Logical Foundations for Typed Object-Oriented Languages

Arnd Poetzsch-Heffter and Peter Müller
Fernuniversität Hagen, D-58084 Hagen, Germany
email: [poetzsch,peter.mueller]@fernuni-hagen.de

Abstract

This paper presents logical foundations for the most important object-oriented language features, including abstract types, dynamic binding, subtyping, and inheritance. These foundations are introduced along with an object-oriented kernel language. We show how object environments of such languages can be formalized in an algebraic way. Based on this foundation, we develop a Hoare-style logic for formal verification of object-oriented programs.

Keywords

Programming logic; verification; object-oriented programming language; programming language semantics

1 INTRODUCTION

Typed object-oriented programming languages like C++, Java, Eiffel, Oberon, Modula-3, BETA, and Ada95 form an important practical class of languages. This paper presents the logical foundations needed to formally verify object-oriented programs from that class. This introduction sketches the motivation for this work, the approach taken, and related work.

Motivation The language class mentioned above is increasingly important, in particular, because OO-techniques support reuse and adaptation via inheritance. Reuse and adaptation of software components have been shown to be very useful for application frameworks and component-based architectures. There are three reasons why formal verification techniques are especially interesting for OO-programs: 1. Verification techniques are needed for software certification in the quickly developing software component industry, which is based on OO-technology. 2. The potential for reuse in OO-programming carries over to reusing proofs. 3. *Formal* verification is important because of the complexity of the underlying languages. Without tool support, large program proofs are tedious to handle and can hardly be kept error-free. Formality is a prerequisite for the construction of computer-based tools.

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 body of a procedure.

Our solutions to these problems build on well-known techniques. Object environments are specified as an abstract data type with operations to create objects and to read and write instance variables/attributes. Based on object environments, abstraction of object structures can be expressed. Using such abstractions, the behavior of abstract types can be formulated. Our programming logic refines a Hoare-logic for procedural languages. To express the relation between environments in pre- and poststates the current object environment can be referenced through a special variable.

Object-oriented languages use dynamically bound method invocations where procedural languages use statically bound calls. Consider e.g. the invocation `x.m()` where variable x is of type T with subtypes T1 and T2 and where m belongs to the interface of T (and consequently to the interfaces of T1 and T2). If x holds an object of type T1 at method invocation time, the implementation associated with T1:m is executed; if it holds a T2-object, T2:m is executed. The basic idea for the verification of method invocations is to consider them as statically bound calls to virtual methods; e.g., the above method invocation is considered as a call to the virtual method T:m, the properties of which are used to verify the call. The distinction between the virtual behavior of a method and the behavior of the associated implementation allows the transfer of verification techniques for procedures to OO-programs.

Related Work In [Lei97], a wlp-calculus for an OO-language is presented that is semantically similar to the language considered here. But the underlying methodology is different. Method specifications are part of the programs. The calculus is used to check that programs satisfy their specifications. Changing the specifications modifies the program and makes a new check of the entire program necessary. In our approach, programs and specifications are separated. The programming logic is used to verify theorems about programs. In section 5, we show that the approach in [Lei97] can be considered as restricting our approach to a certain program development strategy. Thereby, it becomes simpler and more appropriate for automatic checking, but gives up flexibility that seems important to us for general program 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 our language aims to be close to practical OO-languages. In particular, our language supports inheritance.

The programming logic presented in the following is essentially an extension of the partial correctness logic described in [Apt81]. The extension to object-orientation profited from other papers about verification of imperative languages with complex data structures, especially [Suz80].

In practice, formal verification requires elaborate formal specification techniques. Our goal is to verify implementations w.r.t. program specifications according to the two-tiered specification approach developed in the Larch project (cf. [GH93]). In [MPH97], we outline the connection between Larch-style program specifications and our logical foundations. To automate proof steps, weakest precondition transformations are very important (cf. [Gri81]). We apply such techniques for the classical rules. An extension to the new rules is considered future work.

Overview Section 2 introduces the simple kernel object-oriented language SKOOL. Section 3 contains a formalization of object environments. The axiomatic semantics of SKOOL is presented in section 4. Section 5 demonstrates how the logic can be used to verify OO-programs. In section 6, SKOOL is extended by inheritance and the logic is adapted to this new feature.

2 AN OBJECT-ORIENTED KERNEL LANGUAGE

The common features of typed OO-programming languages are (a) type/class declarations that combine records and type-local procedures, so-called *methods*, (b) subtyping, and (c) code inheritance. In this paper, we develop a logic for a **k**ernel of **o**bject-**o**riented **l**anguages called KOOL supporting these features. KOOL is presented in two steps. This section presents the sublanguage SKOOL featuring type declarations and subtyping. Section 6 adds inheritance.

Types and Methods A *type* describes a collection of objects together with their operations. In the context of OO-programming, it is helpful to distinguish between *concrete* and *abstract* types: A concrete type is associated with a complete implementation; an abstract type expresses common behavior of its subtypes. In SKOOL, an abstract type declaration has the form:

```
abstract type TA subtype of <AbsTypeList> is <MethSigList> end
```

A method signature consists of the method name, a parameter list, and a result type. If the result of a method is irrelevant, the result type may be omitted. The signature list is called the *interface* of type TA, denoted by IF(TA). The declaration of a concrete type has the form:

```
type TC subtype of <AbsTypeList> is <MethList> <AttrList> end
```

A method is a type-local procedure with an *implicit parameter* having the enclosing type; the implicit parameter is denoted by `this`. A method definition consists of a method signature, a list of local variable declarations, and a body:

$\mathrm{m}($ $\mathrm{p}_1{:}\mathrm{T}_{\mathrm{p}_1}, .., \mathrm{p}_q{:}\mathrm{T}_{\mathrm{p}_q}$ $){:}\mathrm{T}_{\mathrm{m}}$ `is` $\mathrm{v}_1{:}\mathrm{T}_{\mathrm{v}_1}; ..; \mathrm{v}_r{:}\mathrm{T}_{\mathrm{v}_r};$ `<Body>` `end`

The list of local variables implicitly contains a variable `result` of type T_m. The value of result is returned as result of `m`.The body of a method is either *predefined* or a (usually compound) statement.

Attribute declarations have the form `attr a:TR` where `a` is the attribute name and `TR` is a type name, the *range type* of `a`. A declaration `attr a:TR` is considered as an abbreviation for the following two method declarations:

```
get_a(): TR  is  predefined        set_a( p:TR ) is  predefined
```

In addition to this, a concrete type may contain the predefined method `equ` comparing objects of that type with other objects, and `new` creating objects of that type. Method `new` is a typical case of a static or class method, i.e., a method that does not need an implicit parameter. We avoid a special treatment of such methods here to keep the number of logical rules in later sections small. Such methods can be called using the null-object that is predefined for each concrete type (see below).

The *interface of a concrete type* consists of the signatures of the declared methods and of the attribute access methods. All method names in interfaces must be distinct.

