A Programming Logic for Sequential Java

Arnd Poetzsch-Heffter and Peter Müller

Fernuniversität Hagen, D-58084 Hagen, Germany
[Arnd.Poetzsch-Heffter, Peter.Mueller]@Fernuni-Hagen.de

Abstract. A Hoare-style programming logic for the sequential kernel of Java is presented. It handles recursive methods, class and interface types, subtyping, inheritance, dynamic and static binding, aliasing via object references, and encapsulation. The logic is proved sound w.r.t. an SOS semantics by embedding both into higher-order logic.

1 Introduction

Java is a practically important object-oriented programming language. This paper presents a logic to verify sequential Java programs. The motivations for investigating the logical foundations of Java are as follows:

1. Java plays an important role in the quickly developing software component industry and the smart card technology. Verification techniques can be used for static program analysis, e.g., to prove the absence of null-pointer exceptions. The Java subset used in this paper is similar to JavaCard, the Java dialect for implementing smart cards.
2. As pointed out in [MPH97], logical foundations of programming languages form a basis for program specification technology. They allow for expressive specifications (covering e.g., abstraction, sharing-properties, and side-effects) and are needed to assign a formal meaning to interface specifications.
3. Formality is a prerequisite for tool-based verification. Tool support is necessary to keep large program proofs error-free.
4. Java is typical for a large group of OO-languages including C++, Eiffel, Oberon, Modula-3, BETA, and Ada95. The developed techniques can be adapted to other languages of that group.

The goal underlying this research is the development of interactive programming environments that support specification and verification of OO-programs. Some design decisions have been made w.r.t. this goal.

Approach. Three aspects make verification of OO-programs more complex than verification of programs with just recursive procedures and arbitrary pointer data structures: Subtyping, abstract types, and dynamic binding. Subtyping allows variables to hold objects of different types. Abstract types have different or incomplete implementations. Thus, techniques are needed to formulate type properties without referring to implementations. Dynamic binding destroys the static connection between the calls and the body of a procedure.

Our solutions to these problems build on well-known techniques. Object stores are specified as an abstract data type with operations to create objects and to read and write instance variables/attributes. Based on object stores, abstractions of object structures can be expressed which are used to specify the behavior of abstract types. To express the relation between stores in pre- and poststates, the current object store can be referenced through a special variable.

Our programming logic refines Hoare logics for procedural languages. To handle dynamic binding, the programming logic allows one to prove properties of so-called virtual methods, i.e., methods that capture the common properties of the corresponding methods in subtypes. The distinction between the virtual behavior of a method and the behavior of the associated implementations allows one to transfer verification techniques for procedures to OO-programs.

The logic is proved sound w.r.t. an SOS semantics of the programming language. Since the semantics of modern OO-languages tends to be rather complex, such soundness proofs can become quite long for full-size languages and should therefore be checkable by mechanical proof checkers. To provide a basis for mechanical checking, we embed both semantics into a higher-order logic and derive the axioms and rules of the logic from those of the operational semantics.

Related Work. In [Lei97], a wlp-calculus for an OO-language similar to our Java subset is presented. In contrast to our work, method specifications are part of the programs. The approach in [Lei97] can be considered as restricting our approach to a certain program development strategy (in [PHM98], we discuss this topic). Thereby, it becomes simpler and more appropriate for automatic checking, but gives up flexibility that seems important to us for interactive program development and verification. A different logic for OO-programs that is related to type-systems is presented and proved sound in [AL97]. It is developed for an OO-language in the style of the lambda calculus whereas we are aiming to directly support the verification of an existing practical language. The presented programming logic extends the foundations developed in [PHM98] by covering encapsulation and subclassing. Furthermore, [PHM98] does not discuss the relation between the logic and a formal semantics and does not prove soundness.

In [vON98], type-safety is formally proved for a Java subset similar to ours. Corresponding to our soundness proof, both operational semantics and typing rules are formalized in higher-order logic. However, the type-safety proof has already been mechanically checked in Isabelle. [JvdBH$^+$98] uses an operational semantics of a Java subset to verify various properties of implementations with the PVS proof checker without employing an axiomatic semantics. As will become clear from the presented paper, Hoare-logic provides an additional level of abstraction. This simplifies the handling of subtyping and abstract methods, and proofs become more intuitive. In practice, verification requires elaborate specification techniques like the one described in [Lea96]. In [MPH97], we outline the connection between such specifications and our logic.

Overview. Section 2 presents the operational semantics of the Java kernel, section 3 the programming logic. The soundness proof is contained in Sect. 4.

2 A Semantics for Sequential Java

This section describes the sequential Java kernel, Java-K for short, and presents its dynamic semantics. Compared to Java, Java-K supports only a simple expression and statement syntax, but captures the full complexity of the method invocation semantics. As specification technique we use structural operational semantics (SOS). We assume that the reader is familiar with Java and explain only the restrictions of Java-K. The formal presentation concentrates on those aspects that are needed for the soundness proof in Sect. 4.

Java-K Programs. A Java-K program is a set of type declarations where a type is either a class or an interface. A class declares its name, its superclass, the list of interfaces implemented by the class, and its members. A member is a field, instance method, or static method. Members can be public, protected, or private. Java-K provides the default constructor, but does not support constructor definitions. Method declarations contain the access mode, the method signature, a list of local variables, and a statement as method body. To keep things simple, methods in Java-K have exactly one parameter named p and have always a return type. Overloading is not allowed. The return type, the name of the method, and the parameter of the methods are given by the so-called method signature. An interface declares its name, the list of extended interfaces, and the signatures of its methods:

data type

```
JavaK-Program = list of  TypeDecl
TypeDecl      = ClassDecl( CTypeId  CTypeId  ITypeIdList  ClassBody )
              | InterfaceDecl( ITypeId  ITypeIdList  InterfaceBody )
ClassBody     = list of  MemberDecl
MemberDecl    = FieldDecl( Mode  Type  FieldId )
              | MethodDecl ( Mode  MethodSig  VarList  Statement )
              | StaticMethDecl ( Mode  MethodSig  VarList  Statement )
Mode          = Private() | Protected() | Public()
InterfaceBody = list of  MethodSig
ITypeIdList   = list of  ITypeId
MethodSig     = Sig( Type  MethodId  Type )
VarList       = list of  VarDecl
VarDecl       = Vardcl( Type  VarId )
Type          = booleanT() | intT() | nullT() | ct( CTypeId ) | it( ITypeId )
```

Java-K has the predefined types *booleanT*, *intT*, and *nullT* (the type of the null reference), and the user defined class and interface types. The subtype relation on sort *Type* is defined as in Java and denoted by $\preceq$.

An expression in Java-K is an integer or boolean constant, the null reference, a variable or parameter identifier, the identifier “this” (denoting the reference to the object for which the non-static method was invoked), or a unary or binary expression over relational or arithmetic operators. The statements of Java-K are defined below along with their dynamic semantics.

