.

*Comparison to other formal semantics.* David von Oheimb [25] defines a natural semantics for a subset of Java (9 statements and 12 expressions) in Isabelle with a deep embedding using higher order logic. It is interesting to note that the notion of a ‘state’ is more complicated than our state, even though a reduced language is used. The state contains a store for local variables, and an explicit exception status; the environment contains not only the Java classes, but also local variables. Börger and Schulte [5] (later expanded in [24]) give an ASM semantics that also includes threads, but is not formalized in a proof system. Usage of ASMs requires more notational overhead, because stacks for local variables and a pointer into the program code is needed to model program evaluation. Both use a very short notation that is sometimes difficult to read, and both aim more at meta properties of the Java language than at source code verification of concrete programs. Therefore they omit e.g. most operations on primitive data types. The LOOP project (Jacobs et. al. [17][13]) uses a coalgebraic approach (and a formalization by a shallow embedding in PVS) where the state includes besides the heap also a stack for method calls and a memory for static data. So compared to others, this semantics seems to be more simple (concerning the state and the used formalism, algebraic specifications) and may be easier to understand for non-experts.

## 4 A Calculus for Java Card

### 4.1 Sequents and Dynamic Logic

The calculus is a sequent calculus for dynamic logic [10]. Dynamic logic extends predicate logic with two modal operators, box  $[ \cdot ]$  and diamond  $\langle \cdot \rangle$ . The intuitive meaning of  $\langle H; \alpha \rangle \varphi$  is: with initial heap  $H$  (a variable) the Java statement  $\alpha$  terminates, and afterwards  $\varphi$  holds.  $\varphi$  is again a formula of dynamic logic, i.e. it may contain boxes or diamonds. The meaning of  $[H; \alpha] \varphi$  is: if  $\alpha$  terminates then afterwards  $\varphi$  holds. The formal semantics is

$$\mathcal{A}, tds, v \models \langle H; \alpha \rangle \varphi :\Leftrightarrow \exists v', h'. v \times v(H) \llbracket \alpha \rrbracket_{tds} v' \times h' \text{ and } \mathcal{A}, (v')_{H}^{h'} \models \varphi$$

$\langle H; \alpha \rangle \varphi$  holds in an model  $\mathcal{A}$  with Java type declarations  $tds$  and variable mapping  $v$  iff there exists a new variable mapping  $v'$  and a heap  $h'$  such that  $\alpha$  with initial mapping  $v$  and initial heap  $v(H)$  computes  $v'$  and  $h'$  (this means that  $\alpha$  terminates) and afterwards  $\varphi$  holds under the new variable mapping where the new heap  $h'$  is bound to the variable  $H$ . A sequent  $\varphi_1, \dots, \varphi_m \vdash \psi_1, \dots, \psi_n$  consists of two lists of formulas (often abbreviated by  $\Gamma$  and  $\Delta$ ) divided by  $\vdash$  and is equivalent to the formula  $\varphi_1 \wedge \dots \wedge \varphi_m \rightarrow \psi_1 \vee \dots \vee \psi_n$ .  $\varphi_1, \dots, \varphi_m$  can be thought of as preconditions, while one of  $\psi_1, \dots, \psi_n$  must be proved. A Hoare triple  $\{\varphi\}\alpha\{\psi\}$  can be expressed as  $\varphi \vdash [H; \alpha]\psi$  or  $\varphi \vdash \langle H; \alpha \rangle \psi$  if termination is included. However, more things can be expressed (e.g. program equivalence).

## 4.2 Some Example Rules

The calculus essentially has one rule for every Java expression and statement, plus some generic rules. It works by symbolic execution of the Java program from its beginning to its end (i.e. computation of strongest postcondition). This means it follows the natural execution of the program, and is much more intuitive than inventing intermediate values (as in a Hoare calculus) or computing weakest preconditions (by evaluating a program from the end to the start). Nested expressions and blocks are flattened to a sequence of simple expressions and statements that can be executed directly. Obviously, this flattening must obey the evaluation order of Java. The additional Java statements mentioned in the last section are used to mark the end of a block. A box or diamond never contains isolated expressions but only statements (e.g. an expression statement). The result of an expression is ‘stored’ with an assignment to a local variable. Most of the rules are applicable if the program occurs on the left or on the right hand side of the turnstile  $\vdash$ , and they are applicable for boxes and diamonds.

