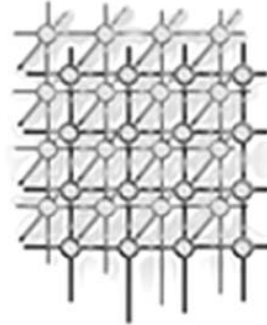*

Analysing the Java Package/Access Concepts in Isabelle/HOL

Norbert Schirmer

Technische Universität München
Department of Informatics
D-85748 Garching, Germany
Email: schirmer@in.tum.de

SUMMARY

Java access modifiers and packages provide a mechanism to restrict access to members and types, as an additional means of information hiding beyond the purely object-oriented concept of classes. In this paper we clarify the semantics of access modifiers and packages by adding them to our formal model of Java in the theorem prover Isabelle/HOL. We analyse which properties we can rely on at runtime, provided that the program has passed the static accessibility tests.

KEY WORDS: Java Packages, Java Access Modifiers, Theorem Proving, Type Safety

The work presented in this article is part of a comprehensive research effort aiming at formalising and verifying key aspects of the Java programming language. In particular we have a type system, an operational semantics (with a proof of type soundness), and an axiomatic semantics (with a proof of its equivalence to the operational semantics) for a large subset of Java [8]. All these formalisations and proofs have been carried out in the Isabelle/HOL system [7].

Access modifiers determine access restrictions and visibility of class or interface types, and their members. Since safety and security properties of Java are based on the bare language itself, access modifiers are the main means to protect data. During the effort to formally model the package/access concept, some intrinsic problems became apparent. The Java Language Specification (JLS) [3] is imprecise and ambiguous concerning the package/access concepts and Java implementations do not exactly follow the JLS. Although some of the problems have already been known for years (cf. BugParade [10], Bug Ids 1240831, 4094611), we have discovered and reported to Sun, two further inconsistencies (Bug Id's 4485402, 4493343).

Information hiding (with packages) and reuse of implementations (with inheritance and overriding) are conflicting goals but they are both implemented using the same concept of Java, namely classes. In this article we clarify the semantics and discuss the runtime properties of access modifiers. Since it is unclear from the JLS what the exact meaning of various relevant notions concerning the package/access concepts are, or even worse, what exactly the relevant notions are, we will introduce and clarify the following definitions in this article:

accessible-in: When is a class or interface accessible in a package?

inheritable-in: When can a member be inherited in a package?

member-of : Which are the members of a class, including inherited members?

member-in: Which are the members of a class and its superclasses? This extends *member-of* with the members of superclasses that are not inherited.

permits-acc-from: Which classes are permitted to access a member?

accessible-from: Which member accesses are statically valid?

dyn-accessible-from: What are the properties of runtime member access?

overrides$_S$: Compile-time (static) variant of overriding.

overrides: Runtime (dynamic) variant of overriding.

We will make the subtle differences between the two pairs of notions, *member-of* vs. *member-in*, and *overrides$_S$* vs. *overrides* precise in this article. The package/access model presented in this article is based on the JLS. Inner classes are not yet part of our formalisation. In case of ambiguities or omissions in the JLS we refer to the Java release of Sun (SDK 1.3.1). The compiler serves as reference for the static semantics and the JVM serves as reference for the dynamic semantics. As far as we know, the present formalisation offers the most comprehensive and detailed model of the Java package/access concepts. Most studies on the formal semantics of Java do not treat access modifiers at all, because of a different focus of interests ([1], [11], [4], [8]). Other approaches which do model access modifiers take an idealistic view on them and avoid a lot of the complexities which arose in our formalisation ([9], [5]). The complete formalisation and the Isabelle theories are available online [12].

This article is structured as follows: Section 1 makes some basic remarks about Isabelle, as far as they concern the rest of this article. In Section 2, we introduce packages and access modifiers to our Java model. Section 3 discusses accessibility on the level of types. Section 4 describes accessibility on the level of members in detail. We also clarify the related notions of overriding and dynamic method lookup here. Section 5 links static accessibility tests to runtime behaviour. We prove that we can trust some runtime properties concerning accessibility if the program passes the static accessibility tests of the compiler. For example we will prove that a private member of a class will never be accessed from outside of the class. Finally we conclude the work in Section 6.

1.   Preliminary Notes on Isabelle

Isabelle is a generic logical framework which allows one to encode different object logics. In this article we are only concerned with Isabelle/HOL [7], an encoding of higher order logic augmented with facilities for defining data types, records, inductive sets as well as primitive and total general recursive functions.

The syntax of Isabelle is reminiscent of ML, so we will not go into detail here. There are the usual type constructors $T_1 \times T_2$ for product and $T_1 \Rightarrow T_2$ for function space. The long arrow $\Longrightarrow$ is Isabelle's meta-implication and appears in conjunction with rules or theorems of the form $[\![P_1; \ldots; P_n]\!] \Longrightarrow C$ to express that from the premises P_1 to P_n we can conclude C. Apart from that there is the implication $\longrightarrow$ of the HOL object logic, along with standard connectives and quantifiers.

To emulate partial functions the polymorphic option type is frequently used:

datatype α *option* = *None* | *Some* α

Here α is a type variable, *None* stands for the undefined value and *Some x* for a defined value *x*. A partial function from type T_1 to type T_2 can be modelled as $T_1 \Rightarrow (T_2 \ option)$.

There is also a selector for the constructor *Some*, the function *the*:: α *option* $\Rightarrow \alpha$. It is defined by the sole equation *the (Some x)* = *x* and is total in the sense that *the None* is a legal, but indefinite value.

We conclude this section with a remark about the usage of fonts and faces in this article. Java source code is written in `typewriter`, Isabelle keywords have a **bold** face, formal entities are written in *italics* except for special mixfix syntax introduced in this article, which is typeset in slanted sans serif.

2.   Basic Definitions

In Java, a program is a collection of interfaces and classes, arranged in packages. The aim of a package is to combine closely related classes (and interfaces) inside a single unit and to offer privileged access between those classes. We model packages by qualifying all type names for interfaces and classes with a package name.

$$\textbf{record } qtname = pid\text{::}pname$$
$$tid\text{::}tname$$

The function *pid* selects the package name out of such a record, *tid* selects the type name. An instance of this record can be written as $(\!|pid{=}MyPackage,\ tid{=}MyType|\!)$. Java organises package names hierarchically. But this internal structure only plays a role during the lookup process for a package and to manage namespaces. However, as this hierarchy is irrelevant for accessibility concerns, we do not model it here. For example packages `java` and `java.lang` are two different packages. Package `java` has no privileged access to the contents of package `java.lang`.