Capturing Statement Contexts. The semantics of a statement depends on the context of the statement occurrence. We assume that the program context of a statement is always implicitly given and that we can refer to method declarations in this context. Method declarations are denoted by T@m where m is a method name in class T. *MethDeclId* is the sort of such identifiers. The function

$$\mathit{body} : \ \mathit{MethDeclId} \rightarrow \mathit{Statement}$$

maps each method declaration to the statement constituting its body. If T is a class type and m a method of T, the function

$$\mathit{impl} : \ \mathit{Type} \times \mathit{MethodId} \rightarrow \mathit{MethDeclId} \cup \{\mathit{undef}\}$$

yields the corresponding declaration; otherwise it yields *undef*. Note that T can inherit the declaration of m from a superclass. Similarly to method declaration identifiers, we introduce field declaration identifiers of the form T@a where a is a field name in class T. The sort of such identifiers is denoted by *FieldDeclId*. They are needed to distinguish instance variables with the same field name occurring in one object.

States. A statement is essentially a partial state transformer. A state in Java-K describes (a) the current values for the local variables and for the method parameters p and this, and (b) the current object store. Values in Java-K are either integers, booleans, the null reference, or references to objects of a class type:

$$\begin{array}{lll@{\qquad\qquad}lcl}
\multicolumn{3}{l}{\textbf{data type}} & \multicolumn{3}{l}{\tau : \mathit{Value} \rightarrow \mathit{Type}} \\
\mathit{Value} & = & b(\ \mathit{Bool}\ ) & \tau(b(B)) & = & \mathit{booleanT} \\
 & \mid & i(\ \mathit{Int}\ ) & \tau(i(I)) & = & \mathit{intT} \\
 & \mid & \mathit{null}() & \tau(\mathit{null}) & = & \mathit{nullT} \\
 & \mid & \mathit{ref}(\ \mathit{CTypeId}, \mathit{ObjId}\ ) & \tau(\mathit{ref}(T, \mathit{OI})) & = & \mathit{ct}(T)
\end{array}$$

Values constructed by *ref* represent the references to objects. The sort *ObjId* denotes some suitable set of object identifiers to distinguish different objects of the same type. The function τ yields the type of a value.

The state of an object is given by the values of its instance variables. We assume a sort *InstVar* for the instance variables of all objects and a function

$$\mathit{instvar} : \ \mathit{Value} \times \mathit{FieldDeclId} \rightarrow \mathit{InstVar} \cup \{\mathit{undef}\}$$

where *instvar*(V,T@a) is defined as follows: If V is an object reference and the corresponding object has an instance variable named T@a, this instance variable is returned. Otherwise *instvar* yields *undef*. The state of all objects and the information whether an object is alive (i.e., allocated) in the current program state is formalized by an abstract data type Object Store with sort *Store* and the following functions:

$$\begin{array}{lll}
\_\langle \_ := \_ \rangle & : \mathit{Store} \times \mathit{InstVar} \times \mathit{Value} & \rightarrow \ \mathit{Store} \\
\_\langle \_ \rangle & : \mathit{Store} \times \mathit{CTypeId} & \rightarrow \ \mathit{Store} \\
\_(\_) & : \mathit{Store} \times \mathit{InstVar} & \rightarrow \ \mathit{Value} \\
\mathit{alive} & : \mathit{Value} \times \mathit{Store} & \rightarrow \ \mathit{Bool} \\
\mathit{new} & : \mathit{Store} \times \mathit{CTypeId} & \rightarrow \ \mathit{Value}
\end{array}$$

$OS\langle IV := V\rangle$ yields the object store that is obtained from OS by updating instance variable IV with value V. $OS\langle T\rangle$ yields the object store that is obtained from OS by allocating a new object of type T. $OS(IV)$ yields the value of instance variable IV in store OS. If V is an object reference, $alive(V, OS)$ tests whether the referenced object is alive in OS. $new(OS, TID)$ yields a reference to an object of type $ct(TID)$ that is not alive in OS. Since the properties of these functions are not needed for the soundness proof in Sect. 4, we do not discuss their axiomatization here and refer the reader to [PHM98].

Program states are formalized as mappings from identifiers to values. To have a uniform treatment for variables and the object store, we use \$ as identifier for the current object store:

$$State \;\equiv\; (\mathit{VarId} \cup \{\,\mathrm{this},\ \mathrm{p}\,\} \rightarrow \mathit{Value} \cup \{\mathit{undef}\}) \times (\{\,\$\,\} \;\rightarrow\; \mathit{Store} \cup \{\mathit{undef}\})$$

For $S \in State$, we write $S(\mathrm{x})$ for the application to a variable or parameter identifier and $S(\$)$ for the application to the object store. By $S[\mathrm{x} := V]$ and $S[\$:= OS]$ we denote the state that is obtained from S by updating variable x and \$, respectively. The canonical evaluation of expression e in state S is denoted by $\epsilon(S, \mathrm{e})$ yielding an element of sort *Value* or *undef* (note that expressions in Java-K always terminate and do not have side-effects). The state in which all variables are undefined is named *initS*.

Statement Semantics. The semantics of Java-K statements is defined by inductive rules. $S : \;\mathrm{s}\; \rightarrow S'$ expresses the fact that executing *Statement* s in *State* S terminates in *State* S'. In the rules, x and y range over variable or parameter identifiers, and e over expressions.

In order to keep the size of the specification manageable, we assume that some Java-K statements are given in a syntax that is decorated with information from type and name analysis. An access to an instance variable a of static type T is written as T@a. Java-K provides statements for reading and writing instance variables with the following semantics (note that the context conditions of Java and the antecedent of the rule guarantee that $instvar(\mathrm{y}, \mathrm{T@a})$ is defined):

$$\frac{S(\mathrm{y}) \neq \mathit{null}}{S : \;\mathrm{x = y.T@a;}\; \rightarrow S[\mathrm{x} := S(\$)(\mathit{instvar}(\mathrm{y}, \mathrm{T@a}))]}$$

$$\frac{S(\mathrm{y}) \neq \mathit{null}}{S : \;\mathrm{y.T@a{=}e;}\; \rightarrow S[\$:= S(\$)\langle \mathit{instvar}(\mathrm{y}, \mathrm{T@a}) := \epsilon(S, \mathrm{e})\rangle]}$$

In Java, there are four kinds of method invocations: (a) invocations of public or protected methods, (b) invocations of private methods, (c) invocations of superclass methods, and (d) invocations of static methods. Invocations of kind (a) are dynamically bound, the others can be bound at compile time. To make the context information visible within the SOS rules, we distinguish kind (a) and (b) syntactically: y.T:m(e) denotes an invocation of kind (a) where T is the static type of y. A statically bound invocation is denoted by y.T@m(e) where T is the class in which m is declared. We can use the same syntax to handle invocations of superclass methods: A Java method invocation of the form super.m(e) occurring in a class C is in Java-K expressed by a call this.CSuper@m() where CSuper is the nearest superclass of C containing a declaration of m. This way, semantics of invocations of kind (b) and (c) can be given by the same rule. Invocations of kind (d) behave similar, but do not have a this-parameter.