A *SKOOL program* is a finite set of type declarations containing an abstract type OBJECT with equality `equ(p:OBJECT):BOOL` and the concrete types INT and BOOL with appropriate operations. The `subtype`-clause defines a binary relation on the types of a program. The reflexive and transitive closure of this relation is called the *subtype relation* (denoted by $\preceq$). In SKOOL, three conditions have to be satisfied: 1. The subtype relation has to be a partial ordering. 2. Concrete types have to be minimal in the subtype relation. 3. Every type has to be a subtype of OBJECT (for INT and BOOL this property is only claimed to avoid separate treatment of these types).

A method signature with (explicit) parameter types $S_1, \ldots, S_z$ and result type SR is called a *subsignature* of a signature with (explicit) parameter types $T_1, \ldots, T_z$ and result type TR, if they have the same method name and $T_j \preceq S_j$ and SR $\preceq$ TR. An interface IF(S) is called *subinterface* of interface IF(T) if for each signature in IF(T) there is a subsignature in IF(S). In SKOOL programs, IF(S) has to be a subinterface of IF(T) if S $\preceq$ T.

Expressions and Statements SKOOL supports only atomic expressions. An expression is either an INT or BOOL constant, a variable or parameter of the enclosing method, or a null-object: For each user-declared concrete type TC there is exactly one null-object denoted by null(TC). Null-objects have no attributes; they are needed for initialization of attributes and to handle recursive types. Typed null-objects can also be used to simulate static methods by using the null-object of the corresponding type as implicit argument in the invocation. For brevity, compound expressions are not considered here. They can be broken up into atomic expressions by introducing auxiliary variables for subexpressions. SKOOL provides if- and while-statements and sequential

statement composition with the usual syntax. Assignments with casting and method invocations are written as follows:

`v := (T) EXP` v := EXP_0 . T:m(EXP_1, ...,EXP_q)

The type T in an assignment must be a subtype of both the left-hand side's and the expression's type. If T equals the expression type, it can be omitted. Casts are provided by KOOL because they occur in most OO-languages. An invocation statement is type-correct, if the type of EXP_0 equals type T and the interface of T contains a method m such that the types of the EXP_i are subtypes of the parameter types, and the result type of T:m is a subtype of the type of the left-hand side. We use the redundant type prefix "T:" for methods to simplify the verification. If it is clear from the context, it may be omitted. To come closer to notational conventions, we write `v:= EXP.a` instead of `v:= EXP.T:get_a()`, and `v.a:= EXP` instead of `v.T:set_a(EXP)`.

3 FORMALIZING OBJECT ENVIRONMENTS

An object environment describes the states of all objects in a program at a certain point of execution. In particular, it describes how objects are linked via references and which objects are alive. A formalization of object environments is important as the semantic foundation of program specifications and is central for the verification of OO-programs. This section summarizes the formal background used and formalizes objects, object states, and object environments.

Formal Background The general techniques underlying our work can be formulated in different formal frameworks. In this paper, we use many-sorted first-order specifications (cf. e.g. [Wir90]) and recursive data type specifications (see below). A *(many-sorted) signature* Σ is a tuple $\langle S, F\rangle$ where S is a set of *sort symbols* and F is a set of *function symbols*. We assume that all many-sorted signatures contain a sort *Bool* with constants TRUE and FALSE as well as an appropriate integer data type with sort *Int*.

The set of Σ-*formulas* contains (1) every term of sort *Bool*, (2) $\neg G, (G \wedge H), (G \vee H), (G \Rightarrow H)$, and $(G \Leftrightarrow H)$, and (3) $(\forall X : G)$ and $(\exists X : G)$, where G and H are Σ-formulas, and X is a logical variable. Substitution of all free occurrences of a variable or constant X by a term t in formula $\mathbf{P}$ is denoted by $\mathbf{P}[t/X]$, where the sort of t has to be equal to the sort of X.

For the specification of recursive data types, we use the following notation:

data type
$DSrt = constr_1(USrt_1^1, \ldots, USrt_1^{m_1}) \mid \ldots \mid constr_n(USrt_n^1, \ldots, USrt_n^{m_n})$
end data type

Such a definition introduces the sort *DSrt* with constructor functions $constr_j$. We assume rules to reason about recursive data types; these rules allow in particular to prove the in-/equality of two terms and support term induction.

Notation We use the following notational conventions: Σ-formulas are denoted by bold capital letters **P**, **Q**, etc. Functions, constants, and program variables are written in lower case. Logical variables are written in upper case.

Objects The set of objects in a given program depends on the declared types, attributes, and their relation. The simplest approach is to assume that a program is fixed and to define the objects for this program. The disadvantage of this approach becomes apparent when new types are added. By such program extensions, the set of objects is enlarged. Thus, formulas quantifying over all objects that are valid in the original program may become invalid in the extended program. To avoid this, we consider a program to be part of all its extensions. I.e., we assume infinite sorts *TypId* and *ATypId* of type identifiers for concrete and abstract types resp., and an infinite sort *AttId* of attribute identifiers where attributes with the same name, but different object types, are considered to be different. A program determines the relation between its types and attributes, but leaves the relation between types and attributes not occurring in the program un(der)specified. Extending a program then means refining the type-attribute relation. To distinguish different objects of the same type, we assume an infinite sort *ObjId* of object identifiers:

data type
Type = *BOOL*
| *INT*
| *ct*(*TypId*)
| *at*(*ATypId*)
end data type

data type
Object = *bool*(*Bool*)
| *int*(*Int*)
| *null*(*TypId*)
| *mkobj*(*TypId*, *ObjId*)
end data type

As can be seen from the definition of data type *Object*, an object of a declared type T is either the null-object of T or a typed object identifier. Notice that all objects are of concrete types. Furthermore, we assume an appropriate axiomatization of the subtype relation $\preceq$ on sort *Type*.

Object States Object states are modeled via *object locations*: For each attribute of its type, an object has a location. Locations can be considered as anonymous variables, i.e., variables that can only be referenced through the object they belong to. Locations are often called instance variables.

data type
Location = *mkloc*(*AttId*, *ObjId*)
end data type

The relations between objects, types, and attributes, and their basic properties are expressed by the following functions: *typ* yields the type of an object; *isnull* asserts that an object is a null-object; *otyp* and *rtyp* yield the object and range type of an attribute; *ltyp* yields the type of a location; *obj* yields the object a location belongs to; given an object and an attribute, *loc* yields the

corresponding location; *static* asserts that an object is not subject to object creation.

$$
\begin{array}{ll}
typ : Object \to Type & isnull : Object \to Bool \\
\quad typ(bool(B)) = BOOL & \quad \neg isnull(bool(B)) \\
\quad typ(int(I)) = INT & \quad \neg isnull(int(I)) \\
\quad typ(null(T)) = ct(T) & \quad \neg isnull(mkobj(T, OI)) \\
\quad typ(mkobj(T, OI)) = ct(T) & \quad isnull(null(T)) \\
\\
ltyp : Location \to Type & otyp : AttId \to TypId \\
\quad ltyp(mkloc(A, OI)) = rtyp(A) & rtyp : AttId \to Type \\
\\
obj : Location \to Object & init : Type \to Object \\
\quad obj(mkloc(A, OI)) = mkobj(otyp(A), OI) & \quad init(BOOL) = false \\
 & \quad init(INT) = int(0) \\
loc : Object \times AttId \to Location & \quad init(ct(T)) = null(T) \\
\quad loc(mkobj(otyp(A), OI), A) = mkloc(A, OI) & \quad typ(init(at(T))) \preceq at(T) \\
 & \quad static(init(at(T))) \\
static : Object \to Bool & \\
\quad static(X) \Leftrightarrow typ(X) = BOOL \vee typ(X) = INT \vee isnull(X) &
\end{array}
$$

All functions except *otyp* and *rtyp*, are program-independent. The object and range types of attributes do not change if a program is extended. We assume that *otyp* and *rtyp* are specified by enumeration for all attributes of a given program.

