A Syntax-Directed Hoare Logic for Object-Oriented Programming Concepts

Cees Pierik[1] and Frank S. de Boer[1,2]

[1] Institute of Information and Computing Sciences
Utrecht University, The Netherlands
[2] CWI, Amsterdam, The Netherlands
{cees,frankb}@cs.uu.nl

Abstract. This paper outlines a sound and complete Hoare logic for a sequential object-oriented language with inheritance and subtyping like Java. It describes a weakest precondition calculus for assignments and object-creation, as well as Hoare rules for reasoning about (mutually recursive) method invocations with dynamic binding. Our approach enables reasoning at an abstraction level that coincides with the general abstraction level of object-oriented languages.

1 Introduction

The concepts of inheritance and subtyping in object-oriented programming have many virtues. But they also pose challenges for reasoning about programs. For example, subtyping enables variables with different types to be aliases of the same object, and it destroys the static connection between a method call and its implementation. Inheritance, without further restrictions, adds complexity by permitting objects to have different instance variables with the same identifier.

This paper outlines a Hoare logic for a sequential object-oriented language that contains all standard object-oriented features, including inheritance, subtyping and dynamic binding. The logic consists of a weakest precondition calculus for assignments and object-creation, as well as Hoare rules for reasoning about (mutually recursive) method invocations with dynamic binding. The resulting logic is complete in the sense that any valid correctness formula can be derived within the logic.

The Hoare logic presented in this paper is *syntax-directed*. By a syntax-directed Hoare logic we mean a Hoare logic that is based on an assertion language of the same abstraction level as the programming language. In particular, there is no explicit reference to the object store in our assertion language, as opposed to [1]. Moreover, our Hoare logic is based on a weakest precondition calculus that consists of purely *syntactical* substitution operations.

Hoare introduced the axiom $\{P[e/x]\}\ x := e\ \{P\}$ for reasoning about simple assignments in his seminal paper [2]. A *semantical* variant would be

$$\{P[\sigma\{x := e\}/\sigma]\}\ x := e\ \{P\}\ ,$$

E. Najm, U. Nestmann, and P. Stevens (Eds.): FMOODS 2003, LNCS 2884, pp. 64–78, 2003.

Table 1. The syntax of COORE.

Below, the operator `op` is an arbitrary operator on elements of a primitive type, and m is an arbitrary identifier.

$e \in$ Expr	$::= \texttt{null} \mid \texttt{this} \mid u \mid e.x \mid (C)e \mid e \texttt{ instanceof } C \mid \texttt{op}(e_1, \ldots, e_n)$
$y \in$ Loc	$::= u \mid e.x$
$s \in$ SExpr	$::= \texttt{new } C() \mid u.m(e_1, \ldots, e_n) \mid \texttt{super}.m(e_1, \ldots, e_n)$
$S \in$ Stat	$::= y = e \texttt{ ;} \mid y = s \texttt{ ;} \mid S\ S \mid \texttt{if } (e) \{ S \} \texttt{ else } \{ S\} \mid \texttt{while } (e) \{ S \}$
$meth \in$ Meth	$::= m(u_1, \ldots, u_n) \{ S \texttt{ return } e \texttt{ ;} \}$
$main \in$ Main	$::= \texttt{main}() \{ S \}$
$exts \in$ Exts	$::= \epsilon \mid \texttt{extends } C$
$class \in$ Class	$::= \texttt{class } C\ exts \{ meth^* \}$
$\pi \in$ Prog	$::= class^*\ main$

where $\sigma\{x := e\}$ denotes the state that results from σ by assigning $\sigma(e)$ to x. Here, the occurrence of σ shows the employed representation of the state, and state updates like $\sigma\{x := e\}$ reveal the encoding of the semantics. In the original approach, assertions have the same abstraction level as the programming language and hide all these details.

Another advantage of the syntax-directed approach can be explained by the following example. Suppose we want to prove $\{y = 1\}\ x := 0\ \{y = 1\}$. Using our approach, this amounts to proving the implication $y = 1 \rightarrow y = 1$. The semantical approach requires proving $\sigma(y) = 1 \rightarrow \sigma\{x := 0\}(y) = 1$. A theorem prover must do one additional reasoning step in this case, namely resolving that y is a different location than x. This step is otherwise encoded in the substitution. The minor difference in this example leads to larger differences, for example when reasoning about aliases. Our substitution operation precisely reveals in which cases we have to check for possible aliases. Finally, observe that the semantical approach requires an encoding of (elements of) the semantics of the programming language in the theorem prover.