To focus on the interesting aspects, Java-K does not support a return statement. The return value has to be assigned to a local variable "result" that is implicitly declared in all methods. Thus, method invocation means passing the parameters and the object store to the prestate, executing the invoked method, and passing the result and object store back to the invocation environment:

$$\frac{\begin{array}{c} S(\mathrm{y}) \neq \mathit{null},\ \tau(S(\mathrm{y})) \preceq \mathrm{T}, \\ \mathit{initS}[\mathrm{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S, \mathrm{e}), \$:= S(\$)] :\ \mathit{body}(\mathit{impl}(\tau(S(\mathrm{y})), \mathrm{m}))\ \rightarrow S' \end{array}}{S :\ \mathrm{x{=}y.T{:}m(e);}\ \rightarrow S[\mathrm{x} := S'(\mathrm{result}),\ \$:= S'(\$)]}$$

$$\frac{S(\mathrm{y}) \neq \mathit{null},\ \mathit{initS}[\mathrm{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S, \mathrm{e}), \$:= S(\$)] :\ \mathit{body}(\mathrm{T@m})\ \rightarrow S'}{S :\ \mathrm{x{=}y.T@m(e);}\ \rightarrow S[\mathrm{x} := S'(\mathrm{result}),\ \$:= S'(\$)]}$$

The rule for the invocation of static methods is identical to the last rule, except that no this-parameter has to be passed. Besides the statements described above, Java-K provides if and while statements, assignment statements with cast, sequential statement composition, and constructor calls. The rules for these statements are straightforward and given in the appendix.

3 A Programming Logic for Java

This section presents a Hoare-style programming logic for Java-K. The logic allows one to formally verify that implementations satisfy interface specifications. For OO-languages, interface specifications are usually given by pre- and postconditions for methods, class invariants, history constraints, etc (cf. e.g. [Lea96]). The formal meaning of such specifications is defined in terms of proof obligations for methods (cf. [PH97]). In this paper, we concentrate on the verification of dynamic properties. For proving properties about the object store, we refer to [PH97]. This section defines the precise syntax of our Hoare triples and explains the axioms and rules of the programming logic.

Specifying Methods and Statements. Properties of methods and statements are expressed by triples of the form $\{\mathbf{P}\}\ \mathrm{comp}\ \{\mathbf{Q}\}$ where $\mathbf{P}$, $\mathbf{Q}$ are sorted first-order formulas and comp is either a statement occurrence within a given Java-K program, a method implementation represented by the corresponding method declaration identifier, or a so-called *virtual method*. Before we clarify the signature over which $\mathbf{P}$, $\mathbf{Q}$ are built, we explain the concept of virtual methods.

Virtual Methods. Java-K supports dynamically bound method invocations. E.g., if T is an interface type, and T1, T2 are classes implementing T, an invocation y.T:m(e) can lead to the execution of T1@m or T2@m depending of the object held by y. To verify dynamically bound method invocations, we need method specifications reflecting the properties of all implementations that might be executed. Such specifications express the behavior of the so-called *virtual methods*. For every non-private instance method m declared in or inherited by a type T, there is a virtual method denoted by T:m. (This notation corresponds to the syntax used for the invocation semantics in Sect. 2.) For private and static methods, virtual methods are not needed, because statically bound invocations can be directly handled using the properties of the corresponding method bodies.

Signatures of Pre- and Postconditions. In program specifications, we have to refer to types, fields, and variables in pre- and postconditions. We enable that by introducing constant symbols for these entities. For a given Java-K program, Σ denotes the signature of sorts, functions, and constant symbols as described in Sect. 2. In particular, it contains constant symbols for the types and fields. Furthermore, we treat parameters, program variables, and the variable \$ for the current object store syntactically as constant symbols of sort *Value* and *Store* to simplify quantification and substitution rules and to define context conditions for pre- and postconditions.

A triple $\{ \mathbf{P} \}$ comp $\{ \mathbf{Q} \}$ is called a *statement annotation*, *implementation annotation*, or *method annotation* if the syntactical component comp is a statement, method implementation, or virtual method, respectively. Pre- and postconditions of statement annotations are formulas over $\Sigma \cup \{\mathrm{this}, \mathrm{p}, \$\} \cup \mathit{VAR}(\mathrm{m})$ where m is the method enclosing the statement and $\mathit{VAR}(\mathrm{m})$ denotes the set of local variables of m. Preconditions in method annotations or implementation annotations are formulas over $\Sigma \cup \{\mathrm{this}, \mathrm{p}, \$\}$. Postconditions in such annotations are formulas over $\Sigma \cup \{\mathrm{result}, \$\}$.

To handle recursive methods, we use *sequents* of the form $\mathcal{A} \rhd \mathbf{A}$ where $\mathcal{A}$ is a set of method and implementation annotations and $\mathbf{A}$ is a triple. Triples in $\mathcal{A}$ are called *assumptions* of the sequent and $\mathbf{A}$ is called the *consequent* of the sequent. Intuitively, a sequent expresses the fact that we can prove a triple based on some assumptions about methods.

Axiomatic Semantics. The axiomatic semantics of Java consists of axioms and rules for statements and methods. The new axioms and rules are described in the following two paragraphs. The standard Hoare rules (e.g., while rule) are presented in Fig. 1. A more detailed discussion of programming logics for OO-languages and their applications is given in [PHM98].

Statements. The cast-axiom is very similar to Hoare's classical assignment axiom. However, to prevent runtime errors, a stronger precondition assures that the type conversion is legal. The constructor-axiom works like an assignment axiom: The new object is substituted for the left-hand-side variable and the modified object store for the initial store. Reading a field substitutes the value held by the addressed instance variable for the left-hand-side variable. Writing field access replaces the initial object store by the updated store:

cast-axiom: $\rhd \{\ \tau(\mathrm{e}) \preceq \mathrm{T} \wedge \mathbf{P}[\mathrm{e}/\mathrm{x}]\ \}\ \ \mathrm{x} = (\mathrm{T})\ \mathrm{e};\ \ \{\ \mathbf{P}\ \}$

constructor-axiom: $\rhd \{\ \mathbf{P}[\textit{new}(\$, \mathrm{T})/\mathrm{x}\ ,\ \$\langle \mathrm{T} \rangle/\$]\ \}\ \mathrm{x} = \mathrm{new\ T}();\ \{\ \mathbf{P}\ \}$

field-read-axiom: $\rhd \{\ \mathrm{y} \neq \textit{null} \wedge \mathbf{P}[\$(\textit{instvar}(\mathrm{y}, \mathrm{S@a}))/\mathrm{x}]\ \}\ \ \mathrm{x} = \mathrm{y.S@a};\ \ \{\ \mathbf{P}\ \}$

field-write-axiom: $\rhd \{\ \mathrm{y} \neq \textit{null} \wedge \mathbf{P}[\$\langle \textit{instvar}(\mathrm{y}, \mathrm{S@a}) := \mathrm{e} \rangle/\$]\ \}\ \ \mathrm{y.S@a} = \mathrm{e};\ \ \{\ \mathbf{P}\ \}$