Object Environments Object environments are modeled by an abstract data type with main sort *ObjEnv* and the following operations: $E\langle L := X\rangle$ denotes updating the object environment E at location L with object X. $E(L)$ denotes reading location L in environment E; $E(L)$ is called the *object held by L in E*. $new(E, T)$ returns a new object of type T in environment E. $E\langle T\rangle$ denotes the environment after allocating a new object of type T in E. $alive(X, E)$ yields true if and only if object X has been allocated in E:

$$
\begin{array}{ll}
\_\langle\_ := \_\rangle & : ObjEnv \times Location \times Object \to ObjEnv \\
\_\langle\_\rangle & : ObjEnv \times TypId \to ObjEnv \\
\_(\_) & : ObjEnv \times Location \to Object \\
alive & : Object \times ObjEnv \to Bool \\
new & : ObjEnv \times TypId \to Object
\end{array}
$$

In the following, we present and explain the axiomatization of these functions. Location update and object allocation construct new environments from given ones; location read and liveness test allow the observation of environments. We first consider the properties of environments observable by location reads, then the liveness properties, and finally the properties of the *new*-operation.

Axiom env1 states that updating one location does not affect the objects held by other locations. Axiom env2 states that reading a location updated by an object X yields X, if the object of the location and X are both alive. We restrict this property to living objects in order to guarantee that loca-

tions never hold non-living objects and that locations of non-living objects are initialized as described by axiom env3. Axiom env4 states that updates by non-living objects do not modify the environment. The assumptions and requirements about the liveness of objects in axioms env2, env3, env4 simplify the definition of equivalence properties on environments. Axiom env5 states that allocation does not affect the objects held by locations:

$$\begin{array}{ll}
\text{env1}: & L1 \neq L2 \;\Rightarrow\; E\langle L1 := X\rangle(L2) = E(L2) \\
\text{env2}: & alive(obj(L), E) \wedge alive(X, E) \;\Rightarrow\; E\langle L := X\rangle(L) = X \\
\text{env3}: & \neg alive(obj(L), E) \;\Rightarrow\; E(L) = init(ltyp(L)) \\
\text{env4}: & \neg alive(X, E) \;\Rightarrow\; E\langle L := X\rangle = E \\
\text{env5}: & E\langle T\rangle(L) = E(L)
\end{array}$$

Axiom env6 states that location updates do not influence liveness of objects. Axiom env7 specifies that an object is alive after allocation if and only if it was alive before allocation or it is the newly allocated object. Axiom env8 ensures that objects held by locations are alive. Together with env2, env3, and env4, this simplifies proofs. Finally, static objects, i.e., objects that are not subject to creation, are considered to be alive:

$$\begin{array}{ll}
\text{env6}: & alive(X, E\langle L := Y\rangle) \;\Leftrightarrow\; alive(X, E) \\
\text{env7}: & alive(X, E\langle T\rangle) \;\Leftrightarrow\; alive(X, E) \vee X = new(E, T) \\
\text{env8}: & alive(E(L), E) \\
\text{env9}: & static(X) \;\Rightarrow\; alive(X, E)
\end{array}$$

The following two axioms specify properties of the new-operation. A newly created object is not alive in the environment in which it was created (env10) and it has the correct type (env11).

$$\begin{array}{ll}
\text{env10}: & \neg alive(new(E, T), E) \\
\text{env11}: & typ(new(E, T)) = ct(T)
\end{array}$$

A model for these axioms can be found in [PH97].

4 A LOGIC FOR OBJECT-ORIENTED PROGRAMS

This section presents a programming logic for SKOOL. In particular, we show how method invocation and subtyping can be handled.

Program Specific Signatures To specify program properties, we have to refer to variables, attributes, and types in formulas. This is enabled by introducing constant symbols for these entities. More precisely, let Π be a SKOOL program and let Σ denote a signature that includes the signature of the object environment as introduced above, a constant symbol T of sort *TypId* or *ATypId* for each concrete or abstract type T declared in Π, and a constant symbol T:att of sort *AttId* for each attribute att declared in type T. To refer to the current object environment in formulas, the constant symbol $ of sort

ObjEnv is used, and Γ denotes $\Sigma \cup \{\$\}$. The current object environment \$ can be considered as a global variable.

Furthermore, we treat program variables and parameters syntactically as constant symbols of sort *Object* to simplify quantification and substitution rules*. The following signatures are used to define context conditions for pre- and postconditions (see below). Let m be a method: 1. The extension of Γ by constant symbols for the parameters of m (in particular, *this*) is denoted by $\Gamma_{pre(m)}$. 2. The extension of Γ by constant symbols for each parameter and local variable of m is denoted by $\Gamma_{body(m)}$. 3. The extension of Γ by the constant symbol result is denoted by Γ_{post}.

Triples and Sequents A *program component* is a method signature occurrence or a statement occurrence within a given program. I.e., we assume that the program context in which each program component occurs is given implicitly. In particular, we can refer to the method enclosing a statement. A *Hoare triple* or simply *triple* has the form { **P** } COMP { **Q** } where COMP is a program component and **P** and **Q** are first-order formulas, called *pre-* and *postconditions*, respectively. If the component in a triple **A** is a method, we call **A** a *method annotation*; otherwise **A** is called a *statement annotation*. Pre- and postconditions of statement annotations are formulas over $\Gamma_{body(m)}$ where m is the enclosing method; pre- and postconditions in annotations of method m are $\Gamma_{pre(m)}$-formulas and Γ_{post}-formulas, respectively.

A triple { **P** } COMP { **Q** } specifies the following *refined* partial correctness property: If **P** holds in a state before executing COMP, then execution of COMP either

1. terminates and **Q** holds in the state after execution or
2. aborts because of errors or actions that are beyond the semantics of the programming language (e.g., memory allocation problems, stack overflow, external interrupts from the execution environment), or
3. runs forever.

In particular, execution of COMP does not abort because of dereferencing of null-objects or illegal casts. Thus, this refined partial correctness logic can be used to prove that a program does not produce such runtime errors.