An instance field assignment has three premises:

$$\begin{array}{l}
 1. \quad \Gamma, H[\text{mode}] \neq \text{normal} \vdash \varphi, \Delta \\
 2. \quad \Gamma, H[\text{mode}] = \text{normal}, e = \text{null} \\
 \quad \vdash \langle H; \text{throw new NullPointerException();} \rangle \varphi, \Delta \\
 3. \quad \Gamma, H[\text{mode}] = \text{normal}, e \neq \text{null}, H_0 = H[e \times f, e_0] \\
 \quad \vdash \langle H_0; x = e_0; \rangle \varphi[H/H_0], \Delta \\
 \hline
 \Gamma \vdash \langle H; x = (e.f = e_0); \rangle \varphi, \Delta
 \end{array}$$

$f$  is the field name,  $e$  the invoking expression. The rule is only applicable if  $e$  and  $e_0$  are *basic* expressions, either local variables or literals. If the mode is not normal the expression is skipped (first premise). If the invoking expression is `null` a `NullPointerException` is raised (second premise). Otherwise the heap  $H$  is modified by setting the field  $e \times f$  to the value of  $e_0$  ( $H[e \times f, e_0]$ ). Then the computation continues with the new heap (bound to  $H_0$ , a new variable). Since  $e$  and  $e_0$  are local variables or literals they require no further evaluation, but can be taken directly as values. If they are other Java expressions they have to be flattened first.

The flattening rule works as follows:

1. For an expression  $x = e$  select the immediate subexpressions  $e_1, \dots, e_n$  of  $e$ .
2. Find the first  $e_i$  that is not a local variable or a literal, and that does not cause a variable conflict (see case 4).
3. Replace  $e_i$  in  $e$  by a new variable  $y$  yielding  $e'$  and add the assignment  $y = e_i$  before  $x = e'$ .
4. A variable conflict occurs if  $e_i$  contains an assignment to a variable that occurs in  $e_1, \dots, e_{i-1}$ , e.g. in  $\mathbf{x} * (\mathbf{x} = 3)$ . In this case a renaming is necessary.

After a finite number of applications the algorithm will return a list of assignments where every subexpression is either a local variable or a literal. Then a rule

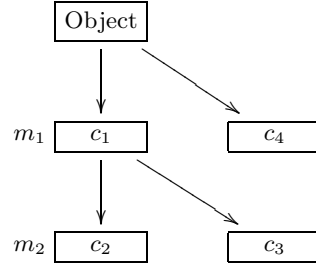
for this expression is applicable. The test of an `if`, the expression of a `switch`, `return`, or `throw` can also be flattened in this manner (but not the test of a `while`, `do` or `for`). A block is also flattened:

$$\frac{\begin{array}{l} 1. \Gamma, H[\text{mode}] \neq \text{normal} \vdash \varphi, \Delta \\ 2. \Gamma, H[\text{mode}] = \text{normal} \vdash \langle H; \alpha'_1[\underline{x}/\underline{y}] \rangle \dots \langle H; \alpha'_n[\underline{x}/\underline{y}] \rangle \varphi, \Delta \end{array}}{\Gamma \vdash \langle H; \{\alpha_1 \dots \alpha_n\} \rangle \varphi, \Delta}$$

$\alpha_i$  are the top level statements of the block,  $\underline{x}$  the local variables declared in the block, and  $\underline{y}$  new variables.  $\alpha'_i$  is  $\alpha_i$  except for a local variable declaration. *ty*  $x = e$  becomes an assignment  $x = e$ . Note that this is not legal Java if  $e$  is an array initializer, but we treat it as a normal expression. The replacement with new variables ensures that the variables really behave as local variables. A similar flattening happens for the `try` statement:

$$\frac{\begin{array}{l} 1. \Gamma, H[\text{mode}] \neq \text{normal} \vdash \varphi, \Delta \\ 2. \Gamma, H[\text{mode}] = \text{normal} \vdash \langle H; \alpha \text{ catches}(\text{catches}) \text{ finally}(\beta) \rangle \varphi, \Delta \end{array}}{\Gamma \vdash \langle H; \text{try } \alpha \text{ catches finally } \beta \rangle \varphi, \Delta}$$