This paper is organized as follows. In Sect. 2 and 3 we introduce the programming language and the assertion language. Section 4 and 5 describe the weakest precondition calculus for assignments and object creation. In Sect. 6 we give Hoare rules for reasoning about method calls. Related research is discussed in the last section. The completeness proof and other details of the logic are described in the full version of this paper [3].

2 The Object-Oriented Language

The language we consider in this paper (dubbed COORE) contains all standard object-oriented features like inheritance and subtyping. For ease of reading, we adopted the syntax of the corresponding subset of Java. The syntax of COORE can be found in Table 1.

The primitive types we consider are `boolean` and `int`. We assume given a set $\mathcal{C}$ of *class names*, with typical element C. The set of types $\mathcal{T}$ is the union of the set $\{\texttt{int}, \texttt{boolean}\}$ and $\mathcal{C}$. In the sequel, t will be a typical element of $\mathcal{T}$. The language is strongly-typed. By $[\![e]\!]$ we denote the (declared) static type

of expression e. The type of `this` is determined by its context. We will silently assume that all expressions are well-typed.

By TVar we denote the set of local (or temporary) variables. Each class C is equipped with a set of instance variables IVar_C. We use u and x as typical elements of the sets TVar and IVar_C, respectively. The location y is either a local variable or an instance variable. Instance variables belong to a specific object and store its internal state. Local variables belong to a method and last as long as this method is active. A method's formal parameters are also local variables.

A program in cOOre is a finite set of classes and a main method which initiates the execution of the program. A class defines a finite set of methods. A method m consists of its formal parameters $u_1, \ldots, u_n$, a statement S, and an expression e without side effect which denotes the return value. A clause `class` C `extends` C' indicates that class C is a subclass of C'. It implies that class C inherits all methods and instance variables of class C'.

The expression $e.x$ refers to the instance variable x of object e as found in class $[\![e]\!]$ or, if not present in $\mathrm{IVar}_{[\![e]\!]}$, the first occurrence of this variable in a superclass of $[\![e]\!]$. Observe that a class C can hide an inherited instance variable x by defining another instance variable x. An expression $e.x$, with $[\![e]\!] = C$, will then refer to the new variable. The cast operator (C) in $(C)e$ changes the type of expression e to C. Thus it can be used to access hidden variables. For example, $((C)\texttt{this}).x$ denotes the first occurrence of an instance variable x of object `this` as found by an upward search starting in class C. This might be a variable different from $\texttt{this}.x$. We assume that $[\![e]\!]$ in $(C)e$ is a reference type. An expression e `instanceof` C is `true` if e is non-null and refers to an instance of (a subclass of) class C.

We have the usual assignments of expressions to variables. An assignment $y =$ `new` $C()$ involves the creation of an object of class C. Note that the language does not include constructor methods declarations. Thus an expression like `new` $C()$ will call the default constructor method, which will assign to all instance variables their default value.

Observe that the callee of a method call can be denoted by a local variable only. We assume that all methods are public. An assignment $y = u.m(e_1, \ldots, e_n)$ involves a call of method m of the object denoted by the local variable u. These calls are bound dynamically to an implementation, depending on the class of the object denoted by u. Calls of the form $y = \texttt{super}.m(e_1, \ldots, e_n)$ are bound statically. The corresponding implementation is found by searching upwards in the class hierarchy for a definition of m, starting in the superclass of $[\![\texttt{this}]\!]$.

The language cOOre permits only side effects in the outermost operator. This is a common restriction in Hoare logics that clarifies the presentation of the logic. However, it is not essential. Early work by Kowaltowski already introduces a general approach to side effects [4]. On the other hand, one could argue that the restriction on side effects leads to more reliable programs. Gosling et al. remark: 'Code is usually clearer when each expression contains at most one side effect, as its outermost operation, and when code does not depend on exactly which exception arises as a consequence of the left-to-right evaluation of expressions.' [5, p. 305]

2.1 Semantics

In this section, we only describe the semantics of expressions because this suffices to understand the rest of the paper. The semantics of COORE is defined in terms of a representation of the state of an object-oriented program and a subtype relation.