A *sequent* has the form $\mathcal{A} \vdash \mathbf{A}$ where $\mathcal{A}$ is a set of method annotations and **A** is a triple. Triples in $\mathcal{A}$ are called *assumptions* of the sequent and **A** is called the *consequent* of the sequent. A sequent expresses the fact that we can prove a triple based on some assumptions about methods. Sequents are necessary to handle recursive procedures and subtyping (see below).

Axiomatic Semantics The axiomatic semantics of SKOOL consists of axioms for the predefined methods, a cast axiom that adapts the classical as-

*This treatment imitates the distinction between global (i.e. logical) variables and local variables in temporal logic (cf. [GU91], p. 233ff.)

signment axiom, and rules explaining statement and method behavior. We concentrate here on the most interesting axioms and rules.

Predefined Methods In SKOOL, there are predefined methods for equality test, object creation, and attribute access. The specifications of these methods illustrate the use of the object environment in triples. In the axioms for the attribute access methods, we use this.T:a as abbreviation for $loc(\text{this}, \text{T:a})$ and $prec(\text{this}, \text{T})$ as abbreviation for $\neg isnull(\text{this}) \wedge typ(\text{this}) = \text{T}$:

$\vdash \{ \mathbf{P}[bool(\text{this} = \text{p})/\text{result}] \}$	T:equ(p:OBJECT):BOOL	$\{ \mathbf{P} \}$
$\vdash \{ \mathbf{P}[new(\$, \text{T})/\text{result} , \$\langle\text{T}\rangle/\$] \}$	T:new():T	$\{ \mathbf{P} \}$
$\vdash \{ prec(\text{this}, \text{T}) \wedge \mathbf{P}[\$(\text{this.T:a})/\text{result}] \}$	T:get_a():TR	$\{ \mathbf{P} \}$
$\vdash \{ prec(\text{this}, \text{T}) \wedge \mathbf{P}[\$\langle\text{this.T:a} := \text{p}\rangle/\$] \}$	T:set_a(p:TR)	$\{ \mathbf{P} \}$

Statements Assignments in SKOOL may be combined with casts to narrow the static type of the right-hand side. The semantics of a cast is only defined if the (dynamic) type of the object denoted by the right-hand side expression is a subtype of the given type T:

cast-axiom:

$\vdash \{ typ(\text{EXP}) \preceq T \wedge \mathbf{P}[\text{EXP/v}] \}$ v := (T) EXP $\{ \mathbf{P} \}$

The rules for the loop, conditional, and sequential statement are standard and not presented here. Because of the syntactical conditions discussed above, the rule for the invocation statement becomes very intuitive. Formal parameters are substituted by the actual parameter expressions and the result variable is substituted by the left-hand side variable:

invocation-rule:

$$\frac{\mathcal{A} \vdash \{ \mathbf{P} \} \ \text{T:m}(\text{p}_1 : \text{T}_1, \ldots, \text{p}_q : \text{T}_q)\text{:TR} \ \{ \mathbf{Q} \}}{\mathcal{A} \vdash \{ \mathbf{P}[\text{E}_0/\text{this}, \text{E}_1/\text{p}_1, \ldots, \text{E}_q/\text{p}_q] \} \ \text{v} := \text{E}_0 \ . \ \text{T:m}(\text{E}_1, \ldots, \text{E}_q) \ \{ \mathbf{Q}[\text{v/result}] \}}$$

The fact that program variables different from v are not modified by an invocation is expressed by the following rule. Local variables and parameters w of the enclosing method different from v may be substituted for logical variables:

var-rule:

$$\frac{\mathcal{A} \vdash \{ \mathbf{P} \} \ \text{v} := \text{E}_0 \ . \ \text{T:m}(\text{E}_1, \ldots, \text{E}_q) \ \{ \mathbf{Q} \}}{\mathcal{A} \vdash \{ \mathbf{P}[\text{w}/Z] \} \ \text{v} := \text{E}_0 \ . \ \text{T:m}(\text{E}_1, \ldots, \text{E}_q) \ \{ \mathbf{Q}[\text{w}/Z] \}}$$

Methods There are two rules explaining the derivation of method annotations in SKOOL. The first rule deals with methods of concrete types, i.e., with methods having a body. The second rule deals with methods of abstract types, i.e., with *virtual methods*.

Essentially, an annotation of an implemented method m holds if it holds for its body. This basic rule is strengthened in two aspects: 1. 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. 2. The requirement that local variables are initialized to static objects of the correct type is established:

implementation-rule:

$$\frac{\mathcal{A}\,,\ \{\mathbf{P}\}\ \text{T:m(..)}\ \{\mathbf{Q}\}\ \vdash\ \{\mathbf{P} \wedge \bigwedge_i(\ \text{v}_i = \mathit{init}(\text{TV}_i)\ )\}\ \text{BODY(T:m)}\ \{\mathbf{Q}\}}{\mathcal{A} \vdash \{\mathbf{P}\}\ \ \text{T:m(..)}\ \ \{\mathbf{Q}\}}$$

where v_i are the local variables of method T:m and TV_i denotes the static type of v_i.

In OO-languages with subtyping and dynamic binding, annotations for virtual methods must be derived. The basic idea is simple: To prove something for a virtual method T:m, we have to prove it for all the corresponding methods Ti:m in the subtypes. In order to get the type assumptions right, we consider a pseudo-implementation of T:m and derive the subtype-rule using the programming logic without a subtype-rule. The pseudo-code performs a case distinction according to the type of the this-object. Depending on the type, the this-object is cast to one of T's subtypes and the corresponding method associated with the subtype is called. A simple example, in which T has only two subtypes T1 and T2, is illustrative:

```
meth  T:m( p: TP ): TR is
  v1: T1 ;  v2: T2 ;
  if  typ(this) ⪯ T1      then  v1     := (T1) this ;
                                result := v1.T1:m(p)
  else if  typ(this) ⪯ T2 then  v2     := (T2) this ;
                                result := v2.T2:m(p)
                          else  abort          end end
end
```

Applying the programming logic to prove $\{\ \mathbf{R}\ \}$ T:m(p:TP):TR $\{\ \mathbf{Q}\ \}$ leads to the following proof obligations:

$\{\ \tau \preceq \text{T1} \wedge \mathbf{R}\ \}$	T1:m(p:TP1):TR1	$\{\ \mathbf{Q}\ \}$
$\{\ \tau \preceq \text{T2} \wedge \mathbf{R}\ \}$	T2:m(p:TP2):TR2	$\{\ \mathbf{Q}\ \}$
$\{\ \tau \not\preceq \text{T1} \wedge \tau \not\preceq \text{T2} \wedge \mathbf{R}\ \}$	abort	$\{\ \mathbf{Q}\ \}$

where τ abbreviates *typ*(this). The last triple can only be proved for arbitrary $\mathbf{R}$ if its precondition is falsified. If the program is complete, we can use the knowledge that T1 and T2 are the only subtypes of T, i.e. $\tau \preceq \text{T} \ \Rightarrow\ \tau \preceq \text{T1} \vee \tau \preceq \text{T2}$. Substituting $\tau \preceq \text{T} \wedge \mathbf{P}$ for $\mathbf{R}$ and applying the implication, the precondition of the abort statement becomes false so that this triple can be proved (see false-axiom in appendix). Because of $\tau \preceq \text{T}i \ \Rightarrow\ \tau \preceq \text{T}$, this development can be summarized by the following rule:

$$\frac{\begin{array}{lll}\{\ \tau \preceq \text{T1} \wedge \mathbf{P}\ \} & \text{T1:m(p:TP1):TR1} & \{\ \mathbf{Q}\ \} \\ \{\ \tau \preceq \text{T2} \wedge \mathbf{P}\ \} & \text{T2:m(p:TP2):TR2} & \{\ \mathbf{Q}\ \}\end{array}}{\{\ \tau \preceq \text{T} \wedge \mathbf{P}\ \} \quad \text{T:m(p:TP):TR} \quad \{\ \mathbf{Q}\ \}}$$