The list of catch clauses *catches* is transformed into an additional Java statement `catches(catches)`, and the finally clause is transformed into an additional Java statement `finally( $\beta$ )`. In this manner expressions and statements can be flattened. Loops can be unwound and treated by induction.



$$\frac{\begin{array}{l} 1. \Gamma, \text{mode}(st) \neq \text{normal} \vdash \varphi, \Delta \\ 2. \Gamma, e = \text{null}, \text{mode}(st) = \text{normal} \vdash \langle H; \text{throw new NPE}() \rangle \varphi, \Delta \\ 3. \Gamma, e \neq \text{null}, \text{mode}(st) = \text{normal} \vdash \text{classOf}(e, st) \in \{c_1, c_2, c_3\}, \Delta \\ 4. \Gamma, e \neq \text{null}, \text{mode}(st) = \text{normal}, \text{classOf}(e, st) \in \{c_1, c_3\}, \\ \text{this}' = e, \underline{z} = es \vdash \langle H; \alpha'_1 \rangle \langle H; \text{targetexpr}(x) \rangle \varphi, \Delta \\ 5. \Gamma, e \neq \text{null}, \text{mode}(st) = \text{normal}, \text{classOf}(e, st) \in \{c_2\}, \\ \text{this}' = e, \underline{z} = es \vdash \langle H; \alpha'_2 \rangle \langle H; \text{targetexpr}(x) \rangle \varphi, \Delta \end{array}}{\Gamma \vdash \langle H; x = e.m(es) \rangle \varphi, \Delta}$$

**Fig. 3.** Example class hierarchy and rule for instance method invocation.

As a last example we show the rule for an instance method invocation. This rule has a variable number of premises depending on the number of possible

methods that can be invoked. Figure 3 shows a class hierarchy where class  $c_1$  contains a method declaration  $m$  with a body  $m_1$  that is overwritten in class  $c_2$  with another body  $m_2$ . If  $e.m(e_1, \dots, e_n)$  is the method invocation the correct method body to invoke depends on the runtime type of  $e$ .  $e$  must be a local variable or literal and should be a reference  $\neq \text{null}$ . If  $e$  is a reference to an object with type  $c_1$  or  $c_3$  then method  $m_1$  is invoked; if the type of  $e$  is  $c_2$  then method  $m_2$  is invoked; the third premise ensures that the type of  $e$  is either  $c_1$ ,  $c_2$ , or  $c_3$ . Note that there is no possibility to omit this premise. A type correct Java program guarantees this if the heap conforms to the program, but the calculus does not enforce the latter. The heap can contain arbitrary entries that have nothing to do with the classes and declarations of the program. Therefore it is necessary to *prove* that the runtime type of  $e$  is ‘legal’. Together with a ‘jump’ case and a **null** invoker the rule has five premises in this instance (Fig. 3). In premises 4. and 5. the parameters  $e_1, \dots, e_n$  (that must be local variables or literals) are bound to new variables  $\underline{z}$ , a new variable *this'* is introduced for **this** and bound to  $e$ , in the method body the formal parameters are replaced with the new variables yielding  $\alpha'_i$ , and a new statement **targetexpr**( $x$ ) is added that will catch a return statement and bind  $x$  to the returned value.

### 4.3 Interactive Proofs

One of the most important features of the calculus is that it is well suited for interactive proofs, much better – we feel – than a Hoare or wp calculus. Because a formula containing a Java statement is only one formula among others, the user can mix predicate logic rules (case distinctions, quantifier instantiations, cut, etc., or advanced rules like rewriting or simplification) freely with Java rules. This helps to reduce the size of non-Java formulas during the proof. Furthermore, a method call (or any other statement) can be either evaluated, or one or several lemmas for the method call can be applied at different points in the proof. The reason is that programs can appear on both sides of the turnstyle  $\vdash$  so that it is possible to argue that if program  $\alpha$  computes  $x$  then program  $\beta$  computes  $y$ . The following proof rule is valid:

$$\frac{\langle H; \alpha \rangle \underline{x} = \underline{y}, \varphi[\underline{x}/\underline{y}], \Gamma \vdash \psi[\underline{x}/\underline{y}], \Delta}{\langle H; \alpha \rangle \varphi, \Gamma \vdash \langle H; \alpha \rangle \psi, \Delta}$$