The invocation-rule uses properties of virtual methods to verify invocations of dynamically bound methods. The fact that local variables different from the left-hand-side variable are not modified by an invocation is expressed by the *invocation-var-rule* that allows one to substitute logical variables Z in pre- and postconditions by local variables w (w different from x):

$$\textit{invocation-rule:}\quad \frac{\mathcal{A} \rhd \{\ \mathbf{P}\ \}\ \ \mathrm{T{:}m}\ \ \{\ \mathbf{Q}\ \}}{\mathcal{A} \rhd \{\ \mathrm{y} \neq \textit{null} \wedge \mathbf{P}[\mathrm{y/this, e/p}]\ \}\ \mathrm{x} = \mathrm{y.T{:}m(e)};\ \{\ \mathbf{Q}[\mathrm{x/result}]\ \}}$$

$$\textit{invocation-var-rule:}\quad \frac{\mathcal{A} \rhd \{\ \mathbf{P}\ \}\qquad\quad \mathrm{x} = \mathrm{y.T{:}m(e)};\ \ \{\ \mathbf{Q}\ \}}{\mathcal{A} \rhd \{\ \mathbf{P}[\mathrm{w}/Z]\ \}\ \ \mathrm{x} = \mathrm{y.T{:}m(e)};\ \ \{\ \mathbf{Q}[\mathrm{w}/Z]\ \}}$$

Static methods are bound statically. Therefore, method implementations are used instead of virtual methods to verify invocations. In a similar way, method implementations are used to verify calls of private methods and invocations using super. In both cases, the implementation to be executed can be determined statically. The var-rules for static invocations and calls can be found in Fig. 1.

$$\textit{static-invoc-rule:}\quad \frac{\mathcal{A} \rhd \{\ \mathbf{P}\ \}\ \ \mathrm{T@m}\ \ \{\ \mathbf{Q}\ \}}{\mathcal{A} \rhd \{\ \mathbf{P}[\mathrm{e/p}]\ \}\ \mathrm{x} = \mathrm{T.m(e)};\ \{\ \mathbf{Q}[\mathrm{x/result}]\ \}}$$

$$\textit{call-rule:}\quad \frac{\mathcal{A} \rhd \{\ \mathbf{P}\ \}\ \ \mathrm{T@m}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \rhd \{\ \mathrm{y} \neq \textit{null} \wedge \mathbf{P}[\mathrm{y/this, e/p}]\ \}\ \mathrm{x} = \mathrm{y.T@m(e)};\ \{\ \mathbf{Q}[\mathrm{x/result}]\ \}}$$

Methods. This paragraph presents the rules to prove properties of method implementations and virtual methods. Essentially, an annotation of a method implementation m holds if it holds for its body. In order to handle recursion, the method annotation may be assumed for the proof of the body. Informally, this is sound, because in any terminating execution, the last incarnation does not contain a recursive invocation of the method:

$$\textit{implementation-rule:}\quad \frac{\mathcal{A}\ ,\ \{\mathbf{P}\}\ \ \mathrm{T@m}\ \ \{\mathbf{Q}\} \rhd \{\mathrm{this} \neq \textit{null} \wedge \mathbf{P}\}\ \ \textit{body}(\mathrm{T@m})\ \ \{\mathbf{Q}\}}{\mathcal{A} \rhd \{\mathbf{P}\}\ \ \mathrm{T@m}\ \ \{\mathbf{Q}\}}$$

Virtual methods have been introduced to model dynamically bound methods. I.e., a method annotation for T:m reflects the common properties of all implementations that might be executed on invocation of T:m. If T is a class, there are two obligations to prove an annotation **A** of a virtual method T:m : 1. Show that the corresponding implementation satisfies **A** if invoked for objects of type T. 2. Show that **A** holds for objects of proper subtypes of T. The second obligation and annotations of interface type methods can be proved by the *subtype-rule*: If S is a subtype of T, an invocation of T:m on an S object is equivalent to an invocation of S:m. Thus, all properties of S:m carry over to T:m as long as T:m is applied to objects of type S:

class-rule:

$$\frac{\begin{array}{l}\mathcal{A} \rhd \{\ \tau(\text{this}) = \mathrm{T} \wedge \mathbf{P}\ \}\ \mathit{impl}(\mathrm{T,m})\ \{\ \mathbf{Q}\ \} \\ \mathcal{A} \rhd \{\ \tau(\text{this}) \prec \mathrm{T} \wedge \mathbf{P}\ \}\quad \mathrm{T{:}m}\quad \{\ \mathbf{Q}\ \}\end{array}}{\mathcal{A} \rhd \{\ \tau(\text{this}) \preceq \mathrm{T} \wedge \mathbf{P}\ \}\quad \mathrm{T{:}m}\quad \{\ \mathbf{Q}\ \}}$$

subtype-rule:

$$\frac{\begin{array}{l}\mathrm{S} \preceq \mathrm{T} \\ \mathcal{A} \rhd \{\ \tau(\text{this}) \preceq \mathrm{S} \wedge \mathbf{P}\ \}\ \mathrm{S{:}m}\ \{\ \mathbf{Q}\ \}\end{array}}{\mathcal{A} \rhd \{\ \tau(\text{this}) \preceq \mathrm{S} \wedge \mathbf{P}\ \}\ \mathrm{T{:}m}\ \{\ \mathbf{Q}\ \}}$$

The subtype-rule enables one to prove an annotation for a particular subtype S. To prove the sequent $\mathcal{A} \rhd \{\ \tau(\text{this}) \prec \mathrm{T} \wedge \mathbf{P}\ \}\mathrm{T{:}m}\{\ \mathbf{Q}\ \}$ let us first assume that the given program is not open to further extensions, i.e., all subtypes $\mathrm{S}_1, \ldots, \mathrm{S}_k$ of T are known. Based on a complete axiomatization of the (finite) subtype relation, we can derive $S_0 \prec \mathrm{T} \ \Leftrightarrow\ S_0 \preceq \mathrm{S}_1 \vee \ldots \vee S_0 \preceq \mathrm{S}_k$. Thus, we can prove the sequent by applying the subtype-rule for all subtypes of T and by using the disjunct-rule (see Fig. 1) and strengthening with the above equivalence.

Usually, object-oriented programs are open to extensions; i.e., they are designed to be used as parts of bigger programs containing additional subtypes. Typical examples of such open programs are libraries. Intuitively, open OO-programs are more difficult to verify because extensions can influence the behavior of virtual methods. To handle open programs, the proof obligations for later added subtypes are collected. When a subtype is added, the corresponding obligations have to be shown. A detailed discussion of this topic and a technique how such obligations can be treated as assumptions are given in [PHM98].

Language-Independent Axioms and Rules. Besides the axiomatic semantics, the programming logic for Java contains language-independent axioms and rules to handle assumptions and to establish a connection between the predicate logic of pre- and postconditions and triples of the programming logic (cf. Fig. 1).

4 Towards Formal Soundness Proofs for Complex Programming Logics

The last sections presented two definitions of the semantics of Java-K. The advantage of the operational semantics is that its rules can be used to generate interpreters for validating and testing the language definition (cf. [BCD+89]). The axiomatic definition can be considered as a higher-level semantics and is better suited for verification of program properties. Its soundness should be proved w.r.t. the operational semantics.