This rule can be generalized to abstract types with an arbitrary, but known number of subtypes. But what happens, if the program is not complete? Extending a program Π means adding new subtypes to some of Π's abstract types AT. In general, this invalidates the proofs of AT's methods, since these proofs were based on the assumption that all subtypes of AT were present. To handle program extensions, we collect proof obligations that have to be met by all future subtypes. To cover these obligations in the logic, we weaken the above rule by adding an assumption to the conclusion. As an aside, the use of assumptions allows us to avoid an arbitrary number of antecedents in the subtype-rule. In summary, we get the following compact form for the subtype-rule:

subtype-rule:

$$\frac{\mathrm{T}' \preceq \mathrm{T} \qquad \mathcal{A} \vdash \{\ \tau \preceq \mathrm{T}' \wedge \mathbf{P}\ \}\ \mathrm{T}'{:}\mathrm{m}\ \{\ \mathbf{Q}\ \}}{\{\ \tau \preceq \mathrm{T} \wedge \tau \not\preceq \mathrm{T}' \wedge \mathbf{P}\ \}\ \mathrm{T}{:}\mathrm{m}\ \{\ \mathbf{Q}\ \}\ ,\ \mathcal{A}\ \vdash \{\ \tau \preceq \mathrm{T} \wedge \mathbf{P}\ \}\ \mathrm{T}{:}\mathrm{m}\ \{\ \mathbf{Q}\ \}}$$

The elimination of the assumptions and the application of the rules for program verification is treated in section 5.

Programming Logic The programming logic for SKOOL is the union of axioms and rules described above and of the language-independent rules given in the appendix. The language-independent rules are essentially an adaptation of the proof systems G and G_0 presented in [Apt81].

5 VERIFICATION OF OBJECT-ORIENTED PROGRAMS

In this section we discuss the verification of "open" programs, the subtype-rule, and top-down development of types.

Open vs. Closed Programs In section 4, the notion of program execution was used to explain the meaning of a triple. To be more precise, we have to define what the executions of a SKOOL program are. We consider two cases: 1. A program may be declared to be *closed* (e.g., by adding some suitable keyword to the program). A closed program is considered complete and can be executed by calling some method of the program that takes only integer and boolean arguments as explicit parameters (recall from section 4 that methods can be invoked in a static fashion by using a null-object as implicit parameter). This defines all executions of a closed program. In a closed program, the subtype relation is complete; this is axiomatized as follows: For all abstract types T we add the following axiom to our programming logic:

$S \prec \mathrm{T} \ \Leftrightarrow\ S \preceq \mathrm{T}_1 \vee \ \ldots\ \vee S \preceq \mathrm{T}_k$
where $\mathrm{T}_1, \ldots, \mathrm{T}_k$ are the direct subtypes of T.

2. A program that is not closed is called *open*. It can be extended by adding new types. Open programs are very common in OO-programming. In particular, all libraries are open programs. The executions of an open program Π are all executions of all closed extensions of Π. Intuitively, open OO-programs are more difficult to verify because extensions can influence abstract types.

Bottom-up Verification According to the semantics of OO-languages, the properties of a virtual method in type T depend on the properties of the corresponding methods in T's subtypes. This kind of bottom-up verification, reflected by the subtype-rule, is illustrated in this paragraph: Let us assume a closed program declaring an abstract container type CO with an insert method and two subtypes: LI, a list implementation, and AR, an array-based implementation. LI and AR possess abstraction functions aL and aA that map their objects in a given object environment to elements of an abstract data type $Multiset$ with an insertion operation ins. Assume we have proved

$$\{\ \tau \preceq ct(\mathrm{LI}) \wedge aL(\mathrm{this}, \$) = M \wedge \mathrm{o} = O\ \}\ \mathrm{LI{:}insert(o)}\ \{\ aL(\mathrm{result}, \$) = ins(M, O)\ \}$$

i.e., if multiset M is the abstraction of *this* in the state before execution of `LI:insert` and O denotes the actual parameter object, the abstraction of the result is $ins(M, O)$. In order to prove a similar triple for `CO:insert`, we have to define an abstraction function for CO that works for objects of LI and AR (we write LI_c for $ct(\mathrm{LI})$, AR_c for $ct(\mathrm{AR})$, and CO_a for $at(\mathrm{CO})$):

$$\begin{array}{l} aC : Object \times ObjEnv \to Multiset \\ \quad typ(X) \preceq \mathrm{LI}_c \quad \Rightarrow \quad aC(X, E) = aL(X, E) \\ \quad typ(X) \preceq \mathrm{AR}_c \quad \Rightarrow \quad aC(X, E) = aA(X, E) \end{array}$$

Using this definition and the fact that the result of `LI:insert` is correctly typed (the latter assumption can be made w.l.g., because we can prove within the logic that SKOOL is a type-safe language; cf. [PH97] for proving type annotations), we can derive

$$\vdash \{\ \tau \preceq \mathrm{LI}_c \wedge aC(\mathrm{this}, \$) = M \wedge \mathrm{o} = O\ \}\ \mathrm{LI{:}insert(o)}\ \{\ aC(\mathrm{result}, \$) = ins(M, O)\}$$

The subtype rule with $\mathrm{LI}_c \preceq \mathrm{CO}_a$ yields:

$$\{\ \tau \preceq \mathrm{CO}_a \wedge \tau \not\preceq \mathrm{LI}_c \wedge \mathbf{P}\ \}\ \mathrm{CO{:}insert(o)}\ \{\ \mathbf{Q}\ \} \vdash \{\ \tau \preceq \mathrm{CO}_a \wedge \mathbf{P}\ \}\ \mathrm{CO{:}insert}\ \{\ \mathbf{Q}\ \}$$

where $\mathbf{P}$ abbreviates $aC(\mathrm{this}, \$) = M \wedge \mathrm{o} = O$ and $\mathbf{Q}$ abbreviates $aC(\mathrm{this}, \$) = ins(M, O)$. If we start from a specification of AR:insert similar to that given above for LI:insert and add $\tau \not\preceq \mathrm{LI}_c$ to the preconditions, we can use the same steps to derive:

$$\begin{array}{l} \{\ \tau \preceq \mathrm{CO}_a \wedge \tau \not\preceq \mathrm{AR}_c \wedge \tau \not\preceq \mathrm{LI}_c \wedge \mathbf{P}\ \}\ \mathrm{CO{:}insert(o)}\ \{\ \mathbf{Q}\ \} \\ \qquad \vdash \{\ \tau \preceq \mathrm{CO}_a \wedge \tau \not\preceq \mathrm{LI}_c \wedge \mathbf{P}\ \}\ \mathrm{CO{:}insert(o)}\ \{\ \mathbf{Q}\ \} \end{array}$$