We know  $\langle H; \alpha \rangle \varphi$ , i.e.  $\alpha$  terminates and afterwards  $\varphi$  holds (this can be a lemma about  $\alpha$ ).  $\alpha$  can only modify a number of variables  $\underline{x}$  (the assigned variables of  $\alpha$  and the heap  $H$ ) and these variables describe the state exactly. Therefore we can introduce new variables  $\underline{y}$  that hold the result and then we know that  $\varphi[\underline{x}, \underline{y}]$  holds ( $\varphi$  with  $\underline{x}$  replaced with  $\underline{y}$ ). Now we want to prove  $\langle H; \alpha \rangle \psi$  and we know that  $\alpha$  computes  $\underline{y}$ . Therefore we can discard  $\alpha$  on the right hand side of the turnstyle  $\vdash$  and it remains to prove that  $\psi[\underline{x}, \underline{y}]$  holds. This is the typical situation if a lemma is used. But the program  $\alpha$  does not disappear from the sequent. It remains in the antecedent and later the user can apply another lemma that introduces another property. Nothing similar is possible in a Hoare

calculus. Another advantage is that induction can be used for loops or recursive methods. This means the user has more freedom to structure proofs.

#### 4.4 Correctness of the Calculus

The proof rules are specified in KIV and their correctness with respect to the semantics has been proved. Since the calculus introduces several new operations (the notion of *free* and *assigned* variables, replacement of variables, flattening of expressions, formulas, sequents etc.) the complete specification is considerably larger than just the Java semantics. Especially operations that are defined for all expressions or statements have lots of axioms, and proofs for these operations require the most time. Furthermore, the semantics' relations for expressions, expression lists, statements, and statement lists are mutually recursive and must be formulated and proved for all four relations at once. The most time consuming proofs are concerned with the correct definition of free and assigned variables (the semantics of a Java statement depends only on the values of the free variables, and only assigned variables can be changed) because the definitions should not include superfluous variables. Other important proofs concern the correctness of variable replacement and flattening. In comparison the rule for instance method invocation is quite simple to prove because in the semantics and in the proof rule the method lookup just travels up in the class hierarchy. All 57 rules have been proved correct. The specification and verification effort required several months of work. Currently the calculus is not (relatively) complete for two reasons: First, there is no possibility to argue about the number of recursive method calls. Experience shows that usually induction on the used data structure is sufficient (a counter that becomes smaller, or the length of a list that decreases). However, it is not possible to prove that a method that just calls itself (`void m() {m(); }`) does not terminate. Second, the semantics does not depend on type correct programs (or programs that pass a Java compiler – the compiler checks more than just type correctness). But for verification we are interested only in type correct programs, and for type correct programs more efficient rules can be designed. (They are correct but incomplete for type incorrect programs.)

As can be imagined several errors were found during verification. Most of them are errors only for type incorrect programs. For example, the definition of free variables must handle the case that a local variable declaration occurs outside a block or that a local variable occurs outside the scope of a local variable declaration. Other errors were more serious because they concerned type correct programs:

1. The third premise in the instance method invocation was missing (that checks that the runtime type of the invoker is correct, see Fig. 3). As mentioned above this is important even for type correct programs because the heap may not conform to the program.
2. The flattening rule contained no check for variable conflicts. `y = x * (x = 3);` was flattened to `z = (x = 3); y = x * z;` which is wrong. (A renaming is needed: `x0 = x; z = (x = 3); y = x0 * z;`).

3. A similar problem occurred for the postfix increment: `x = y++` was transformed into `x = y; y = y + 1;` which is wrong for `x = x++;` (the right hand side is fully evaluated before the assignment occurs).

These errors were found because the verification of the proof rules failed. However, some errors were found in the semantics as well.