Since such soundness proofs can be quite long for full-size programming languages, it is desirable to enable mechanical proof checking (cf. [vON98] for the corresponding argumentation about type safety proofs). That is why we built on the techniques developed by Gordon in [Gor89]: Both semantics are embedded into a higher-order logic in which the axioms and rules of the axiomatic semantics are derived from those of the operational semantics. The application of Gordon's technique to Java-K made extensions necessary: a systematic treatment of SOS rules, and handling of virtual methods and recursion.

while-rule:

$$\frac{\mathcal{A} \triangleright \{\ \mathrm{e} = b(\mathit{true}) \wedge \mathbf{P}\ \}\ \mathrm{stm}\ \{\ \mathbf{P}\ \}}{\mathcal{A} \triangleright \{\ \mathbf{P}\ \}\ \mathrm{while\ (e)}\ \{\ \mathrm{stm}\ \}\ \ \{\ \mathrm{e} = b(\mathit{false}) \wedge \mathbf{P}\ \}}$$

if-rule:

$$\frac{\begin{array}{l}\mathcal{A} \triangleright \{\ \mathrm{e} = b(\mathit{true}) \wedge \mathbf{P}\ \}\ \mathrm{stm1}\ \{\ \mathbf{Q}\ \} \\ \mathcal{A} \triangleright \{\ \mathrm{e} = b(\mathit{false}) \wedge \mathbf{P}\ \}\ \mathrm{stm2}\ \{\ \mathbf{Q}\ \}\end{array}}{\mathcal{A} \triangleright \{\ \mathbf{P}\ \}\ \mathrm{if\ (e)}\ \{\ \mathrm{stm1}\ \}\ \mathrm{else}\ \{\ \mathrm{stm2}\ \}\ \ \{\ \mathbf{Q}\ \}}$$

seq-rule:

$$\frac{\begin{array}{l}\mathcal{A} \triangleright \{\ \mathbf{P}\ \}\ \mathrm{stm1}\ \{\ \mathbf{Q}\ \} \\ \mathcal{A} \triangleright \{\ \mathbf{Q}\ \}\ \mathrm{stm2}\ \{\ \mathbf{R}\ \}\end{array}}{\mathcal{A} \triangleright \{\ \mathbf{P}\ \}\ \mathrm{stm1\ stm2}\ \{\ \mathbf{R}\ \}}$$

call-var-rule:

$$\frac{\mathcal{A} \triangleright \{\ \mathbf{P}\ \}\qquad \mathrm{x{=}y.T@m(e);}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \triangleright \{\ \mathbf{P}[\mathrm{w}/Z]\ \}\ \ \mathrm{x{=}y.T@m(e);}\ \{\ \mathbf{Q}[\mathrm{w}/Z]\ \}}$$

where x and w are distinct program variables and Z is an arbitrary logical variable.

static-invoc-var-rule:

$$\frac{\mathcal{A} \triangleright \{\ \mathbf{P}\ \}\qquad \mathrm{x{=}T.m(e);}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \triangleright \{\ \mathbf{P}[\mathrm{w}/Z]\ \}\ \ \mathrm{x{=}T.m(e);}\ \{\ \mathbf{Q}[\mathrm{w}/Z]\ \}}$$

where x and w are distinct program variables and Z is an arbitrary logical variable.

false-axiom:

$$\triangleright \{\ \mathrm{FALSE}\ \}\ \mathrm{comp}\ \{\ \mathrm{FALSE}\ \}$$

assumpt-axiom:

$$\mathbf{A} \triangleright \mathbf{A}$$

assumpt-intro-rule:

$$\frac{\mathcal{A} \triangleright \mathbf{A}}{\mathbf{A}_0\ ,\ \mathcal{A} \triangleright \mathbf{A}}$$

assumpt-elim-rule:

$$\frac{\begin{array}{l}\mathcal{A} \triangleright \mathbf{A}_0 \\ \mathbf{A}_0\ ,\ \mathcal{A} \triangleright \mathbf{A}\end{array}}{\mathcal{A} \triangleright \mathbf{A}}$$

conjunct-rule:

$$\frac{\begin{array}{l}\mathcal{A} \triangleright \{\ \mathbf{P}_1\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}_1\ \} \\ \mathcal{A} \triangleright \{\ \mathbf{P}_2\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}_2\ \}\end{array}}{\mathcal{A} \triangleright \{\ \mathbf{P}_1 \wedge \mathbf{P}_2\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}_1 \wedge \mathbf{Q}_2\ \}}$$

disjunct-rule:

$$\frac{\begin{array}{l}\mathcal{A} \triangleright \{\ \mathbf{P}_1\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}_1\ \} \\ \mathcal{A} \triangleright \{\ \mathbf{P}_2\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}_2\ \}\end{array}}{\mathcal{A} \triangleright \{\ \mathbf{P}_1 \vee \mathbf{P}_2\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}_1 \vee \mathbf{Q}_2\ \}}$$

strength-rule:

$$\frac{\begin{array}{l}\mathbf{P}' \ \Rightarrow\ \mathbf{P} \\ \mathcal{A} \triangleright \{\ \mathbf{P}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}\ \}\end{array}}{\mathcal{A} \triangleright \{\ \mathbf{P}'\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}\ \}}$$

weak-rule:

$$\frac{\begin{array}{l}\mathcal{A} \triangleright \{\ \mathbf{P}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}\ \} \\ \mathbf{Q} \ \Rightarrow\ \mathbf{Q}'\end{array}}{\mathcal{A} \triangleright \{\ \mathbf{P}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}'\ \}}$$

inv-rule:

$$\frac{\mathcal{A} \triangleright \{\ \mathbf{P}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \triangleright \{\ \mathbf{P} \wedge \mathbf{R}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q} \wedge \mathbf{R}\ \}}$$

where $\mathbf{R}$ is a Σ-formula, i.e. doesn't contain program variables or \$.

subst-rule:

$$\frac{\mathcal{A} \triangleright \{\ \mathbf{P}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \triangleright \{\ \mathbf{P}[t/Z]\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}[t/Z]\ \}}$$

where Z is an arbitrary logical variable and t a Σ-term.

all-rule:

$$\frac{\mathcal{A} \triangleright \{\ \mathbf{P}[Y/Z]\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \triangleright \{\ \mathbf{P}[Y/Z]\ \}\ \mathrm{comp}\ \{\ \forall Z : \mathbf{Q}\ \}}$$

where Z, Y are arbitrary, but distinct logical variables.

ex-rule:

$$\frac{\mathcal{A} \triangleright \{\ \mathbf{P}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}[Y/Z]\ \}}{\mathcal{A} \triangleright \{\ \exists Z : \mathbf{P}\ \}\ \mathrm{comp}\ \{\ \mathbf{Q}[Y/Z]\ \}}$$

where Z, Y are arbitrary, but distinct logical variables.

Fig. 1. Additional axioms and rules

This section outlines the translation of SOS rules into higher-order formulas; it embeds the programming logic of Java-K into higher-order logic and relates it to the operational semantics. Furthermore, it presents the soundness proof for the most interesting rules of the programming logic.