The consequent of the derived sequent equals the assumption of the sequent derived first. Thus, we conclude:

$$\begin{array}{l} \{\ \tau \preceq \mathrm{CO}_a \wedge \tau \not\preceq \mathrm{AR}_c \wedge \tau \not\preceq \mathrm{LI}_c \wedge \mathbf{P}\ \}\ \mathrm{CO{:}insert(o)}\ \{\ \mathbf{Q}\ \} \\ \qquad \vdash \{\ \tau \preceq \mathrm{CO}_a \wedge \mathbf{P}\ \}\ \mathrm{CO{:}insert(o)}\ \{\ \mathbf{Q}\ \} \end{array}$$

As we assumed that the program is closed, the precondition of the assumption is false, so that the assumption can be eliminated (see false-axiom and assumpt-elim-rule in appendix). Thus, we have derived a property of the virtual method `CO:insert`.

Top-down Verification Usually, OO-programs are developed by refining existing types in open programs, i.e. by adding new subtypes to the program. Verification then means proving that a new subtype satisfies the specification of its supertypes. I.e., programs are developed together with their verified properties in a top-down manner. This methodology can be realized in our logic by the following *strategy*: 1. Provide a specification for each method of the program. We assume that a method specification SPEC(T:m) is a triple of the form $\{\,\tau \preceq \mathrm{T} \wedge \mathbf{P}_{\mathrm{T:m}}\,\}\ \mathrm{T{:}m}\ \{\,\mathbf{Q}_{\mathrm{T:m}}\,\}$. 2. If a new type S is added to the program, prove that the methods of S satisfy the specifications of supertype methods, i.e. for all direct supertypes T of S and all methods T:m show $\mathrm{SPEC(S{:}m)} \vdash \{\,\tau \preceq \mathrm{S} \wedge \mathbf{P}_{\mathrm{T:m}}\,\}\,\mathrm{S{:}m}\,\{\,\mathbf{Q}_{\mathrm{T:m}}\,\}$. 3. If S is a concrete type prove in addition that method implementations satisfy their specifications. In these proofs, the method specifications of the original program can be used.

Lemma: All methods of a closed program Π that was developed according to the strategy sketched above satisfy their specifications.

Proof: To prove the lemma, we show that all methods of Π satisfy their specification under certain assumptions that can be discarded for closed programs. Let $\mathrm{DST(T,\Pi)}$ denote the direct subtypes of type T in program Π and $\mathrm{VM(\Pi)}$ the virtual methods of Π. By $\mathcal{A}_\Pi$ we denote the following set of triples:

$$\bigcup_{\mathrm{T:m}\in\mathrm{VM}(\Pi)} \{\,\tau \preceq \mathrm{T} \ \wedge \bigwedge_{\mathrm{T'}\in\mathrm{DST}(\mathrm{T},\Pi)} \tau \not\preceq \mathrm{T'} \ \wedge\ \mathbf{P}_{\mathrm{T:m}}\,\}\ \mathrm{T{:}m}\ \{\,\mathbf{Q}_{\mathrm{T:m}}\,\}$$

$\mathcal{A}_\Pi$ formalizes the assumption that new subtypes satisfy the supertype specification. We show that $\mathcal{A}_\Pi \vdash \mathrm{SPEC(T{:}m)}$ can be derived for all T:m in a program Π that is developed together with its specification according to the above strategy, i.e. this sequent is a development invariant. The lemma is proved by induction on the number of extension steps*.

Induction Base: In the minimal program containing the types OBJECT, BOOL, and INT, the specifications of the predefined methods are given by axioms.

Induction Step: Let Π′ be an extension of Π by type S. We have to prove $\mathcal{A}_{\Pi'} \vdash \mathrm{SPEC(U{:}m)}$ for all U:m in Π′. We distinguish three cases: 1. U:m is a method of Π. 2. U = S and S is a concrete type. 3. U = S and S is abstract. The last case is trivial because $\mathrm{DST(S,\Pi')}$ is empty, so that $\mathrm{SPEC(S{:}m)} \in \mathcal{A}_{\Pi'}$.

*To focus on the application of the subtype-rule, we assume here that programs are extended type by type, i.e., we do not consider extensions by mutually recursive types.

For the first two cases we can assume $\mathcal{A}_\Pi \vdash \mathrm{SPEC(U{:}m)}$ (induction hypothesis in case 1, part 3 of strategy in case 2). Thus, it suffices to show $\mathcal{A}_{\Pi'} \vdash \mathcal{A}_\Pi$. To do that, we prove for each virtual method $\mathrm{T{:}m} \in \mathrm{VM}(\Pi)$ that the corresponding assumption can be derived from $\mathcal{A}_{\Pi'}$: If $\mathrm{S} \notin \mathrm{DST}(\mathrm{T}, \Pi')$, then $\mathrm{DST}(\mathrm{T}, \Pi') = \mathrm{DST}(\mathrm{T}, \Pi)$ and the triple about T:m is the same in $\mathcal{A}_\Pi$ and $\mathcal{A}_{\Pi'}$. The interesting case is $\mathrm{S} \in \mathrm{DST}(\mathrm{T}, \Pi')$. Starting with the subtype requirement guaranteed by the strategy (part 2), applying precondition strengthening, the subtype rule, and enlarging the assumption set, we derive (EXT(this) denotes $\bigwedge_{\mathrm{T}' \in \mathrm{DST}(\mathrm{T},\Pi)} \tau \not\preceq \mathrm{T}'$):

$$\frac{\mathrm{SPEC(S{:}m)} \quad \vdash \quad \{\, \tau \preceq \mathrm{S} \wedge \mathbf{P}_{\mathrm{T:m}} \,\}\ \mathrm{S{:}m}\ \{\, \mathbf{Q}_{\mathrm{T:m}} \,\}}{\mathrm{SPEC(S{:}m)} \quad \vdash \quad \{\, \tau \preceq \mathrm{S} \wedge \mathrm{EXT(this)} \wedge \mathbf{P}_{\mathrm{T:m}} \,\}\ \mathrm{S{:}m}\ \{\, \mathbf{Q}_{\mathrm{T:m}} \,\}}$$

$$\frac{\begin{array}{rl}\mathrm{SPEC(S{:}m)}\ , & \{\, \tau \preceq \mathrm{T} \wedge \tau \not\preceq \mathrm{S} \wedge \mathrm{EXT(this)} \wedge \mathbf{P}_{\mathrm{T:m}} \,\}\ \mathrm{T{:}m}\ \{\, \mathbf{Q}_{\mathrm{T:m}} \,\} \\ & \vdash\ \{\, \tau \preceq \mathrm{T} \wedge \mathrm{EXT(this)} \wedge \mathbf{P}_{\mathrm{T:m}} \,\}\ \mathrm{T{:}m}\ \{\, \mathbf{Q}_{\mathrm{T:m}} \,\}\end{array}}{\mathrm{SPEC(S{:}m)}\ , \mathcal{A}_{\Pi'} \quad \vdash \quad \{\, \tau \preceq \mathrm{T} \wedge \mathrm{EXT(this)} \wedge \mathbf{P}_{\mathrm{T:m}} \,\}\ \mathrm{T{:}m}\ \{\, \mathbf{Q}_{\mathrm{T:m}} \,\}}$$