In our formalisation a Java program is a mapping from qualified type names to the structures describing the corresponding classes and interfaces. The access modifiers are described as an enumeration:

$$\textbf{datatype } acc\text{-}modi = Private \mid Package \mid Protected \mid Public$$

The nomenclature resembles the original keywords of Java, except for *Package*, which models the nameless default access modifier of Java, that is applied when no other modifier is given explicitly. We define an ordering on the access modifiers, from the most restrictive to the most liberal:

$$Private < Package < Protected < Public.$$

The order will be used later on to describe the constraints on access modifiers when we override a method (p. 9). An access modifier is attached to every member (field, method) and to every class or interface. Consequently there is an accessibility concept on the level of types and on the level of members.

3.   Accessibility of Types

We capture accessibility of types in a predicate $\Gamma \vdash T$ *accessible-in* P stating that in the context of a program Γ the type T is accessible in package P.

T	$\Gamma \vdash T$ *accessible-in* P
$PrimT$	$True$
$Iface\ I$	$pid\ I = P \lor is\text{-}public\ \Gamma\ I$
$Class\ C$	$pid\ C = P \lor is\text{-}public\ \Gamma\ C$
$Array\ elemT$	$\Gamma \vdash elemT$ *accessible-in* P

Primitive types like `int` or `bool` are accessible in all packages. If an interface or class has the modifier *Public* then it is accessible in any package. Otherwise it is only accessible inside the same package. An array type is accessible in a package if the element type is accessible in this package.

4.   Accessibility of Members

Accessibility of members is more involved than accessibility of types. First we have to clarify what the members of a class are. $\Gamma \vdash m$ *member-of* C states that m is member of class C in context of program Γ.

$$\frac{\Gamma \vdash mbr\ m\ \textit{declared-in}\ C \qquad declclass\ m = C}{\Gamma \vdash m\ \textit{member-of}\ C} \qquad (\textsc{Immediate})$$

$$\frac{\begin{array}{cc} \Gamma \vdash m \text{ member-of } S \qquad \Gamma \vdash C \prec_{C1} S \qquad \Gamma \vdash \text{Class } S \text{ accessible-in } pid\ C \\ \Gamma \vdash memberid\ m \text{ undeclared-in } C \qquad \Gamma \vdash m \text{ inheritable-in } pid\ C \end{array}}{\Gamma \vdash m \text{ member-of } C} \ (\text{Inherited})$$

Here m is a pair $qtname \times memberdecl$ consisting of the declaration class of the member and the member declaration itself. With $declclass$ and mbr we can select the parts. The member declaration itself does not contain the declaration class. It contains information like the identifier of the member, its access modifier, the type of the member or whether it is a static or an instance member. Every freshly declared member is immediately member of the class. $\Gamma \vdash mbr\ m$ **declared-in** C demands that the declaration $mbr\ m$ is present in the body of class C and $declclass\ m = C$ guarantees that the formal declaration class of m is indeed class C. A class can also inherit members from its direct superclass[†]: $\Gamma \vdash C \prec_{C1} S$. In the following $\prec_C$ is the transitive closure and $\preceq_C$ is the reflexive transitive closure of the direct subclass relation $\prec_{C1}$. The Inherited rule describes under which conditions a member is inherited. If m is an inheritable member of the direct superclass S and S is accessible in the current package and the class C does not declare a new member with the same $memberid$, then m is inherited by C. The $memberid$ of a field is its name, and the $memberid$ of a method is its complete signature (name plus parameter types). A $Private$ member is not inheritable, $Protected$ and $Public$ members are always inheritable and $Package$ members are only inheritable inside the package of the member's declaration class. With $accmodi$ we can select the access modifier of a member.

$accmodi\ m$	$\Gamma \vdash m$ inheritable-in P
$Private$	$False$
$Package$	$pid\ (declclass\ m) = P$
$Protected$	$True$
$Public$	$True$

The following example illustrates the impact of inheritance on the notion **member-of**:

```
public class A {
  private int n;
}
public class B extends A {
}
```

Since `n` is `private` in class `A` it is not inherited by `B`. So `n` is not **member-of** class `B`. If a member is not inherited by a class, it is not **member-of** that class, although it is **member-of**

[†]In the JLS a class also inherits members from the direct super-interfaces it implements. This is not needed in our model for the following reasons: A wellformedness condition ensures that all interface methods are implemented by the class hierarchy (abstract classes are not supported). So for method inheritance it is sufficient to focus on the class hierarchy. Interface fields are not supported in our current model.

---

the superclass. Also, if we declare a new member in a class, a member of the superclass with the same *memberid* is hidden or overridden. Consequently the member of the superclass is not *member-of* the current class, according to this definition. This is important to note, since at runtime the dynamic type of a reference may be a subclass of the static type. Because of this fact the member we want to access may not be *member-of* the dynamic class anymore. Of course the referenced object will still contain the field, like an object of class B in the example will contain the field n, since B extends A. There must be a superclass S providing this member, so we define:

$$\Gamma \vdash m \text{ member-in } C \equiv \exists\ S.\ \Gamma \vdash C \preceq_C S \wedge \Gamma \vdash m \text{ member-of } S$$

All members that are *member-of* a class are also *member-in* that class. But additionally all members of the superclasses that are not inherited are also *member-in* the class. Therefore in the previous example n is *member-in* class B.