SOS rules in HOL. The SOS rules can be directly translated into a recursive predicate definition of the form:

$$\mathit{sem}(S_i, \mathrm{stm}, S_t) \;\Leftrightarrow_{\mathit{def}}\; \bigvee_{R \in \mathit{SOS\text{-}rules}} (\mathrm{stm\ matches}\ \mathit{stmpattern}(R) \;\wedge\; \mathit{antecedents}(R)\ )$$

where $\mathit{stmpattern}(R)$ is the statement pattern occurring in the succedent of R and $\mathit{antecedents}(R)$ denotes the antecedents of the rule where free occurrences of logical variables are existentially bound. E.g., the SOS rule for virtual method invocation is transformed to:

$$\begin{array}{l}
\mathrm{stm\ matches\ (x = y.T{:}m(e);)} \;\wedge\; \exists S' :\; S_i(\mathrm{y}) \neq \mathit{null} \;\wedge\; \tau(S_i(\mathrm{y})) \preceq \mathrm{T} \\
\;\wedge\; \mathit{sem}(\mathit{initS}[\mathrm{this} := S_i(\mathrm{y}), \mathrm{p} := \epsilon(S_i, \mathrm{e}), \$:= S_i(\$)],\; \mathit{body}(\mathit{impl}(\tau(S_i(\mathrm{y})), \mathrm{m})),\; S') \\
\;\wedge\; S_t = S_i[\mathrm{x} := S'(\mathrm{result}),\; \$:= S'(\$)]
\end{array}$$

The semantics of Java-K is given by the least fixpoint of the defining equivalence for *sem*. To simplify inductive proofs and the embedding of sequents into the semantics framework, we introduce an auxiliary semantics predicate *nsem* with an additional parameter of sort *Nat*:

$$\mathit{nsem}(N, S_i, \mathrm{stm}, S_t) \;\Leftrightarrow_{\mathit{def}}\; \bigvee_{R \in \mathit{SOS\text{-}rules}} (\mathrm{stm\ matches}\ \mathit{stmpattern}(R) \;\wedge\; \mathit{antecedents}_{\mathit{nsem}}(R)\ )$$

where $\mathit{antecedents}_{\mathit{nsem}}(R)$ is obtained from $\mathit{antecedents}(R)$ by substituting all occurrences of $\mathit{sem}(S, \mathrm{stm}, S')$ by $N > 0 \wedge \mathit{nsem}(N-1, S, \mathrm{stm}, S')$. It is easy to show that *nsem* is monotonous w.r.t. N, i.e., $\mathit{nsem}(N, S_i, \mathrm{stm}, S_t) \Rightarrow \mathit{nsem}(N + 1, S_i, \mathrm{stm}, S_t)$. The following lemma relates *sem* and *nsem*:

$$\mathit{sem}(S_i, \mathrm{stm}, S_t) \;\Leftrightarrow\; \exists\, N : \mathit{nsem}(N, S_i, \mathrm{stm}, S_t)$$

Semantics for Triples and Sequents. To embed triples into HOL, we consider the pre- and postconditions as predicates on states, i.e., as functions from *State* to *Boolean*. A triple of the form $\{\ \mathbf{P}\ \}$ comp $\{\ \mathbf{Q}\ \}$ is viewed as an abbreviation for $H(\lambda S.\mathbf{P}^*,\ \mathrm{comp}\ , \lambda S.\mathbf{Q}^*)$ where $\mathbf{P}^*$ and $\mathbf{Q}^*$ are obtained from $\mathbf{P}$ and $\mathbf{Q}$ by substituting all occurrences of program variables v, parameters p, and the constant symbol \$ by $S(\mathrm{v})$, $S(\mathrm{p})$, and $S(\$)$ (for simplicity, we assume here that S does not occur in $\mathbf{P}$ or $\mathbf{Q}$). Based on this syntactical embedding, we can define the semantics of triples in terms of *sem*:

$$\begin{array}{lcl}
H(P, \mathrm{stm}, Q) & \Leftrightarrow & \forall S, S' : P(S) \wedge \mathit{sem}(S, \mathrm{stm}, S') \Rightarrow Q(S') \\
H(P, \mathrm{T@m}, Q) & \Leftrightarrow & H(\ \lambda S.\, S(\mathrm{this}) \neq \mathit{null} \wedge P(S),\ \mathit{body}(\mathrm{T@m}),\ Q) \\
H(P, \mathrm{T_0{:}m}, Q) & \Leftrightarrow & \bigwedge_{\mathit{class}\,(\mathrm{T}), \mathrm{T} \preceq \mathrm{T_0}} H(\ \lambda S.\, \tau(S(\mathrm{this})) = \mathrm{T} \wedge P(S),\ \mathit{impl}(\mathrm{T}, \mathrm{m}),\ Q)
\end{array}$$

The first equivalence formulates the usual meaning of Hoare triples for statements (cf. [Gor89]). The second defines implementation annotations in terms of the method body. The third expresses the concept of virtual methods: A virtual method abstracts the properties of all corresponding implementations. The conjunct ranges over all class types T that are subtypes of $\mathrm{T_0}$.

The most interesting aspect of the embedding is the treatment of sequents and rules. Sequents cannot be directly translated into implications with assumptions as premises and consequents as conclusions. For the implementation-rule, this translation would lead to the following **incorrect** rule:

$$\frac{\mathcal{A} \wedge H(P, \mathrm{T@m}, Q) \;\Rightarrow\; H(\,\lambda S.\, S(\mathrm{this}) \neq \mathit{null} \wedge P(S),\; \mathit{body}(\mathrm{T@m}),\; Q)}{\mathcal{A} \;\Rightarrow\; H(P, \mathrm{T@m}, Q)}$$

Using the second equivalence, we can show that the antecedent is a tautology. Since $\mathcal{A}$ can be empty, the rule would allow one to prove that implementations satisfy arbitrary properties. The implementation-rule implicitly contains an inductive argument that has to be made explicit in the embedding. This is done using a predicate K that is related to *nsem* just as H is related to *sem*:

$$\begin{array}{ll}
K(N, P, \mathrm{stm}, Q) & \Leftrightarrow\; \forall S, S' : P(S) \wedge \mathit{nsem}(N, S, \mathrm{stm}, S') \Rightarrow Q(S') \\
K(0, P, \mathrm{T@m}, Q) & \Leftrightarrow\; \mathit{true} \\
K(N+1, P, \mathrm{T@m}, Q) & \Leftrightarrow\; K(N,\; \lambda S.\, S(\mathrm{this}) \neq \mathit{null} \wedge P(S),\; \mathit{body}(\mathrm{T@m}),\; Q) \\
K(N, P, \mathrm{T_0{:}m}, Q) & \Leftrightarrow\; \bigwedge_{\substack{\mathit{class}(\mathrm{T}),\\ \mathrm{T} \preceq \mathrm{T_0}}} K(N,\; \lambda S. \tau(S(\mathrm{this})) = \mathrm{T} \wedge P(S),\; \mathit{impl}(\mathrm{T}, \mathrm{m}),\; Q)
\end{array}$$