If S:m is a virtual method, we are finished because $\mathrm{SPEC(S{:}m)} \in \mathcal{A}_{\Pi'}$. If S:m has an implementation, we can eliminate SPEC(S:m) from the assumptions using the implementation-rule together with the fact that part 3 of the strategy guarantees the existence of a proof of $\mathcal{A}_\Pi \vdash \mathrm{SPEC(S{:}m)}$. For brevity, we omit the technical details here. **end of proof**

This lemma shows that our logic can be used as the foundation for program development methods and that the assumptions occurring in the subtype rule can be kept implicit.

6 TREATING INHERITANCE

This section defines the kernel language KOOL by adding inheritance to SKOOL and presents the logical rules for verifying KOOL programs.

Inheritance Inheritance means using implementation parts from a type T to implement a type S. We impose the restriction that S has to be a subtype of T. (In most OO-languages S has to be a direct subtype.) To keep the definition of KOOL simple, we support only single inheritance, i.e., a type may only inherit from one type, as it is e.g. in Java.

A *KOOL program* is a finite set of type declarations of the following form (brackets indicate that the surrounded syntactic construct is optional):

```
[abstract] type T subtype  of    <AbsTypeList>
                 inherits from  <AbsType>
                 is             <MethDeclList> <AttrList> end
```

The `inherits`-clause defines a binary relation on types, denoted by $\sqsubset_{inh}$, i.e., S $\sqsubset_{inh}$ T means that the implementation of S directly inherits from T. The reflexive and transitive closure of this relation is denoted by $\sqsubseteq$. A method declaration in KOOL is either a method signature or method definition, i.e., abstract types may have implementations as well. These are defined as in SKOOL except that KOOL supports an additional invocation statement (see below). The *interface* IF(S) of a type S consists of (1) the signatures of methods declared in S (including attribute access methods) and (2) all signatures in the interface of T, S $\sqsubset_{inh}$ T, that are not overridden in S where *overridden* means that S contains a signature with the same method name. A method name may appear only once in an interface.

We say that a method S:m in IF(S) is *associated with implementations* if it is declared in S and has a (possibly predefined) body or if it is not declared in S and S $\sqsubset_{inh}$ T and T:m is associated with an implementation. In the first case, we say that S:m is *defined in* S. In concrete types, all methods must be associated with an implementation.

It is important to understand the relation between method signatures, implementations, and invocations in OO-programs. The implementation associated with a method T:m is not necessarily the implementation being executed on invocations of T:m. We illustrate this by a small example:

```
abstract type  T  subtype of OBJECT    is
  m(): OBJECT         is  result := false    end
  mm( x: T ): OBJECT  is  result := x.T:m()  end

type S  subtype of T  inherits from T  is
  m(): S              is  result := null(S)  end
end
```

Method T:m is overridden in S. An invocation `T:mm( null(S) )` would lead to the execution of the implementation associated with S:m. In particular, to verify properties of invocations like `result:= x.T:m()` one has to use the properties of the virtual method T:m that are derived from the subtypes and cannot rely on the implementation associated with T:m. To be able to distinguish between a virtual method and its associated implementation, we write T@m to refer to the implementation associated with m in T.

Most OO-languages make it possible to invoke overridden methods from overriding methods, mostly via static binding (e.g., in C++ overridden methods can be invoked using the scope resolution operator, in Java it can be done using the keyword `super`). In KOOL, implementations associated with overridden methods can be called with the following syntax*:

```
v :=  this . T@m( .. )
```

*This syntax can be considered as a verbose form of the Java syntax `super.m(..)`, because `super` refers to the superclass of the static type of `this`.

Such a *call-statement* is context-correct, if the parameter types are covariant, if the result type of T:m is a subtype of v, if the enclosing type of the call inherits from T, and if T:m is associated with an implementation.

Logic for KOOL The specification of the object environment is as in SKOOL. The sort *AttId* needs clarification: A concrete type TC contains all attributes being declared in TC or in a type from which TC inherits. As overriding of attributes is possible, there may exist several different attributes with the same name in TC. To avoid such ambiguities, attribute identifiers in KOOL are not only prefixed by the concrete type (as in SKOOL), but as well by the type the attribute is declared in. I.e., we write TC:T@a for an attribute a that is declared in type T and belongs to objects of type TC.

KOOL distinguishes between virtual methods and method implementations. Thus, we introduce implementation annotations as a third form of triples. Implementation annotations have the form $\{\ \mathbf{P}\ \}\ \ \mathrm{T@m(..)}\ \ \{\ \mathbf{Q}\ \}$ where $\mathbf{P}$ and $\mathbf{Q}$ are $\Gamma_{pre(m)}$-formulas and Γ_{post}-formulas, respectively. They may be used as assumptions in sequents. The programming logic of KOOL is described by modifying and extending the logic presented in section 4.

In SKOOL, the predefined implementations could be associated only with concrete types, but KOOL provides more flexibility. The predefined implementation of `equ` can be used for any type and can possibly be overridden in subtypes. Thus, we have to replace T:equ by T@equ in the axiom for `equ`, where T are the types in which `equ` is defined. In the axiom for `new`, we replace T:new by T@new, but T may only range over concrete types. The predefined implementations of the access methods for an attribute `a` declared in type T can operate on all objects of concrete types TC with $\mathrm{TC} \sqsubseteq \mathrm{T}$. Abbreviating $loc(\mathrm{this}, \mathrm{TC{:}T@a})$ by this.TC:T@a, we get:

$$\vdash \{\ prec(\mathrm{this}, \mathrm{TC}) \wedge \mathrm{TC} \sqsubseteq \mathrm{T} \wedge \mathbf{P}[\$(\mathrm{this.TC{:}T@a})/\mathrm{result}]\ \}\ \mathrm{T@get\_a():TR}\ \ \{\ \mathbf{P}\ \}$$

$$\vdash \{\ prec(\mathrm{this}, \mathrm{TC}) \wedge \mathrm{TC} \sqsubseteq \mathrm{T} \wedge \mathbf{P}[\$\langle \mathrm{this.TC{:}T@a} := \mathrm{p}\rangle/\$]\ \}\ \mathrm{T@set\_a(p:TR)}\ \{\ \mathbf{P}\ \}$$

These axioms enable overriding and dynamic binding of attribute access methods. In Java and C++, attributes are statically bound. We didn't use this semantics for KOOL since it leads to more rules. On the other hand, such rules become simpler, at least if inheritance and subtyping are strongly connected as e.g. in Java and C++.