By $t \preceq t'$ we denote that t is a subtype of t'. The relation $\preceq$ is given by the class definitions in the program. The declaration `class A extends B` implies that $A \lhd B$ (where $A \lhd B$ denotes that class A is a direct subclass of class B). In fact, the $\lhd$ relation is a partial function that defines the superclass of a class. Therefore we will assume that $F_{\lhd}(C)$ denotes the direct superclass of a class C. The partial order $\preceq$ is the reflexive, transitive closure of the $\lhd$ relation. We say that t' is a *proper* subtype of t, denoted by $t' \prec t$, if $t' \preceq t$ and $t' \neq t$.

We represent objects as follows. Each object has its own identity and belongs to a certain class. For each class $C \in \mathcal{C}$ we introduce therefore the infinite set $O^C = \{C\} \times \mathbb{N}$ of object identities in class C (here $\mathbb{N}$ denotes the set of natural numbers). Let $\texttt{subs}(C)$ be the set $\{C' \in \mathcal{C} | C' \preceq C\}$. By $\mathrm{dom}(C)$ we denote the set $(\bigcup_{C' \in \texttt{subs}(C)} O^{C'}) \cup \{\bot\}$. Here $\bot$ is the value of `null`. In general, $\bot$ stands for 'undefined'. For $t = \texttt{int}, \texttt{boolean}$, $\mathrm{dom}(t)$ denotes the set of boolean and integer values, respectively.

The internal state of an object $o \in O^C$ is a total function that maps the instance variables of class C and its superclasses to their values. Let $\texttt{supers}(C)$ be the set $\{C' \in \mathcal{C} | C \preceq C'\}$. The internal state of an instance of class C is an element of the set internal(C), which is defined by the (generalized) cartesian product

$$\prod_{C' \in \texttt{supers}(C)} \Big(\prod_{x \in \mathrm{IVar}_{C'}} \mathrm{dom}([\![x]\!]) \Big) \ .$$

A configuration σ is a partial function that maps each *existing* object to its internal state. We will assume that σ is an element of the set Σ, where

$$\Sigma = \prod_{C \in \mathcal{C}} \Big(\mathbb{N} \rightharpoonup \mathrm{internal}(C) \Big) \ .$$

In the sequel, we will write $\sigma(o)$ for some object $o = (C, n)$ as shorthand for $\sigma(C)(n)$. In this way, $\sigma(o)$ denotes the internal state of an object. It is not defined for objects that do not exist in a particular configuration σ. Thus σ specifies the set of existing objects. We will only consider configurations that are *consistent*. We say that a configuration is consistent if no instance variable of an existing object refers to a non-existing object.

The local context $\tau \in \mathrm{T}$ specifies the active object and the values of the local variables. Formally, T is the set

$$(\bigcup_{C \in \mathcal{C}} \mathrm{dom}(C)) \times \prod_{u \in \mathrm{TVar}} \mathrm{dom}([\![u]\!]).$$

The first component of any τ is the active object and the second component is a function which assigns to every local variable u its value. The first component

Table 2. Evaluation of expressions.