The basic access restrictions associated with the modifiers are expressed in the predicate $\Gamma \vdash m \text{ in } C \text{ permits-acc-from } accC$. This is a crucial building block to define accessibility later on. A member m in class C permits access from an accessing class $accC$ according to the following table:

accmodi m	$\Gamma \vdash m$ in C permits-acc-from $accC$
Private	$declclass\ m = accC$
Package	$pid\ (declclass\ m) = pid\ accC$
Protected	$pid\ (declclass\ m) = pid\ accC\ \vee$
	$\Gamma \vdash accC \prec_C declclass\ m \wedge (\Gamma \vdash C \preceq_C accC \vee \textit{is-static } m)$
Public	True

A *Private* member only permits access from the declaration class itself. A *Public* member permits access from every class. A *Package* member permits access from all classes in the same package. The restrictions of *Protected* access are twofold. First the member permits access from any class in the same package. Secondly the member can also be accessed from outside the package: all classes involved have to be in the same branch of the class hierarchy. Note that this may concern three different classes: The declaration class of the member, the class C the member belongs to (possibly a subclass of the declaration class), and the class $accC$ that tries to access the member. With $\Gamma \vdash accC \prec_C declclass\ m$ we ensure that the accessing class $accC$ already "knows" of the existence of the member by being a subclass of the declaration class. We could also take $\preceq_C$ instead of $\prec_C$ here, but the case were the accessing class and the declaration class are the same is already captured by the first conjunct, since then both classes are in the same package. For non-static members (also called instance members or object members) the accessing class must also be a superclass of class C: $\Gamma \vdash C \preceq_C accC$. This is circumscribed as class $accC$ is "involved in the implementation" of class C in the JLS. The intuition behind this is that a class should be able to access the known content of all possible subclasses for its implementation. But it should not be possible for a class to change the content of a superclass. The behaviour of a subclass can always depend on the behaviour of one of its superclasses. That is why a superclass is involved in the implementation of a

subclass. For static members (class members) this additional constraint is not necessary, since the member is in fact the same for all objects of a subclass of the declaration class. Consider the following example:

```
package P;            package Q;           package R;
public class A {      import P.A;          import Q.B;
  protected int n;    public class B       public class C
}                             extends A {          extends B {
                      }                    }
```

The member `n` is inherited by both classes `B` and `C`. With `A.n`, `B.n` or `C.n` I will circumscribe the access to field `n` via a reference of class `A`, `B` or `C` respectively. `B` is permitted to access `C.n` (formally $\Gamma \vdash$ n *in* C *permits-acc-from* B), since `B` is a superclass of `C`. But class `B` is not permitted to access `A.n` (formally $\neg \Gamma \vdash$ n *in* A *permits-acc-from* B), since `B` is not a superclass of `A`. Or, in the words of the JLS, class `B` is involved in the implementation of class `C` but not of class `A`. Of course, class `B` is permitted to access its own member `B.n`, since `B` is both the accessing and the accessed class and therefore trivially lies in the same package and access of a *Protected* member is granted inside the same package (formally $\Gamma \vdash$ n *in* B *permits-acc-from* B). If we consider the field `n` to be declared `static` in class `A`, then `A.n`, `B.n` and `C.n` will all refer to the same field. Therefore to forbid class `B` to access `A.n` would be useless. Class `B` could always access the same location over `B.n`.

Note the differences between the *Protected* case of *inheritable-in* and of *permits-acc-from*. In the JLS inheritance is not defined with an extra notion like *inheritable-in*, but with accessibility. That way *Protected* instance members would never be inherited across package boundaries (Bug ID: 4485402). This becomes obvious if we again refer to the example. Class `B` is not permitted to access `A.n`. So `n` would not be inherited by class `B` if inheritance would be based on this restriction. Our formalisation models the behaviour of all common compilers here, were *Protected* members can always be inherited by subclasses. The JLS does not capture the intended behaviour of inheritance of *Protected* members.

Now we are ready to define static accessibility of a member: $\Gamma \vdash$ m *of* C *accessible-from* accC. In context of program Γ member m of class C is statically accessible from class $accC$. The statically valid member accesses are determined by the two following rules.

$$\frac{\begin{array}{c} \Gamma \vdash m \text{ \textit{member-of} } C \\ \Gamma \vdash m \text{ \textit{in} } C \text{ \textit{permits-acc-from} } accC \\ \Gamma \vdash Class \ C \text{ \textit{accessible-in} } pid \ accC \end{array}}{\Gamma \vdash m \text{ \textit{of} } C \text{ \textit{accessible-from} } accC} \quad (\text{IMMEDIATE})$$

$$\frac{\begin{array}{c} \Gamma \vdash (declC, newM) \text{ \textit{overrides}}_S \text{ old} \\ new = (declC, mdecl \ newM) \quad \Gamma \vdash new \text{ \textit{member-of} } C \\ \Gamma \vdash C \prec_C S \quad \Gamma \vdash old \text{ \textit{of} } S \text{ \textit{accessible-from} } accC \\ \Gamma \vdash Class \ C \text{ \textit{accessible-in} } pid \ accC \end{array}}{\Gamma \vdash new \text{ \textit{of} } C \text{ \textit{accessible-from} } accC} \quad (\text{OVERRIDING})$$

If a member of a class permits access and the class itself is accessible then the member is accessible. That is the IMMEDIATE rule. Note that the class has to be accessible, too. *Public* members of a non *Public* class are only visible inside the package. If a subclass is *Public* however, these members become accessible from outside the package as they are inherited (and not overridden). It may be puzzling, why we have not already required $\Gamma \vdash m$ *member-of* C in the definition of $\Gamma \vdash m$ *in* C *permits-acc-from* $accC$. The reason is that we will also use the core notion of *permits-acc-from* for dynamic accessibility (p. 13) where we no longer require the member m to be *member-of* the class C but are satisfied with *member-in*.

The OVERRIDING rule needs more motivation, because it is not apparent in the JLS (Bug ID 4493343). It states that a method becomes accessible from a class $accC$ if it overrides another method that is already accessible from class $accC$. With $new = (declC, mdecl\ newM)$ we ensure that the member new is a method $newM$; $mdecl$ constructs a member from a method. This rule is only necessary to cover the special case of *Protected* methods, the other ones can be treated by the IMMEDIATE rule — *Public* methods always permit access, *Private* methods cannot be overridden at all and from outside of the package we can neither override nor access *Package* methods. Consider the following example:

```
package P;                          package Q;
public class A {                    import P.A;
  protected void foo(){}            public class B extends A {
}                                      protected void foo(){}
                                       protected void bar(){}
package P;                          }
import Q.B;
public class C {
  ...
  B b = new B();
  b.bar();  // not accessible
  b.foo();  // accessible
}
```