The two rules given in section 4 for the invocation-statement hold the same way for the call-statement with method implementations instead of virtual methods. We show here the call-rule:

call-rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \ \mathrm{T@m}(\mathrm{p}_1, \ldots, \mathrm{p}_z)\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}[\mathrm{E}_1/\mathrm{p}_1, \ldots, \mathrm{E}_z/\mathrm{p}_z]\ \}\ \mathrm{v} := \mathrm{this}\ .\mathrm{T@m}(\mathrm{E}_1, \ldots, \mathrm{E}_z)\ \{\ \mathbf{Q}[\mathrm{v}/\mathrm{result}]\ \}}$$

We also use a similar adaptation of the var-rule. An implementation S@m is either defined in S or inherited from a type T. In the first case, properties of

S@m can be proved by the adapted implementation-rule, which is obtained from the rule for SKOOL by substituting T@m for T:m. In the second case, the properties of S@m are derived from those of T@m by the inheritance-rule, given below. Finally, we have to relate properties of implementations to properties of virtual methods. This can be done in concrete types where the behavior of virtual methods is defined by the associated implementation:

inheritance-rule:

$$\frac{\mathcal{A} \vdash \{\, \mathbf{P} \,\}\quad \text{T@m}\quad \{\, \mathbf{Q} \,\}}{\mathcal{A} \vdash \{\, \mathbf{P} \,\}\quad \text{S@m}\quad \{\, \mathbf{Q} \,\}}$$

if S@m is inherited from T.

concrete-type-rule:

$$\frac{\mathcal{A} \vdash \{\, \mathbf{P} \,\}\quad \text{T@m}\quad \{\, \mathbf{Q} \,\}}{\mathcal{A} \vdash \{\, \mathbf{P} \,\}\quad \text{T:m}\quad \{\, \mathbf{Q} \,\}}$$

if T is a concrete type.

7 CONCLUSIONS

This paper has presented logical foundations for the essential OO-language features: abstract types, multiple-subtyping, single-inheritance, and dynamic binding. We have shown how object environments can be formalized as first-order values. Based on this foundation, we presented a Hoare-style programming logic that allows one to prove properties of OO-programs. In particular, we discussed how subtyping, inheritance, and abstract types can be handled.

Our logic can be used for verifying OO-programs. This requires specifications of program properties, which are usually given as pre- and postconditions of methods, and as class invariants. [PH97] shows how such specifications can be transformed into triples of the logic and thus be verified.

We validated the logic by simulating dynamic binding via dispatching in pseudo-code. A formal validation against a complete operational semantics of the programming language would be desirable, in particular, if logics using these techniques are to be applied to realistic size languages like Java.

Acknowledgments
We are glad to thank the referees for their helpful comments.

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.

[Apt81] K. R. Apt. Ten years of Hoare logic: A survey — part I. *ACM Trans. on Prog. Languages and Systems*, 3:431–483, 1981.

[GH93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[GU91] T. Gergely and L. Úry. *First-Order Programming Theories*. Springer-Verlag, 1991.

[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. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[PH97] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

[Suz80] N. Suzuki, editor. *Automatic Verification of Programs with Complex Data Structures*. Garland Publishing, 1980.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. North-Holland, Amsterdam, 1990.

APPENDIX 1 SUMMARY OF PROGRAMMING LOGIC

In addition to the axioms and rules described in the sections above, the programming logic consists of the usual rules for if, while, sequential statements, strengthening/weakening of pre-/postconditions, and the following language-independent axioms and rules:

assumpt-axiom:

$$\mathbf{A} \vdash \mathbf{A}$$

false-axiom:

$$\vdash \{ \text{FALSE} \} \text{ COMP } \{ \text{FALSE} \}$$

assumpt-intro-rule:

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

assumpt-elim-rule:

$$\frac{\mathcal{A} \vdash \mathbf{A}_0 \qquad \mathbf{A}_0 \, , \, \mathcal{A} \vdash \mathbf{A}}{\mathcal{A} \vdash \mathbf{A}}$$

conjunct-rule:

$$\frac{\mathcal{A} \vdash \{ \mathbf{P}_1 \} \text{ COMP } \{ \mathbf{Q}_1 \} \qquad \mathcal{A} \vdash \{ \mathbf{P}_2 \} \text{ COMP } \{ \mathbf{Q}_2 \}}{\mathcal{A} \vdash \{ \mathbf{P}_1 \wedge \mathbf{P}_2 \} \text{ COMP } \{ \mathbf{Q}_1 \wedge \mathbf{Q}_2 \}}$$

disjunct-rule:

$$\frac{\mathcal{A} \vdash \{ \mathbf{P}_1 \} \text{ COMP } \{ \mathbf{Q}_1 \} \qquad \mathcal{A} \vdash \{ \mathbf{P}_2 \} \text{ COMP } \{ \mathbf{Q}_2 \}}{\mathcal{A} \vdash \{ \mathbf{P}_1 \vee \mathbf{P}_2 \} \text{ COMP } \{ \mathbf{Q}_1 \vee \mathbf{Q}_2 \}}$$

inv-rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P} \wedge \mathbf{R}\ \}\ \text{COMP}\ \{\ \mathbf{Q} \wedge \mathbf{R}\ \}}$$

where $\mathbf{R}$ is a Σ-formula, i.e. does not contain program variables.

subst-rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}[t/Z]\ \}\ \text{COMP}\ \{\ \mathbf{Q}[t/Z]\ \}}$$

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

all-rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}[Y/Z]\ \}\ \text{COMP}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}[Y/Z]\ \}\ \text{COMP}\ \{\ \forall Z : \mathbf{Q}\ \}}$$

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

ex-rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}[Y/Z]\ \}}{\mathcal{A} \vdash \{\ \exists Z : \mathbf{P}\ \}\ \text{COMP}\ \{\ \mathbf{Q}[Y/Z]\ \}}$$

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

BIOGRAPHIES

Arnd Poetzsch-Heffter is Professor at the University of Hagen. He received a Doctor in Computer Science from the Technical University of Munich in 1991 with a thesis about specification techniques for static semantics of programming languages. During a post-doc year at the CS department of Cornell University he developed an approach to integrate program specification and verification techniques for object-oriented programs. This is the topic of his Habilitation thesis. His main research interests include programming techniques and methods, language design, and system support. Currently, his work focuses on the construction of provably correct programs from correct components.

Peter Müller is a member of the Lopex project at the University of Hagen, Germany. He works in the field of specification and verification of object-oriented programs. In particular, he studies the verification of component-based programs. Before that, he worked at the Technical University of Munich, where he received a diploma in computer science. Topic of his thesis was the formal semantics of Sather and the development of an operational assertion language.