$$
\begin{array}{rcl}
\mathcal{E}(\texttt{null})(\sigma,\tau) & = & \bot \\
\mathcal{E}(\texttt{this})(\sigma,\tau) & = & \tau(\texttt{this}) \\
\mathcal{E}(u)(\sigma,\tau) & = & \tau(u) \\
\mathcal{E}(e.x)(\sigma,\tau) & = & \begin{cases} \bot & \text{if } \mathcal{E}(e)(\sigma,\tau) = \bot \\ \sigma(\mathcal{E}(e)(\sigma,\tau))(\mathsf{resolve}([\![e]\!])(x))(x) & \text{otherwise} \end{cases} \\
\mathcal{E}(\ (C)e\ )(\sigma,\tau) & = & \begin{cases} \bot & \text{if } \mathcal{E}(e)(\sigma,\tau) = (C',n) \text{ and } C' \not\preceq C \\ \mathcal{E}(e)(\sigma,\tau) & \text{otherwise} \end{cases} \\
\mathcal{E}(e\ \texttt{instanceof}\ C)(\sigma,\tau) & = & \begin{cases} \mathsf{false} & \text{if } \mathcal{E}(e)(\sigma,\tau) = \bot \\ C' \preceq C & \text{if } \mathcal{E}(e)(\sigma,\tau) = (C',n) \end{cases} \\
\mathcal{E}(\texttt{op}(e_1,\ldots,e_n))(\sigma,\tau) & = & \begin{cases} \bot & \text{if } e_i' = \bot \text{ for some } i = 1,\ldots,n \\ \bar{\texttt{op}}(e_1',\ldots,e_n') & \text{otherwise,} \end{cases} \\
 & & \text{where } \bar{\texttt{op}} \text{ denotes the fixed interpretation of } \texttt{op}, \text{ and} \\
 & & e_i' = \mathcal{E}(e_i)(\sigma,\tau), \text{ for } i = 1,\ldots,n.
\end{array}
$$

will be $\bot$ if there is no active object, which is the case during execution of the main method. In the sequel, we denote the first component o of a local context $\tau = \langle o, f\rangle$ by $\tau(\texttt{this})$ and $f(u)$ by $\tau(u)$. Although the local state of the main method can be $\langle \bot, f\rangle$, we will assume in other methods that the first element is an existing object. A local state is consistent with a global configuration if all local variables do not refer to non-existing objects. A state is a pair (σ,τ), where the local context τ is required to be consistent with the configuration σ.

To find fields in an internal state we need a way to determine in which class a field is declared. As explained above, the type of the quantifier e determines to which field an expression $e.x$ refers. We introduce a function $\mathsf{resolve}$ that yields the class of the field to which the expression $e.x$ refers given $[\![e]\!]$ and x. It is defined as follows.

$$
\mathsf{resolve}(C)(x) = \begin{cases} C & \text{if } x \in \mathrm{IVar}_C \\ \mathsf{resolve}(F_{\lhd}(C))(x) & \text{otherwise} \end{cases}
$$

Expressions are evaluated relative to a subclass relation $\preceq$, a configuration σ, and a local context τ. The result of the evaluation of an expression e is denoted by $\mathcal{E}(e)(\sigma,\tau)$. The $\preceq$ relation is left implicit. The definition of $\mathcal{E}$ can be found in Table 2.

3 The Assertion Language

The proof system is tailored to a specific assertion language called AsO (Assertion language for Object structures). The syntax of AsO is defined by the following grammar.

$$
\begin{array}{rcl}
l \in \mathrm{LExpr} & ::= & \texttt{null} \mid \texttt{this} \mid u \mid z \mid l.x \mid (C)l \mid l_1 = l_2 \mid l\ \texttt{instanceof}\ C \\
 & & \mid \texttt{op}(l_1,\ldots,l_n) \mid \texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi} \\
P, Q \in \mathrm{Ass} & ::= & l_1 = l_2 \mid \neg P \mid P \wedge Q \mid \exists z : t(P)
\end{array}
$$

Table 3. Evaluation of assertions.

$$
\begin{array}{rcl}
\mathcal{L}(z)(\sigma,\tau,\omega) & = & \omega(z) \\
\mathcal{L}(l_1 = l_2)(\sigma,\tau,\omega) & = & \begin{cases} \mathsf{true} \text{ if } \mathcal{L}(l_1)(\sigma,\tau,\omega) = \mathcal{L}(l_2)(\sigma,\tau,\omega) \\ \mathsf{false} \text{ otherwise} \end{cases} \\
\mathcal{L}(\texttt{if } l_1 \texttt{ then } l_2 \texttt{ else } l_3 \texttt{ fi})(\sigma,\tau,\omega) & = & \begin{cases} \bot & \text{if } \mathcal{L}(l_l)(\sigma,\tau,\omega) = \bot \\ \mathcal{L}(l_2)(\sigma,\tau,\omega) & \text{if } \mathcal{L}(l_1)(\sigma,\tau,\omega) = \mathsf{true} \\ \mathcal{L}(l_3)(\sigma,\tau,\omega) & \text{if } \mathcal{L}(l_1)(\sigma,\tau,\omega) = \mathsf{false} \end{cases}
\end{array}
$$

In the assertion language we assume a set of (typed) *logical* variables LVar with typical element z. We include expressions of the form $(C)l$ to be able to access hidden instance variables. The use of l `instanceof` C will become clear in Sect. 6. We sometimes omit the type information in $\exists z : t(P)$ if it is clear from the context.