Equipped with the IMMEDIATE rule, C could only access *Protected* members declared in package P or in subclasses of C. It could not access B.foo(). But the Sun compiler (SDK 1.3.1) also implements the OVERRIDING rule. It will allow C to access B.foo(), because C can access A.foo() (A and C are both in package P), but will reject access to method B.bar(). The authors of [6] already reported this irritating behaviour. They consider this a flaw in the language definition. The intention of the language should be that a method overriding another method should semantically permit "at least as much access" as the overridden one. But the obvious syntactic indicator that a *Protected* or *Public* member permits "at least as much access" as a *Protected* one is not sufficient to guarantee this. Due to the twofold nature of *Protected* access we can only guarantee "at least as much access" if we actually weaken the modifier to *Public* when we override it outside of the package. If we require this the OVERRIDING rule is dispensable and the semantics gets clearer. The JLS however, is satisfied with *Protected* or *Public*. So [6] suggests to add this restriction and to additionally introduce a new modifier

`private protected` permitting access to all super and subclasses, but not to other classes in the same package. Sun considered the OVERRIDING rule a bug in their compiler and omitted it in the new release (SDK 1.4.0)[‡]. Now `C` can access `A.foo()` but is not allowed to access `B.foo()`. This is also irritating: if `B` would not redefine `foo()` it would inherit `A.foo()`. In this case, class `C` would be allowed to access `B.foo()`. Hence accessibility of `B.foo()` depends on class `B` overriding `foo()` or not. This does not fit well into the object-oriented paradigm. Whether it is preferable to support the OVERRIDING rule or not, is not clear from a software-engineering perspective. As just explained, both solutions lead to some irritating behaviour. This illustrates the difficulty of integrating module-based encapsulation and inheritance using the concept of classes. The clearest semantics is obtained when we only allow *Public* methods to override *Protected* methods of a different package.

4.1.   Overriding

Since overriding plays a major role for accessibility we now investigate under which circumstances a new method overrides an old one:

$$
\frac{
\begin{array}{c}
\textit{msig new} = \textit{msig old} \qquad \neg\ \textit{is-static new} \\
\Gamma \vdash \textbf{\textit{Method}}\ \textit{old}\ \textbf{\textit{inheritable-in}}\ \textit{pid}\ (\textit{declclass new}) \\
\Gamma \vdash\ \textit{declclass new} \prec_{C1} S \qquad \Gamma \vdash \textbf{\textit{Method}}\ \textit{old}\ \textbf{\textit{member-of}}\ S \\
\Gamma \vdash \textbf{\textit{Method}}\ \textit{old}\ \textbf{\textit{declared-in}}\ \textit{declclass old} \\
\Gamma \vdash \textbf{\textit{Method}}\ \textit{new}\ \textbf{\textit{declared-in}}\ \textit{declclass new}
\end{array}
}{
\Gamma \vdash\ \textit{new}\ \textbf{\textit{overrides}}_S\ \textit{old}
}
\qquad (\text{DIRECT})
$$

$$
\frac{
\Gamma \vdash\ \textit{new}\ \textbf{\textit{overrides}}_S\ \textit{inter} \qquad \Gamma \vdash\ \textit{inter}\ \textbf{\textit{overrides}}_S\ \textit{old}
}{
\Gamma \vdash\ \textit{new}\ \textbf{\textit{overrides}}_S\ \textit{old}
}
\qquad (\text{INDIRECT})
$$

Let us first focus on the DIRECT rule. The new and the old method must have the same signature. Overriding (and dynamic binding) is only defined for instance methods and not for static methods ($\neg$ *is-static new*). The old method has to be inheritable in the declaration class of the new method. In contrast to this the JLS again refers to accessibility here. This is wrong for the same reasons as discussed for inheritance. In that case a *Protected* method could never be overridden in another package. The old method has to be member of the direct superclass of the new method's declaration class. Of course, all methods have to be declared properly. The INDIRECT rule is just a transitivity rule for overriding. The overridden method *old* must also be an instance method. This is not visible in this rule but is ensured by a more general wellformedness predicate not shown here. This wellformedness predicate combines a lot of structural constraints on Java programs that we usually have in mind. For example, all methods in a wellformed program have to be welltyped. Concerning overriding, wellformedness restricts the overridden method *old*: if $\Gamma \vdash\ \textit{new}\ \textbf{\textit{overrides}}_S\ \textit{old}$ holds then *old*

[‡]The compilers of IBM (JDK version 1.1.8 and 1.3.1) both implement the OVERRIDING rule.

must be an instance method ($\neg$ *is-static old*) and its access modifier must syntactically be at least as liberal (*accmodi old* $\leq$ *accmodi new*). The following example demonstrates the rules for overriding.

```
package P;                          package Q;
public class A {                    import P.A;
   void foo(){}                     public class B extends A {
}                                     public void foo(){}
                                    }
package P;
import Q.B;
public class C extends B {
  public void foo(){}
}
```

`B.foo()` does not override `A.foo()`, since `A.foo()` has *Package* access and therefore is not inheritable in package `Q`. Moreover `C.foo()` overrides `B.foo()`, since `B.foo()` is *Public* and the DIRECT rule is applicable. But does `C.foo()` override `A.foo()` of the same package? According to our rules it does not, since `A` is not the direct superclass of `C` and the transitivity rule is not applicable either, because `B.foo()` does not override `A.foo()`. `A.foo()` and `B.foo()` are treated as uncorrelated methods and so it seems obvious that `C.foo()` should not override both of them at the same time. The Java compiler of Sun also behaves in a way that is compatible with these rules. Sun's Java virtual machine, however, does not. In their JVM, `C.foo()` overrides both `A.foo()` and `B.foo()`[§]. The Sun JVM seems to implement the following rules for overriding:

$$\frac{\begin{array}{c} msig\ new = msig\ old \qquad \neg\ is\text{-}static\ new \\ \Gamma \vdash \textbf{\textit{Method}}\ old\ \textbf{\textit{inheritable-in}}\ pid\ (declclass\ new) \\ \Gamma \vdash declclass\ new \prec_C declclass\ old \\ \neg\ is\text{-}static\ old \qquad resTy\ new = resTy\ old \qquad accmodi\ new \neq Private \\ \Gamma \vdash \textbf{\textit{Method}}\ old\ \textbf{\textit{declared-in}}\ declclass\ old \\ \Gamma \vdash \textbf{\textit{Method}}\ new\ \textbf{\textit{declared-in}}\ declclass\ new \end{array}}{\Gamma \vdash new\ \textbf{\textit{overrides}}\ old} \text{(DIRECT)}$$

$$\frac{\Gamma \vdash new\ \textbf{\textit{overrides}}\ inter \qquad \Gamma \vdash inter\ \textbf{\textit{overrides}}\ old}{\Gamma \vdash new\ \textbf{\textit{overrides}}\ old} \text{(INDIRECT)}$$

I will refer to these rules as dynamic overriding and to the previous ones as static overriding (indicated by the subscript S in **overrides**$_S$). The DIRECT rule now allows to override not only methods of the direct superclass but also methods from any superclass if they are inheritable.