Using the lemma that relates *sem* and *nsem*, it is easy to show that

$$H(P,\ \mathrm{comp}\ , Q) \;\Leftrightarrow\; \forall\, N : K(N, P,\ \mathrm{comp}\ , Q)$$

Based on K, sequents can be directly embedded into HOL. A sequent of the form $\{\mathbf{P}_1\}\mathrm{m}_1\{\mathbf{Q}_1\}, ..., \{\mathbf{P}_l\}\mathrm{m}_l\{\mathbf{Q}_l\} \triangleright \{\mathbf{P}\}$ comp $\{\mathbf{Q}\}$ is considered as abbreviation for

$$\forall N : (\; K(N, P_1, \mathrm{m}_1, Q_1) \wedge \ldots \wedge K(N, P_l, \mathrm{m}_l, Q_l) \Rightarrow K(N, P,\ \mathrm{comp}\ , Q)\;)$$

Because of the relation between H and K, a sequent without assumptions is equivalent to the semantics of triples described by H. The complexity of the embedding is a strong argument for using Hoare rules in practical verification instead of the axiomatization of the operational semantics. Many of the proof steps encapsulated in the soundness proof have to be done again and again when verification is directly based on the rules of the operational semantics.

Soundness of the Programming Logic. In the last paragraph, we formalized the semantics of triples and sequents in terms of *sem* and *nsem*. Having a semantics for the sequents, we can prove the soundness of the Java-K logic. The embedding into HOL was chosen in such a way that the soundness proof can be done separately for each logical rule. We illustrate the needed proof techniques by showing the soundness of the implementation- and the invocation-rule. In the proofs, we abbreviate $S(\mathrm{x}) \neq \mathit{null}$ by $\nu(\mathrm{x})$.

implementation-rule. The soundness proof of the implementation-rule illustrates the implicit inductive argument of that rule and demonstrates the treatment of assumptions. We show:

$$\begin{array}{l}
\big(\forall M : A(M) \;\wedge\; K(M, P, \mathrm{T@m}, Q) \Rightarrow K(M, \lambda S. \nu(\mathrm{this}) \wedge P(S), \mathit{body}(\mathrm{T@m}), Q)\big) \\
\Rightarrow\; \forall N : A(N) \;\Rightarrow\; K(N, P, \mathrm{T@m}, Q)
\end{array}$$

where $A(L)$ denotes the conjunction of the embedded assumptions and P and Q abbreviate $\lambda S.\mathbf{P}^*$ and $\lambda S.\mathbf{Q}^*$, respectively. The proof runs by induction on N:

Induction base for $N = 0$: $K(0, P, \mathrm{T@m}, Q)$ is true by definition of K.

Induction step: Assuming that the hypothesis holds for N.

$\big(\forall M : A(M) \;\wedge\; K(M, P, \mathrm{T@m}, Q) \Rightarrow K(M, \lambda S.\nu(\mathrm{this}) \wedge P(S), \mathit{body}(\mathrm{T@m}), Q)\big)$
$\Rightarrow$ [Conjoining the induction hypothesis]
$\big(\forall M : A(M) \;\wedge\; K(M, P, \mathrm{T@m}, Q) \Rightarrow K(M, \lambda S.\nu(\mathrm{this}) \wedge P(S), \mathit{body}(\mathrm{T@m}), Q)\big)$
$\wedge\; \big(A(N) \;\Rightarrow\; K(N, P, \mathrm{T@m}, Q)\big)$
$\Rightarrow$ [Instantiate M by N & propositional logic]
$A(N) \;\Rightarrow\; K(N, \lambda S.\nu(\mathrm{this}) \wedge P(S), \mathit{body}(\mathrm{T@m}), Q)$
$\Rightarrow$ [Definition of K]
$A(N) \;\Rightarrow\; K(N + 1, P, \mathrm{T@m}, Q)$
$\Rightarrow$ [$A(N + 1) \;\Rightarrow\; A(N)$, see below]
$A(N + 1) \;\Rightarrow\; K(N + 1, P, \mathrm{T@m}, Q)$

The implication $A(N + 1) \;\Rightarrow\; A(N)$ follows from the definition of K and the monotonicity of *nsem*.

invocation-rule. The soundness proof of the invocation-rule demonstrates how substitution is handled and why the restrictions on the signatures of pre- and postcondition formulas are necessary. We simplify the proof a bit by leaving out the assumptions in the antecedent and succedent of the rule. The extension to the complete proof is straightforward. Thus, we have to show:

$\forall M : K(M, \lambda S.\mathbf{P}^*, \mathrm{T{:}m}, \lambda S.\mathbf{Q}^*)$
$\Rightarrow\; \forall N : K(N, \lambda S.\nu(\mathrm{y}) \wedge (\mathbf{P}[\mathrm{y/this, e/p}])^*, \mathrm{x{=}y.T{:}m(e);}, \lambda S.(\mathbf{Q}[\mathrm{x/result}])^*)$

Assuming the premise, we prove the conclusion, i.e., for arbitrary N, S, S'':

$\nu(\mathrm{y}) \wedge (\mathbf{P}[\mathrm{y/this, e/p}])^* \wedge \mathit{nsem}(N, S, \mathrm{x{=}y.T{:}m(e);}, S'') \Rightarrow (\lambda S.(\mathbf{Q}[\mathrm{x/result}])^*)(S'')$

This is proved by case distinction on N:

Case $N = 0$: From the definition of *nsem* we get that the premise is false.

Case $N > 0$: The following lemma relates substitution and state update:

$(\mathbf{P}[t_1/x_1, \ldots, t_n/x_n])^* \;=\; (\lambda S.\mathbf{P}^*)(S[x_1 := \epsilon(S, t_1), \ldots, x_n := \epsilon(S, t_n)])$

By this lemma and with P for $\lambda S.\mathbf{P}^*$ and Q for $\lambda S.\mathbf{Q}^*$, the proof goal becomes:

$\nu(\mathrm{y}) \wedge P(S[\mathrm{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S, \mathrm{e})]) \wedge \mathit{nsem}(N, S, \mathrm{x{=}y.T{:}m(e);}, S'')$
$\Rightarrow Q(S''[\mathrm{result} := S''(\mathrm{x})])$

To show this, we will use the following implication (+):

$\nu(\mathrm{y}) \wedge P(\sigma) \wedge \tau(S(\mathrm{y})) \preceq \mathrm{T} \wedge \mathit{nsem}(N - 1, \sigma, \mathit{body}(\mathit{impl}(\tau(S(\mathrm{y})), \mathrm{m})), S') \;\;\Rightarrow\;\; Q(S')$

where σ abbreviates $\mathit{initS}[\mathrm{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S, \mathrm{e}), \$:= S(\$)]$. The proof of (+) uses the general proof assumption $\forall M : K(M, P, \mathrm{T{:}m}, Q)$, the definition of K, and the fact that $\tau(S(\mathrm{y}))$ is a class type. $\tau(S(\mathrm{y}))$ is a class type, because the context conditions of Java/Java-K imply that y is of a reference type and because Java and thus Java-K are type-safe. In addition to (+), we need the fact that for any state S_0 the value of $P(S_0)$ only depends on $S_0(\mathrm{this})$, $S_0(\mathrm{p})$, and $S_0(\$)$, because other variables are not allowed within P (cf. Sect. 3), i.e.,