1. *first active use* was not handled correctly in the semantics rules, i.e. whether static initialization occurs before or after the arguments are evaluated. (compound assignment to static field and `new` class: before evaluation of arguments, simple assignment to static field and static method invocation: after evaluation of arguments.)
2. The semantics rules for compound assignment failed to cast the result back to `byte` or `short` if the right hand side was `byte` or `short`. (In `byte b = 127; b += 1;` the result is `(byte)-128`.)

These errors were found because the verification of the proof rules also failed, and analysis revealed the errors to be in the semantics. However, both semantics and calculus could be wrong. It is possible to validate the semantics by ‘running’ test programs in KIV (automatically applying the proof rules) and comparing the output with a run of a Java compiler and JVM (currently 150 examples), and this certainly increases confidence in the semantics, but who would think about writing programs like `x = x++;`?

## 5 An Example Program

The aim of the example is to show the ‘look and feel’ of the calculus for a small example that involves a `for` loop, exceptions and abrupt termination of the loop. It is typical for Java Card programs to use this programming style. We consider the Java Card application for storing tickets mentioned in section 2. Since the available memory for a Java Card program (an *applet*) is severely limited the maximal number of tickets that can be stored must be fixed in advance. The missing garbage collector means that storage cells cannot be reclaimed. Therefore it is usual programming practice to allocate all objects when the applet is loaded onto the smart card, and to reuse these objects. Our applet has a capacity of 20 tickets that are stored in an array. A field `free` indicates if the entry is free or not. If a ticket is loaded the value is set to false, if it is deleted after usage the entry is set back to true.

```
class Ticket {boolean free = true; (rest of class)}

public class Cardlet extends Applet {
    final static byte MAX = 20;
    Ticket[] tickets = new Ticket[MAX];
    Cardlet() {
        for(byte c=0;c<MAX; c++) tickets[c] = new Ticket();}
    (rest of class)}
}
```

A Java Card applet works as follows: An `install` method is called once when the applet is loaded onto the card. This method typically calls the constructor. All data (fields, objects, and arrays) is stored permanently in the EEPROM of the smart card. When the smart card is inserted in a reader and the applet is selected, a method `select` is called once. Afterwards all communication takes place through a `process` method. The JVM receives the input, a sequence of bytes, stores them in a byte array and calls the `process` method with this array. `process` normally computes an answer that it stores in the byte array, and that will be sent back to the terminal by the JVM. Another possibility is to throw an exception that is converted by the JVM into an error message.

The following method can be used somewhere in the applet to find a free position:

```
byte findFree () {
    for(byte c=0; c < MAX; c++)
        if (tickets[c].free) return c;
    ISOException.throwIt(SW_FILE_FULL);
}
```

If no free position is available an exception is thrown (without creating a new exception object!) from the predefined method `ISOException.throwIt`. This exception ends the `process` method and results in an error message to be sent to the terminal (`SW_FILE_FULL`). If the `findFree` method is used several times in the code it is good proving practice to formulate some lemmas about its behaviour and re-use them wherever possible. For example,

$$\begin{aligned}
& \text{findFree-install} : \text{install}(H) \vdash \langle H; \text{by} = \text{cardlet.findFree}(); \rangle \text{install}(H) \\
& \text{findFree-throw} : \\
& \quad \# \text{tickets}(H) = \text{b2i}(\text{MAX}), \text{install}(H), H[\text{mode}] = \text{normal} \\
& \vdash \langle H; \text{by} = \text{cardlet.findFree}(); \rangle \text{ISOException}(\text{SW\_FILE\_FULL}, H) \\
& \text{findFree-ok} : \\
& \quad \# \text{tickets}(H) < \text{b2i}(\text{MAX}), H = H_0, \text{install}(H), H[\text{mode}] = \text{normal} \\
& \vdash \langle H; \text{by} = \text{cardlet.findFree}(); \rangle (H = H_0 \wedge \text{free\_ticket}(\text{b2i}(\text{by}), H))
\end{aligned}$$