[§]The JVMs of IBM (JDK version 1.1.8 and 1.3.1) implement yet another alternative: `C.foo()` overrides `A.foo()` but not `B.foo()`.

The other novelties in line 4 of the DIRECT rule can be viewed as wellformedness conditions that ensure type safety at runtime. They are built into the notion of dynamic overriding in the JVM because they are neither tested by the bytecode verifier nor by a runtime check. The JVM only regards a method *new* as overriding another method *old* if it is safe in a certain sense to call method *new* instead of the overridden method *old*. In particular, this is only the case if the result types conform. A dynamic method lookup can then execute the overridden method safely. The compiler on the other hand tests these type safety constraints along with overriding: If $\Gamma \vdash new$ **overrides**$_S$ *old* then the compiler enforces that the result types of the method conform (*resTy new = resTy old*),that the new access modifier is at least as liberal as the old one (*accmodi old* $\leq$ *accmodi new*) and that the overridden method also is an instance method ($\neg$ *is-static old*). As mentioned before, this is also ensured in our Java model by a general wellformedness predicate. Note that dynamic overriding does not ensure that the access modifier is at least as liberal as the old one. It only has to be non *Private*.

4.2. Method Lookup

Dynamic overriding is the key component for dynamic method lookup. If we search for a method at runtime we cannot just start with the dynamic class of the object, walk up the class hierarchy, and take the first method that matches. We have to take into account whether the method really overrides the statically expected method. Let the dynamic class of an object be *dynC* and the static class be *statC*. Type safety will ensure that the proper subtype relationship $\Gamma \vdash dynC \preceq_C statC$ will hold. With *super dynC* we select the superclass of *dynC*. In the context of an acyclic finite class hierarchy we can then define dynamic method lookup for a method with signature *sig* the following way:

> *dynmethd* Γ *statC dynC sig* =
>   (*case methd* Γ *statC sig of*
>   *None* $\Rightarrow$ *None*
> | *Some statM* $\Rightarrow$ (*case methd* Γ *dynC sig of*
>   *None* $\Rightarrow$ *dynmethd* Γ *statC* (*super dynC*) *sig*
> | *Some dynM* $\Rightarrow$
>   (*if* $\Gamma \vdash dynM$ **overrides** *statM* $\vee$ *dynM = statM*
>   *then Some dynM*
>   *else* (*dynmethd* Γ *statC* (*super dynC*) *sig*))))

The statically expected method is *statM* and will be found by the function *methd* if the method call is well typed. Therefore, in a well-typed program, *methd* Γ *statC sig* will always be of the form *Some statM*. With *methd* we can look up all methods that are **member-of** the class given as second argument, by searching up the class hierarchy for the first matching method that is also **member-of** the given class. So in *dynmethd* we start in the dynamic class *dynC* and look for a method with the correct signature which is **member-of** the class *dynC*. If we do not find a method in the dynamic class itself we just continue the search in the superclass. Note that it is perfectly valid that we do not find a method which is **member-of** the dynamic class itself. If we for example statically expect a method with *Package* modifier but the dynamic class is

defined in another package, the method will not be inherited. In the other case, if we find a method *dynM* in the dynamic class, we can stop searching if this method either overrides the statically expected method *statM*, or if we have already reached *statM*. Otherwise we continue the search in the superclass. Let us reconsider the previous example (p. 10) with the following code fragment:

```
A v = new B(); v.foo();
```

Here `v.foo()` will call `A.foo()` and not `B.foo()`, since `B.foo()` does not override `A.foo()`. Our Java formalisation also deals with interfaces and arrays so we have to lift dynamic method lookup to these types as well:

$$
\begin{aligned}
&dynlookup \ \Gamma \ statT \ dynC \ sig \equiv \\
&\quad (case \ statT \ of \\
&\qquad IfaceT \ I \ \Rightarrow \ dynimethd \ \Gamma \ I \ dynC \ sig \\
&\quad | \ ClassT \ statC \ \Rightarrow \ dynmethd \ \Gamma \ statC \ dynC \ sig \\
&\quad | \ ArrayT \ ty \ \Rightarrow \ methd \ \Gamma \ Object \ sig)
\end{aligned}
$$

For classes we use the previously defined *dynmethd* function. The methods we can call on an array are those of *Object*. For interfaces the auxiliary function *dynimethd* handles the lookup:

$$
\begin{aligned}
&dynimethd \ \Gamma \ I \ dynC \ sig \equiv \\
&\quad if \ imethds \ \Gamma \ I \ sig \neq \{\} \\
&\quad then \ methd \ \Gamma \ dynC \ sig \\
&\quad else \ dynmethd \ \Gamma \ Object \ dynC \ sig
\end{aligned}
$$

At runtime, a reference to an interface will actually be an object of class *dynC*. We do not know which static class this reference had. We just know that it has to implement the interface. On interfaces we can call the methods defined by the interface itself (including multiple inheritance of interfaces) and those defined in *Object*. If the method is defined in the interface (*imethds* Γ *I sig* $\neq \{\}$) we can assume that the method is implemented by the dynamic class *dynC*. As we lack the static class we cannot use the function *dynmethd* for method lookup. However, this is not a problem here, since all interface methods are *Public*. We know that a method corresponding to the signature *sig* has to be **member-of** the dynamic class *dynC* itself, since *Public* methods are always inherited or overridden by other *Public* methods. The method cannot be blocked by package boundaries. That is why we can use *methd* here. If we have called an *Object* method which is not overridden by an interface method we just use *dynmethd* with *Object* as static class. Looking at the description of class `Object` in the JLS we see that there are just *Public* and *Protected* methods there, no *Private* or *Package* methods. Therefore we could again reduce *dynmethd* Γ *Object dynC sig* to *methd* Γ *dynC sig* with the same argumentation as above. Our method lookup is a little bit more general than it has to be.