The assertion language is strongly-typed similar to the programming language. Logical variables can additionally have type t^*, for some t in the old set of types $\mathcal{T}$. This means that its value is a finite sequence of elements of $\mathrm{dom}(t)$. Therefore $\mathrm{dom}(t^*)$ is the set of finite sequences of elements of $\mathrm{dom}(t)$.

Assertion languages for object-oriented programs inevitably contain expressions like $l.x$ and $(C)l$ that are normally undefined if, for example, l is `null`. However, as an assertion their value should be defined. We solved this problem by giving such expression the same value as `null`. By only allowing the non-strict equality operator as a basic formula, we nevertheless ensure that formulas are two-valued. If we omit this operator the value is implicitly compared to `true`. An alternative solution which is employed in JML [6] is to return an arbitrary element of the underlying domain.

Logical expressions are evaluated relative to a subclass relation $\preceq$, a configuration σ, a local context τ, and a logical environment $\omega \in \prod_{z \in \mathrm{LVar}} \mathrm{dom}([\![z]\!])$, which assigns values to the logical variables. The logical environment is restricted similar to a local context: no logical variable is allowed to point to an object that does not exist in the current configuration.

The result of the evaluation of an expression l is denoted by $\mathcal{L}(l)(\sigma,\tau,\omega)$. Again, we leave the $\preceq$ relation implicit. The function $\mathcal{L}$ is similar to $\mathcal{E}$ for all constructs that are present in cOOre. All new cases are listed in Table 3.

A formula $\exists z : C(P)$ states that P holds for an *existing* instance of (a subclass of) C or `null`. Thus the quantification domain of a variable depends not only on the type of the variable but also on the configuration. Let $\mathrm{qdom}(t,\sigma)$ denote the quantification domain of a variable of type t in configuration σ. We define $\mathrm{qdom}(C,\sigma) = \{o \in \mathrm{dom}(C) | \sigma(o) \text{ is defined } \} \cup \{\bot\}$. A formula $\exists z : C^*(P)$ states the existence of a sequence of existing objects. Therefore, we define

$$\mathrm{qdom}(C^*,\sigma) = \{\alpha \in \mathrm{dom}(C^*) | \forall n \in \mathbb{N}.\alpha[n] \in \mathrm{qdom}(C,\sigma)\} \ .$$

Finally, we have $\mathrm{qdom}(t,\sigma) = \mathrm{dom}(t)$ for $t \in \{\texttt{int}, \texttt{boolean}, \texttt{int}^*, \texttt{boolean}^*\}$.

The evaluation of a formula P can be defined similar to the evaluation of a logical expression. The resulting value is denoted by $\mathcal{A}(P)(\sigma,\tau,\omega)$. The only

interesting case is $\mathcal{A}(\exists z : t(P))(\sigma, \tau, \omega)$, which yields true if $\mathcal{A}(P)(\sigma, \tau, \omega\{\alpha/z\}) = \mathsf{true}$ for some $\alpha \in \mathrm{qdom}(t, \sigma)$ and false otherwise.

The standard abbreviations like $\forall z P$ for $\neg\exists z \neg P$ are valid. The statement $\sigma, \tau, \omega \models P$ means that $\mathcal{A}(\mathrm{P})(\sigma, \tau, \omega)$ yields true.

4 Assignments and Aliasing

This section shows how we model simple assignments by means of a substitution operation. The basic underlying idea as originally introduced in [2] is that the assertion that results from the substitution has the same meaning in the state before the assignment as the unmodified assertion in the state after the assignment. In other words, the substitution computes the *weakest precondition*.

First we observe that given an assignment $u = e$, and a *postcondition* P, the assertion $P[e/u]$ obtained from P by replacing every occurrence of u by e is *not* the weakest precondition. Subtyping combined with dereferencing is the cause of this phenomenon. Subtyping allows u and e to have different types. This implies that the substitution $[e/u]$ might change the type of an expression l. The only restriction imposed by the language is that $[\![e]\!] \preceq [\![u]\!]$.