The method assumes that the array entries are not `null`. This is the case after installation. Hence an invariant  $\text{install}(H)$  is needed for the applet. This invariant will contain other properties of the applet that should hold before and after every communication step (i.e. before and after every call of `process`), including logical properties related to the cryptographic protocols. Finding this invariant is not trivial, but essential for the correctness of the applet.  $\text{install}(H)$  is a user defined predicate for the heap. The second property,  $\text{findFree-throw}$ , states that the method will throw an exception `SW_FILE_FULL` if the number of tickets stored in the heap is already `MAX` ( $\# \text{tickets}(H) = \text{b2i}(\text{MAX})$ , `b2i` converts a byte into an integer). Here we can assume that the invariant holds, and we must assume that no abrupt transfer of control happens initially ( $H[\text{mode}] = \text{normal}$ ) because then the method call will be skipped. Finally,  $\text{findFree-ok}$  states that the method will return a free position if there is one.

We show the proof for *findFree-ok*. Method call and initialization of the `for` loop results in

$$\begin{aligned} & H = H_0, \text{ this} = \text{cardlet}, c = 0, H[\text{mode}] = \text{normal}, \\ & \# \text{tickets}(H) < \text{b2i}(\text{MAX}), \text{install}(H) \\ \vdash & \langle H; \text{for}(c < \text{MAX}; c++) \text{ if } (\text{this.tickets}[c].\text{free}) \text{ return } c; \text{ else } \{ \} \rangle \\ & \langle H; \text{ISOException.throwIt}(\text{SW\_FILE\_FULL}); \rangle \\ & \langle H; \text{targetexpr}(\text{by}) \rangle (\text{free.ticket}(\text{b2i}(\text{by}), H) \wedge H = H_0) \end{aligned}$$

The `for` loop contains no initialization so it can be unwound; *cardlet* is a reference to an object of type `Cardlet` that becomes the value of `this` inside the method. Now we use induction on  $|\text{MAX} - c|$  and generalize the goal by replacing  $c = 0$  with  $0 \leq c \wedge c \leq \text{MAX}$  (this is done automatically), and add the formula  $\# \text{tickets}(H[\text{cardlet} - \text{tickets}].\text{refval}, \text{b2i}(c) - 1, H) = \text{b2i}(c)$  stating that the number of tickets from 0 to  $c - 1$  in the array is  $c$  (this means that all tickets below  $c$  are not free).  $H[\text{cardlet} - \text{tickets}].\text{refval}$  returns the reference that is stored in the `tickets` field of the `cardlet` object. This property is needed to prove that the loop counter  $c$  can never reach `MAX`. Then we unwind the `for` loop once and obtain

$$\begin{aligned} & \text{Ind-Hyp}, \dots \langle \text{other preconditions} \rangle \dots \\ \vdash & \langle H; \text{if } (\text{b2i}(c) < \text{b2i}(\text{MAX})) \\ & \quad \text{if } (\text{this.tickets}[c].\text{free}) \text{ return } c; \text{ else } \{ \} c++; \rangle \\ & \langle H; \text{for}(c < \text{MAX}; c++) \text{ if } (\text{this.tickets}[c].\text{free}) \text{ return } c; \text{ else } \{ \} \rangle \\ & \langle H; \text{ISOException.throwIt}(\text{SW\_FILE\_FULL}); \rangle \\ & \langle H; \text{targetexpr}(\text{by}) \rangle (\text{free.ticket}(\text{b2i}(\text{by}), H) \wedge H = H_0) \end{aligned}$$

If the first `if` test is true and the second one is false we obtain after the postfix increment `c++`

$$\begin{aligned} & \text{Ind-Hyp}, c_0 = \text{i2b}(\text{b2i}(c) + 1), \neg H[r_0 - \text{free}].\text{boolval} \\ & \dots \langle \text{other preconditions} \rangle \dots \\ \vdash & \langle H; \text{for}(c_0 < \text{MAX}; c_0++) \text{ if } (\text{this.tickets}[c_0].\text{free}) \text{ return } c_0; \text{ else} \{ \} \rangle \\ & \langle H; \text{ISOException.throwIt}(\text{SW\_FILE\_FULL}); \rangle \\ & \langle H; \text{targetexpr}(\text{by}) \rangle (\text{free.ticket}(\text{b2i}(\text{by}), H) \wedge H = H_0) \end{aligned}$$