5. Runtime Properties

5.1. Dynamic Accessibility of Members

In an object-oriented setting it is usual that we can receive an object of class B at runtime if we statically expect a reference to an object of class A. Class B then has to be a subclass of A. In Java, it is possible that A is declared *Public* but B is not. So we can receive an object of class B outside of its packages, although B is not statically accessible. As accessibility of the class is a precondition for static accessibility of a member, we cannot expect that at runtime only the statically accessible members are the members valid to access. This will be illustrated by the next example.

```
package P;                                  package Q;
public class A {                            import P.A;
   public void foo(){}                      import P.C;
}                                           class B extends A {
                                               void do() {
package P;                                       B b = new B();
public class C {                                 C.callfoo(b)
   public static void callfoo(A b) {          }
      b.foo();                              }
   }
}
```

The class `B` is not *Public* in package `Q` and extends class `A`. From package P we cannot even statically access class `B` and hence least of all any of its members. Therefore any access to `B.foo()` in package P is prohibited by the static accessibility rules. The method call `b.foo()` in class `C` statically attempts to access `A.foo()`, since the parameter `b` has the static type `A`. This conforms to static accessibility. But evaluation of method `do()` will pass a reference of type `B` to the parameter `b` of the method `callfoo()`. This will lead to a runtime access to `B.foo()` which is forbidden by static accessibility. That is why we need a more liberal predicate to capture the runtime properties we should expect: $\Gamma \vdash m$ *in* C *dyn-accessible-from* $accC$. In context of program Γ member m of class C is dynamically accessible from class $accC$. The dynamically valid member accesses are determined by the two following rules.

$$\frac{\begin{array}{c} \Gamma \vdash m \text{ \emph{member-in} } C \\ \Gamma \vdash m \text{ \emph{in} } C \text{ \emph{permits-acc-from} } accC \end{array}}{\Gamma \vdash m \text{ \emph{in} } C \text{ \emph{dyn-accessible-from} } accC} \quad \text{(IMMEDIATE)}$$

$$\frac{\begin{array}{c} \Gamma \vdash (declC,newM) \text{ \emph{overrides} } old \\ new = (declC,mdecl\ newM) \qquad \Gamma \vdash new \text{ \emph{member-in} } C \\ \Gamma \vdash C \prec_C S \qquad \Gamma \vdash old \text{ \emph{in} } S \text{ \emph{dyn-accessible-from} } accC \end{array}}{\Gamma \vdash new \text{ \emph{in} } C \text{ \emph{dyn-accessible-from} } accC} \quad \text{(OVERRIDING)}$$

These rules of dynamic accessibility resemble the rules of static accessibility, but leave out the precondition that the types must be accessible and switch from *member-of* to *member-in* and from static overriding (*overrides_S*) to dynamic overriding (*overrides*). We want to ensure that for a wellformed program (where only statically accessible members are accessed at compile-time) only dynamically accessible members are accessed at runtime. Static accessibility is tested by the compiler to decide whether a given program is wellformed or not. It can also be tested by the bytecode verifier to decide whether or not to run a program. Interestingly, current bytecode verifiers do not care about these accessibility concerns at all. Instead the JVM performs some runtime checks. In our model static accessibility is incorporated into the type system. Dynamic accessibility captures the properties of the actual member accesses that can occur during execution of the wellformed program. Dynamic accessibility is integrated into the operational semantics as special tests. They will cause an error if dynamic accessibility is violated at runtime. The OVERRIDING rule for the dynamic case is not as questionable as for the static case. If a method overrides another one it will be called anyway, due to dynamic binding, and therefore we have to accept such calls during runtime. Regardless whether we support the OVERRIDING rule in the static case or not, in the dynamic case we have to deal with it. Only in one scenario we can omit the OVERRIDING rule for both the dynamic and static case: if we enforce that a *Protected* method can only be overridden outside of its package by a *Public* method, all legal accesses are captured by the IMMEDIATE rules.

5.2. Main Runtime Theorem

After these auxiliary definitions for dynamic overriding and accessibility, we will now formulate the main theorem about the connection of static and dynamic accessibility. We model the runtime behaviour of Java with a big step semantics. Whenever the dynamic accessibility is violated we throw a special abruption that halts the program and signals the error. The following theorem states that this situation will never occur when executing wellformed programs.

Theorem: $[\![\Gamma \vdash s_0 -t \succ \to (v, s_1); (\!|prg{=}\Gamma, cls{=}accC, lcl{=}L|\!) \vdash t :: T; \textit{wf-prog } \Gamma;$
$s_0 {::} \preceq (\Gamma, L)]\!] \Longrightarrow \textit{error-free } s_0 = \textit{error-free } s_1$

The assumptions of the theorem are written in the brackets $[\![\dots]\!]$ and are separated with a semicolon. The conclusion comes after the implication $\Longrightarrow$ of Isabelle's meta logic. The assumptions of the theorem can be explained as follows. Evaluating the Java term t leads us from state s_0 to state s_1 and gives us v as result. Java statements and expressions are generalised to terms in this semantics. Statements evaluate to a dummy result. The term t is welltyped in the body of class $accC$ ($(\!|prg{=}\Gamma, cls{=}accC, lcl{=}L|\!) \vdash t :: T$) and the whole program is wellformed. This guarantees that only statically accessible members are accessed. Finally the starting state conforms to the environment ($s_0 {::} \preceq (\Gamma, L)$). This implies that all values within the state are compatible with their static types. L is the static typing environment for local variables. Abrupt completion is encoded into the state. Under these assumptions the conclusion of the theorem then guarantees that if we start in a error-free state we will end up in an error-free state (no access violation has occurred during evaluation). The state is a

pair of type *abrupt option × store*. We have defined the corresponding selectors *abrupt* and *store* for this pair. The *abrupt*-component signals all kinds of possible reasons for an abrupt completion: Exceptions, `break`, `continue`, `return` and additional artificial runtime errors. The usual reasons for abrupt completion can be caught and handled with ordinary language constructs. The additional runtime errors cannot. We aim to prove that they cannot occur in a wellformed and welltyped program. The function *error-free* just ensures that no error is present in the state:

$$\text{error-free } s \equiv \neg \ (\exists \ err. \ abrupt \ s = Some \ (Error \ err))$$