$S_0(\mathrm{this}) = S_1(\mathrm{this}) \wedge S_0(\mathrm{p}) = S_1(\mathrm{p}) \wedge S_0(\$) = S_1(\$) \;\Rightarrow\; P(S_0) = P(S_1)$

Similarly, Q only depends on $S_0(\mathrm{result})$ and $S_0(\$)$. By this, the remaining goal can be proved as follows:

$$
\begin{array}{l}
\quad \nu(\mathrm{y}) \wedge P(S[\mathrm{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S, \mathrm{e})]) \wedge \mathit{nsem}(N, S, \mathrm{x{=}y.T{:}m(e);}, S'') \\
\Rightarrow\ [\text{Definition of } \mathit{nsem} \text{ (disjunct for ``x=y.T:m(e);''); cf. paragraph ``SOS in HOL''}] \\
\quad \nu(\mathrm{y}) \wedge P(S[\mathrm{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S, \mathrm{e})]) \wedge\ N > 0\ \wedge\ \exists S' : \nu(\mathrm{y}) \wedge \tau(S(\mathrm{y})) \preceq \mathrm{T} \\
\quad \wedge\ \mathit{nsem}(N-1, \sigma, \mathit{body}(\mathit{impl}(\tau(S(\mathrm{y})), \mathrm{m})), S')\ \wedge\ S'' = S[\mathrm{x} := S'(\mathrm{result}), \$:= S'(\$)] \\
\Rightarrow\ [\text{case assumption } N > 0\text{; general logic}] \\
\quad \exists S' : S'' = S[\mathrm{x} := S'(\mathrm{result}), \$:= S'(\$)] \\
\quad \wedge\, \nu(\mathrm{y}) \wedge P(S[\mathrm{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S, \mathrm{e})]) \wedge \tau(S(\mathrm{y})) \preceq \mathrm{T} \\
\quad \wedge\ \mathit{nsem}(N-1, \sigma, \mathit{body}(\mathit{impl}(\tau(S(\mathrm{y})), \mathrm{m})), S') \\
\Rightarrow\ [\ P(S[\mathrm{this} := S(\mathrm{y}), \mathrm{p} := \epsilon(S, \mathrm{e})]) = P(\sigma)\text{; lemma } (+)] \\
\quad \exists S' : S'' = S[\mathrm{x} := S'(\mathrm{result}), \$:= S'(\$)]\ \wedge\ Q(S') \\
\Rightarrow\ [\ S'(\mathrm{result}) = S''(\mathrm{x}) = S''[\mathrm{result} := S''(\mathrm{x})](\mathrm{result}),\ S'(\$) = S''[\mathrm{result} := S''(\mathrm{x})](\$)] \\
\quad \exists S' : Q(S''[\mathrm{result} := S''(\mathrm{x})]) \\
\Rightarrow \\
\quad Q(S''[\mathrm{result} := S''(\mathrm{x})])
\end{array}
$$

5 Conclusions

We introduced the sequential Java subset Java-K, which provides the typical OO-language features such as classes and interfaces, subtyping, inheritance, dynamic dispatch, and encapsulation. Based on a formalization of object stores as first-order values, we presented a Hoare-style programming logic for Java-K. A central concept of this logic is the notion of virtual methods to handle overriding and dynamic dispatch. Virtual methods represent the common properties of all corresponding subtype methods. We showed how virtual methods and method implementations can be used to cover statically and dynamically bound method invocations, subtyping, and inheritance in programming logics.

The logic has been proved sound w.r.t. an SOS semantics of Java-K. Following the ideas of [Gor89], we embedded both semantics into a higher-order logic and derived the axioms and rules of the programming logic from those of the operational semantics. We presented the proofs for two typical rules. This technique for soundness proofs provides a good basis for applying proof checkers. Mechanical checking of the soundness proof is considered further work.

References

[AL97] M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, 1997.

[BCD$^+$89] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SIGPLAN Notices 24(2), 1989.

[Gor89] M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.

[JvdBH$^+$98] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1998. Also available as TR CSI-R9812, University of Nijmegen.

[Lea96] G. T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In H. Kilov and W. Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996.

[Lei97] K. R. M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In B. Pierce, editor, *Proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages*, 1997. Available from: `www.cs.indiana.edu/hyplan/pierce/fool/`.

[MPH97] P. Müller and A. Poetzsch-Heffter. Formal specification techniques for object-oriented programs. In M. Jarke, K. Pasedach, and K. Pohl, editors, *Informatik 97: Informatik als Innovationsmotor*, Informatik Aktuell. Springer-Verlag, 1997.

[PH97] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, Jan. 1997. `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[PHM98] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods (PROCOMET)*, 1998.

[vON98] D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer, 1998. To appear.

Appendix

These are the SOS rules for static method invocations, constructor calls, sequential statement composition, cast, while, and if statements:

$$\frac{\mathit{initS}[\mathrm{p} := \epsilon(S,\mathrm{e}), \$:= S(\$)] :\ \mathit{body}(\mathrm{T@m})\ \rightarrow S'}{S :\ \mathrm{x{=}T.m(e);}\ \rightarrow S[\mathrm{x} := S'(\mathrm{result}),\ \$:= S'(\$)]}$$

$$\frac{\mathit{true}}{S :\ \mathrm{x{=}new\ T();}\ \rightarrow S[\mathrm{x} := \mathrm{new}(S(\$),\mathrm{T}), \$:= S(\$)\langle\mathrm{T}\rangle]}$$

$$\frac{S :\ \mathrm{stm1}\ \rightarrow S',\ S' :\ \mathrm{stm2}\ \rightarrow S''}{S :\ \mathrm{stm1\ stm2}\ \rightarrow S''}$$

$$\frac{\tau(\epsilon(S,\mathrm{e})) \preceq \mathrm{T}}{S :\ \mathrm{x{=}(T)e;}\ \rightarrow S[\mathrm{x} := \epsilon(S,\mathrm{e})]}$$

$$\frac{\epsilon(S,\mathrm{e}) = b(\mathit{true}),\ S :\ \mathrm{stm}\ \rightarrow S',\ S' :\ \mathrm{while(e)\{stm\}}\ \rightarrow S''}{S :\ \mathrm{while(e)\{stm\}}\ \rightarrow S''}$$

$$\frac{\epsilon(S,\mathrm{e}) = b(\mathit{false})}{S :\ \mathrm{while(e)\{stm\}}\ \rightarrow S}$$

$$\frac{\epsilon(S,\mathrm{e}) = b(\mathit{true}),\ S :\ \mathrm{stm1}\ \rightarrow S'}{S :\ \mathrm{if(e)\{stm1\}\ else\{stm2\}}\ \rightarrow S'}$$

$$\frac{\epsilon(S,\mathrm{e}) = b(\mathit{false}),\ S :\ \mathrm{stm2}\ \rightarrow S'}{S :\ \mathrm{if(e)\{stm1\}\ else\{\ stm2\ \}}\ \rightarrow S'}$$