The formula on the right hand side of  $\vdash$  (with the `for` loop) is identical to the formula where the induction started, except that  $c$  is replaced by  $c_0$ . This means we can apply the induction hypothesis (This requires proving that no overflow occurs for the `byte` value  $c$ .), and obtain a program formula on the left hand side of  $\vdash$ . The result is an axiom:

$$\langle H; \text{for}(c_0 < \text{MAX}; c_0++) \dots \vdash \langle H; \text{for}(c_0 < \text{MAX}; c_0++) \dots$$

Aside from the rather longish sequents the proof proceeds as a proof done on paper. The same principle (induction and unwinding of the loop) can be used for `while` or `do` loops. In other proofs only the lemmas for the `findFree` method are used. This allows a nice structuring of the proofs.

## 6 Comparison to other Proof Systems for Java

Java Card verification in KIV seems to be unique in that an existing prover was extended to incorporate a Java Card calculus. In the KeY project [16][20] a new prover is developed from scratch; Oheimb [27][25] models a formal (operational) semantics and a Hoare calculus (for a subset of Java) in Isabelle; in the LOOP project [16][17][14] Java together with JML [6] annotations are translated into a formal (denotational) semantics enriched by proof rules for a Hoare- and weakest precondition calculus in PVS; the Krakatoa tool [21] translates into Coq; Jack [7] into a prover for the B method. The KIV approach has two advantages compared to the others: First, an already good prover can be used for the non-Java parts; second, the prover can be tailored to the goals that arise in Java verification. This includes simple things like pretty printing to make the goals more readable, but also special heuristics and simplification strategies (e.g. a special treatment of the heap). The drawback is, of course, that access to the internals of the prover is necessary.

KeY also uses a dynamic logic, but the calculus is not proved correct w.r.t. a formal semantics. The three main differences are: exceptions are modeled as non-termination, blocks are not flattened (though expressions are), and there is no explicit heap. The last two lead to more complex formulas: programs contain a ‘prefix’ representing nested blocks (including try catch blocks) and ‘updates’ to objects to cope with aliasing. On the other hand, omitting the heap may help to prove that a method does *not* modify some objects. Oheimb uses a pure Hoare calculus that is tailored for backward reasoning (i.e. computes weakest preconditions), and has proved its correctness and completeness in Isabelle. One specialty of the calculus is that the state must always conform to the (type correct) program. This may require unnecessary proof work when the consequence rule is used (the Java rules preserve conformity), and it is not clear how theorems that are proved for a library or API class can be reused (because the new state contains new objects that do not conform to the original classes). We do not require this conformity. The LOOP project also uses a Hoare calculus and weakest preconditions (formally proved correct). As mentioned earlier, we feel that a dynamic logic allows more flexibility. It is interesting to note that Oheimb uses quadruples as pre- and postconditions, while LOOP uses 8-tuples. So it seems that the calculus presented here has the most simple structure (a heap and two modal operators for the programs).

Another difference to most other approaches is that in KIV the user plays an important role and is expected (and encouraged) to interact with the system to keep the proofs manageable. In KeY, LOOP, Krakatoa, and Jack the prover is viewed as a back end system that is best used fully automatically. Ideally, the prover is fed with some Java goals, generates proof obligations that contain no longer Java statements, and proves these goals automatically. However, this works only up to a given size of the formulas. Better support for interactive proofs can help to reduce the size while proving, but this requires (currently) a rather experienced user.

## 7 Conclusion

We presented (in excerpts) a formal semantics and calculus for Java Card (essentially sequential Java) that supports all language constructs. The semantics is a natural (big-step) semantics, which is adequate for deterministic sequential programs. The semantics is defined relatively to an algebraic specification of a heap and Java's primitive types. The complete specification is big, but Java is a complex language. The calculus is a sequent calculus for dynamic logic that is more expressive than a Hoare calculus and – we feel – better suited for interactive proofs. The calculus has been proved correct formally with KIV, and is also implemented in KIV. Currently, KIV seems to be the only existing prover that was extended for a Java Card calculus.