The proof of the theorem is closely related to the type safety proof in [8]. For a better technical understanding of the theorem we will now have a closer look at how static accessibility is integrated into the type system, how dynamic accessibility is integrated into the operational semantics, how the theorem above connects these two levels, and where type safety comes in. As an example we will examine access to an object field, written *e..fn* for an expression *e* and a field name *fn* in our Java model. The typing rule for access to a field variable is the following:

$$\frac{\begin{array}{c} (\!|prg{=}\Gamma,cls{=}accC,lcl{=}L|\!) \vdash e{::}{-}Class \ statC \\ accfield \ \Gamma \ accC \ statC \ fn \ = \ Some \ (statDeclC,f) \end{array}}{(\!|prg{=}\Gamma,cls{=}accC,lcl{=}L|\!) \vdash \{accC,statDeclC,\text{is-static } f\}e..fn{::}{-}(type \ f)}$$

The expression *e* must be of static type *Class statC* and this class *statC* must contain an accessible field with name *fn*: the lookup of the field name *accfield Γ accC statC fn* yields a pair *(statDeclC,f)* where field *f* is declared in class *statDeclC* which may be a superclass of class *statC*. The type of the whole field access is the type of the field itself. The function *accfield* only searches the accessible fields. It guarantees static accessibility: Γ⊢**Field *fn* (*statDeclC*,*f*) of *statC* accessible-from** *accC*. Another point worth mentioning is that the syntax of the field access *e..fn* is enriched with annotations in braces: {*accC*,*statDeclC*,*is-static f*}. The accessing class *accC*, the declaration class of the Field *statDeclC*, and a flag whether the field is static or not (*is-static f*) is encoded into the syntax. We can then access these entities in the operational semantics to actually perform the field access and execute the sanity checks. This becomes clearer if we look at the evaluation rule of field access:

$$\frac{\begin{array}{c} \Gamma \vdash (None,s_0) \ -Init \ statDeclC \rightarrow s_1 \\ \Gamma \vdash s_1 \ -e{-}\succ a \rightarrow s_2 \\ (v,s_3) = fvar \ statDeclC \ stat \ fn \ a \ s_2 \\ s_4 = check\text{-}field\text{-}access \ \Gamma \ accC \ statDeclC \ fn \ stat \ a \ s_3 \end{array}}{\Gamma \vdash (None,s_0) \ -\{accC,statDeclC,stat\}e..fn \succ \rightarrow (v,s_4)}$$

Starting in an initial state (*None*, s_0), a state where no abruption is present, evaluation of the field access {*accC*,*statDeclC*,*stat*}*e..fn* will result in the value *v* and the final state s_4. The operational semantics models class initialisation. Thus we first trigger this class initialisation of the static declaration class *statDeclcC* of the field. Then we evaluate the expression *e* to its

address a and look up the value of the field with the auxiliary function *fvar* which may cause a null-pointer exception and therefore potentially updates the state. Finally *check-field-access* tests if the current field is dynamically accessible and otherwise throws an error. Here, the annotations {*accC*,*statDeclC*,*stat*} of the typing rule serve as parameters. Keep in mind that we aim to prove that *check-field-access* will never actually throw an error in a wellformed and welltyped program. The definition of the auxiliary function *check-field-access* is quite tedious because it has to handle static and dynamic fields and involves some details about the store.

check-field-access Γ *accC statDeclC fn stat a s* $\equiv$
   let *oref*   = if *stat* then *Stat statDeclC* else *Heap* (*the-Addr a*);
      dynC = case *oref* of
               Heap a $\Rightarrow$ *obj-class* (*the* (*globs* (*store s*) *oref*))
              | *Stat C* $\Rightarrow$ *C*;
      f   = (*the* (*table-of* (*fields* Γ *dynC*) (*fn*,*statDeclC*)))
   in *abupd*
       (*error-if* ($\neg\Gamma\vdash$**Field** *fn* (*statDeclC*,*f*) **in** *dynC* **dyn-accessible-from** *accC*)
               AccessViolation)
      s

First we obtain the object reference *oref*. Depending on the flag *stat*, the field we want to access can be either a static field of class *statDeclC* or an instance field of an object. Object references in our formalisation generalise these two cases in a new datatype with the constructors *Stat* or *Heap*. For static fields, which are stored once per class, the reference is just the class name *statDeclC* itself. For object fields we look up the reference in the heap at address a. Next we calculate the dynamic class *dynC* of the reference, which is stored in the heap for instance fields, and the class *statDeclC* itself for static fields. Then we look up the field f with the extended name (*fn*,*statDeclC*) in the field map of the dynamic class *dynC*. We have to extend the field name with the static declaration class of the field, since fields may be overloaded in Java. The final step is to check whether the access to the field is valid or not and to cause the evaluation error *AccessViolation* if dynamic accessibility is violated. The function *error-if* performs the test and *abupd* updates the state according to the outcome of the test.

If we want to prove that no errors can occur, we have to show that the predicate $\Gamma\vdash$**Field** *fn* (*statDeclC*,*f*) **in** *dynC* **dyn-accessible-from** *accC* holds at runtime. We can assume that the type system has already checked the static accessibility condition: $\Gamma\vdash$**Field** *fn* (*statDeclC*,*f*) **of** *statC* **accessible-from** *accC*. We have to prove that we can switch from class *statC* to class *dynC* and from static to dynamic accessibility. If we refer to the definitions of static and dynamic accessibility we have to ensure that the field we want to access is still defined in the dynamic class and that access is permitted. Both can be guaranteed if we can rely on the fact that the dynamic class *dynC* is a subclass of (or the same class as) *statC*: $\Gamma\vdash dynC \preceq_C statC$. This is implied by type safety.

For method accesses the basic ideas and lines of reasoning are the same, but here we must also deal with dynamic binding and therefore with overriding. The basic corner-stone of the proof is again type safety. Additionally we need a lot of case distinctions for interface methods, instance methods, static methods etc. and have to relate static overriding with dynamic overriding. Therefore we omit the technical details of this proof.

5.3. Derived Runtime Properties

As we know now that dynamic accessibility is guaranteed during execution of wellformed programs, we can look at some derived, more concise properties we can rely on at runtime.

    Lemma: $\llbracket \Gamma \vdash m$ *in* C **dyn-accessible-from** $accC$; $accmodi\ m = Private \rrbracket$
                      $\implies accC = declclass\ m$

A *Private* member can only be accessed from its declaration class. This is a simple conclusion from the definition of **permits-acc-from**. Since overriding cannot occur for *Private* members, we do not need any further static wellformedness conditions here.