To see where things go wrong, consider the following case. Suppose we have two classes C_1 and C_2 such that $C_2 \prec C_1$. Furthermore, assume that in each of the two classes an instance variable x of type `int` is defined. Finally, suppose that we have two local variables u_1 and u_2, such that $[\![u_1]\!] = C_1$ and $[\![u_2]\!] = C_2$. Now consider the specification of the following assignment.

$$\{u_2.x = 3\}\ u_1 = u_2;\ \{u_1.x = 3\}$$

This specification is not valid. Clearly, we have that $u_1.x = 3[u_2/u_1] \equiv u_2.x = 3$ (denoting syntactical equality by $\equiv$). But the expressions $u_1.x$ and $u_2.x$ point to different locations, even if u_1 and u_2 refer to the same object, because the types of u_1 and u_2 are different. A correct specification would be

$$\{((C_1)u_2).x = 3\}\ u_1 = u_2;\ \{u_1.x = 3\}\ .$$

The above specification presents the key to the solution of this problem. The result of the substitution $[e/u]$ should be changed in such a way that the types remain unchanged. This can be done by changing the result of $u[e/u]$ to $\mathsf{cast}?([\![u]\!], e)$. The auxiliary function $\mathsf{cast}?(t, l)$ is defined as follows.

$$\mathsf{cast}?(t, l) = \begin{cases} l & \text{if } t \text{ is a primitive type} \\ (t)l & \text{otherwise} \end{cases}$$

All other cases of the substitution $[e/u]$ correspond to the standard notion of (structural) substitution. We will assume in the rest of this paper that a substitution of the form $[e/u]$ corresponds to this modified substitution operation. The following theorem states that $P[e/u]$ is the weakest precondition of P with respect to the assignment $u = e$.

Theorem 1. *If* $[\![e]\!] \preceq [\![u]\!]$ *we have*

$$\sigma, \tau, \omega \models P[e/u] \text{ if and only if } \sigma, \tau', \omega \models P \ ,$$

where τ' *denotes the local state that results from* τ *by assigning* $\mathcal{E}(e)(\sigma, \tau)$ *to* u*.*

The above theorem justifies the axiom $\{P[e/u]\}\ u = e\ \{P\}$. A crucial part of the proof consists of showing type preservation of $[e/u]$, that is, proving that $[\![l[e/u]]\!] = [\![l]\!]$.

For the same reason, the usual notion of substitution does not suffice for an assignment $e.x = e'$. But such assignments are also complicated because of possible aliases of the location $e.x$, namely expressions of the form $l.x$. It is possible that l refers to the object denoted by e (before the assignment), which implies that $l.x$ denotes the same location as $e.x$ and should be substituted by e'. If l does not refer to object e no substitution should take place. If we cannot decide between these possibilities by the form of the expression and their types, a conditional expression is constructed which decides dynamically.

The definition of the substitution operation $[e'/e.x]$ is straightforward in most cases. The most interesting case is $l.x[e'/e.x]$. It results in one of the two following expressions. If $[\![l]\!] \preceq [\![e]\!]$ or $[\![e]\!] \preceq [\![l]\!]$ and, moreover, $\mathsf{resolve}([\![l]\!])(x) = \mathsf{resolve}([\![e]\!])(x)$, we have

$$l.x[e'/e.x] \equiv \texttt{if}\ l[e'/e.x] = e\ \texttt{then}\ \mathsf{cast}?([\![l.x]\!], e')\ \texttt{else}\ (l[e'/e.x]).x\ \texttt{fi}\ .$$

Otherwise, we simply have $l.x[e'/e.x] \equiv (l[e'/e.x]).x$. This can be explained as follows. Note that if $[\![l]\!] \not\preceq [\![e]\!]$ and $[\![e]\!] \not\preceq [\![l]\!]$ then the expressions l and e cannot refer to the same object, because the domains of $[\![l]\!]$ and $[\![e]\!]$ are disjoint. The clause $\mathsf{resolve}([\![l]\!])(x) = \mathsf{resolve}([\![e]\!])(x)$ checks if the two occurrences of x denote the same instance variable. We define $l.y[e'/e.x] \equiv (l[e'/e.x]).y$ if x and y are syntactically different. The definition is extended to assertions in the standard way.