The main application area is in the context of smart cards where interesting and security critical (because money and privacy is involved) e-commerce applications like electronic ticketing make a formal verification very desirable. It must be proved that the program correctly implements a cryptographic protocol that guarantees the security of the application. Without formal methods it is very difficult to assess the correctness and security of an interesting smart card application.<sup>2</sup> The programming style in Java Card requires that certain features (byte and short arithmetic, arrays, `for` loops, exceptions etc.) are modeled precisely and handled efficiently in the prover. Experience shows that the main difficulties are reasoning about bytes, shorts, byte arrays, cryptographic methods, and invariants that hold between communications.

One can ask why a prover should support all the complex features of sequential Java. There are three reasons. First, Java Card programs typically use these features; second, to show people who are critical towards formal methods that the field has reached at least that maturity; third, because formal methods do not scale up by themselves. It is not clear if a double sized program requires double effort or more. And even if the increase is linear every prover will eventually fail (see e.g. [17]: “PVS can run for hours without completing the proof, or it can crash because the proof state becomes too big.”). So work has still to be done to reduce the complexity of formal proofs.

## References

1. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*. Springer LNCS 1523, 1999.
2. I. Attali and T. Jensen, editors. *Java on Smart Cards: Programming and Security*. Springer LNCS 2041, 2001.
3. M. Balsler, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, Springer LNCS 1783, 2000.

---

<sup>2</sup> The author had to inspect (without formal methods) 35.000 lines of Java Card code written by students in a practical course, so this statement comes from own experience.

4. B. Beckert. A dynamic logic for the formal verification of java card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards*. Springer LNCS 2041, 2000.
5. E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In [1].
6. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Proceedings FMICS '03*. Volume 80 of Electronic Notes in Theoretical Computer Science, Elsevier, 2003.
7. N. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In *Formal Methods Europe (FME)*, Springer LNCS, 2003.
8. S. Drossopoulou and S. Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In [1].
9. S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors. *Formal Techniques for Java Programs, Proceedings ECOOP2000 Workshop*. Technical Report 269, 5/2000, Fernuniversität Hagen.
10. D. Harel. *First Order Dynamic Logic*. LNCS 68. Springer, Berlin, 1979.
11. P. Hartel and L. Moreau. Formalizing the safety of Java, the Java virtual machine, and Java card. *ACM Computing Surveys (CSUR)*, 33(4), December 2001.
12. KIV home page. <http://www.informatik.uni-augsburg.de/swt/fmg/>.
13. M. Huisman. *Reasoning about JAVA programs in higher order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, IPA dissertation series, 2001-03, 2001.
14. M. Huisman and B. Jacobs. Java Program Verification via a Hoare Logic with Abrupt Termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE'00)*. Springer LNCS 1783, 2000.
15. B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors. *Formal Techniques for Java Programs*. Technical Report 251, Fernuniversität Hagen, 1999.
16. B. Jacobs and E. Poll. A logic for the java modeling language JML. In *Proceedings FASE 2001*, Genova, Italy, 2001. Springer LNCS 2029.
17. B. Jacobs and E. Poll. Java program verification at nijmegen: Developments and perspective. Technical Report NIII-R0318, University of Nijmegen, 2003.
18. *Java Card 2.2 Specification*, 2002. <http://java.sun.com/products/javacard/>.
19. Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java (tm) Language Specification, Second Edition*. Addison-Wesley, 2000.
20. KeY project homepage. <http://i12www.ira.uka.de/~key>.
21. Krakatoa home page. <http://krakatoa.lri.fr/>.
22. T. Nipkow and D. von Oheimb. Java light is Type-Safe – Definitely. In *25th ACM Symposium on Principles of Programming Languages*. ACM, 1998.
23. W. Reif, G. Schellhorn, K. Stenzel, and M. Balsler. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*. Kluwer Academic Publishers, 1998.
24. R.F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001.
25. D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.
26. D. von Oheimb and T. Nipkow. Machine-checking the Java Specification: Proving Type-Safety. In [1].
27. D. von Oheimb. Axiomatic semantics for Java<sup>light</sup> in Isabelle/HOL. In [9].