    Lemma: $\llbracket \Gamma \vdash m$ *in* C **dyn-accessible-from** $accC$; $accmodi\ m = Package$;
                    $wf\text{-}prog\ \Gamma \rrbracket \implies pid\ accC = pid\ (declclass\ m)$

A *Package* member can only be accessed from inside the package. This obviously is a desirable property. For fields it is a simple conclusion from the definition of **permits-acc-from**. For methods it is rather involved, since we have to deal with dynamic overriding. As mentioned before, dynamic overriding does not ensure that the access modifier of the overriding method is at least as liberal as the modifier of the overridden method. By this we can override a *Public* method with a *Package* method and violate the lemma. But for wellformed programs (*wf-prog* Γ) we know that static overriding is only allowed if the modifier is at least as liberal. From a detailed analysis of the similarities and connections between static and dynamic overriding we can show that the restrictions for static overriding suffice to prove the lemma.

Unfortunately the JVM does not ensure these restrictions for static overriding, neither during bytecode verification nor via a runtime test. Therefore the bytecode verifier accepts programs that are not necessarily wellformed in this sense and the JVM executes them. In hand-written bytecode it is possible to call a *Package* method that overrides a *Public* method from outside of the package without an error. Consider the following example:

```
package P;                              package Q;
public class A {                        import P.A;
   public void foo(){}                  public class B extends A {
}                                           void foo(){} // not wellformed
                                                         // static overriding
package P;                              }
import Q.B;
public class C {
  public void callfoo() {
    B b = new B();
    b.foo(); // JVM will call
    // B.foo() without complaining.
  }
}
```

In class B we override the *Public* method `A.foo()` with the *Package* method `B.foo()`. A proper Java compiler will reject this program as not wellformed. But by subsequent separate compilation and modification of the Java classes we can cheat the compiler and get the corresponding class files out of it. The JVM will not complain about anything and so evaluation of `C.callfoo()` will access `B.foo()` from outside of its package. Neither the bytecode verifier nor a runtime check will signal an error.

The following lemma captures a property for *Protected* members.

Lemma: $[\![\Gamma \vdash f \text{ in } C \text{ \emph{dyn-accessible-from}} \ accC; \ accmodi \ f = Protected;$
$is\text{-}field \ f; \ \neg \ is\text{-}static \ f; \ pid \ (declclass \ f) \neq pid \ accC]\!]$
$\implies \Gamma \vdash C \preceq_C accC$

Outside of the package a *Protected* instance field can only be accessed from a superclass. This is also a rather simple conclusion from the definition of **permits-acc-from**. Unfortunately this lemma cannot be extended to methods due to overriding. Consider the following example:

```
package P;                          package Q;
public class A {                    import P.A;
    protected void foo(){}          public class B extends A {
}                                       protected void foo(){}
                                    }
package P;
import Q.B;
public class C {
  public void callfoo(A b) {
    b.foo();
  }
  public void do() {
    B b = new B();
    callfoo(b);
  }
}
```

If we invoke `C.callfoo(b)` with an object `b` of class `B`, like in method `C.do()`, class `C` will actually access the *Protected* method `B.foo()` of a different package without being a superclass of `B`. This problem can be avoided if we would force the modifier of `B.foo()` to be `public` instead of just `protected`.

6.   Conclusion

In this article we clarified the semantics of Java access modifiers and packages by formalising them in the theorem prover Isabelle/HOL and proving some key properties. Our model reflects the subtle interaction between inheritance, overriding, and accessibility in Java. The Java

formalisation [8] was mature enough to let us add and analyse access modifiers. This shows that it is feasible to formally investigate aspects of a realistic programming language in a theorem prover. Although Java technology is now available for almost eight years, oversights and lack of clarity still exist in the language specification. Even in its second edition some aspects of the semantics remained unclear and ambiguous. Therefore different implementations of overriding at the compiler and JVM level have arisen. Also various Java compilers disagree on whether or not to support the OVERRIDING rule. This causes portability problems among different implementations, which is unsuitable for a language like Java where portability is a major design goal. Most of the sophisticated rules introduced in this article became apparent only during theorem proving. Initially, using simpler rules, we failed to prove some expected properties. The use of proof assistants therefore help in the design of a clear and unambiguous programming language semantics. Working with a proof assistant forces the user to be precise and unambiguous. Otherwise it becomes infeasible or impossible to prove even some basic properties. During theorem proving, problems that have been overlooked become apparent, particularly for large specifications like the JLS.

From a language design point of view, this work indicates a general problem of object-oriented languages that combine subtyping, module based encapsulation, and inheritance, using the concept of classes. The tension between hiding information behind module boundaries and inheriting members over them causes a lot of trouble. The problems with the `protected` modifier make this sufficiently clear. Therefore an approach which keeps the concepts of subtyping, modules, classes, and inheritance apart [2] can avoid such tedious complications.

ACKNOWLEDGEMENTS

REFERENCES

1. Sophia Drossopoulou and Susan Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1532 of *LNCS*, pages 41–82. Springer, 1999.
2. Kathleen Fisher and John H. Reppy. The design of a class mechanism for moby. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 37–49, 1999.
3. James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.
4. Marieke Huisman. *Java program verification in higher order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2000.
5. Thomas Jensen, Daniel Le Métayer, and Tommy Thorn. Security and dynamic class loading in Java: A formalisation. In *IEEE International Conference on Computer Languages*, pages 4–15, Chicago, Illinois, 1998.
6. Peter Müller and Arnd Poetzsch-Heffter. Kapselung und Methodenbindung: Javas Designprobleme und ihre Korrektur. In C. H. Cap, editor, *JIT '98 Java-Informations-Tage 1998*, Informatik Aktuell. Springer, 1998.

Concurrency Computat.: Pract. Exper. 2003; **0**:1–10

7. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* Springer, 2002. LNCS 2283.
8. David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Universität München, 2001.
9. Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer, 2001.
10. SUN. Java Developer Connection. Available from http://java.sun.com/jdc.
11. Don Syme. Proving Java Type Soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1532 of *LNCS*, pages 83–118. Springer, 1999.
12. Verificard at Munich. Available from http://isabelle.in.tum.de/verificard.