As a simple example, we consider the assignment $\texttt{this}.x = 0$ and the postcondition $u.y.x = 1$, where x and y are instance variables and u is a local variable. Considering types, we assume in this example that $[\![u.y]\!] = [\![\texttt{this}]\!] = C$ and $[\![u]\!] = C'$. Applying the corresponding substitution $[0/\texttt{this}.x]$ to the assertion $u.y.x = 1$ results in the assertion

$$(\texttt{if}\ u.y = \texttt{this}\ \texttt{then}\ 0\ \texttt{else}\ u.y.x\ \texttt{fi}) = 1\ .$$

This assertion clearly is logically equivalent to $\neg(u.y = \texttt{this}) \wedge u.y.x = 1$.

The following theorem states that $P[e'/e.x]$ is indeed the weakest precondition of the assertion P with respect to the assignment $e.x = e'$. Thus it justifies the axiom $\{P[e'/e.x]\}e.x = e'\{P\}$. Its proof again requires showing type preservation of the substitution.

Theorem 2. *If* $[\![e']\!] \preceq [\![e.x]\!]$ *and* $\mathcal{E}(e)(\sigma, \tau) \neq \perp$ *we have*

$$\sigma, \tau, \omega \models P[e'/e.x] \text{ if and only if } \sigma', \tau, \omega \models P \ ,$$

where $\sigma'(o)(\mathsf{resolve}([\![e]\!])(x))(x) = \mathcal{E}(e')(\sigma, \tau)$*, for* $o = \mathcal{E}(e)(\sigma, \tau)$*, and in all other cases* σ *agrees with* σ'*.*

5 Object Creation

Next we consider the creation of objects. Our goal is to define a substitution $[\texttt{new}\ C/u]$ which computes the weakest precondition of the assignment $u = \texttt{new}\ C()$. Note that an assignment $e.x = \texttt{new}\ C()$ can be simulated by the sequence of assignments $u = \texttt{new}\ C(); e.x = u$, where u is a fresh local variable. The weakest precondition of an assignment $e.x = \texttt{new}\ C$ w.r.t. postcondition P is therefore $P[u/e.x][\texttt{new}\ C/u]$, where u is a fresh local variable which does not occur in P and e.

For certain logical *expressions* l we cannot define a weakest precondition $l[\texttt{new}\ C/u]$, because they refer to the new object, and there is no expression that refers to this object in the state before its creation, because it does not exist in that state. Therefore the result of the substitution must be left undefined in some cases. However, we can define the substitution on any logical expression l that is not of the form u, $(C')u$ or $\texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi}$ with $[\![l_2]\!] = [\![l_3]\!] \in \mathcal{C}$. This suffices to define $[\texttt{new}\ C/u]$ on any *formula*. We will do so by means of a contextual analysis of the occurrences of u.

To simplify the definition of $[\texttt{new}\ C/u]$ we start by rewriting logical expressions into a normal form. This proceeds in two steps. Firstly, we remove all occurrences of casts of conditional expressions by means of the following equivalence.

$$(C)\texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi} = \texttt{if}\ l_1\ \texttt{then}\ (C)l_2\ \texttt{else}\ (C)l_3\ \texttt{fi}$$

Secondly, we remove all expressions of the form $(C')(C'')u$. Observe that such an expression is either equivalent to $(C')u$ or $(C')\texttt{null}$, depending on the validity of $C \preceq C''$. In both steps we replace an expression by an expression of the same type.

Due to space limitations we cannot give all cases of $l[\texttt{new}\ C/u]$. But we trust that the general idea becomes clear by considering the following examples, starting with $l.x[\texttt{new}\ C/u]$. In general, we have to give special attention to cases where $l[\texttt{new}\ C/u]$ may be undefined. Therefore we single out the case where $l \equiv u$.

$$u.x[\texttt{new}\ C/u] \equiv \begin{cases} \mathsf{false} & \text{if } [\![u.x]\!] = \texttt{boolean} \\ 0 & \text{if } [\![u.x]\!] = \texttt{int} \\ \texttt{null} & \text{otherwise} \end{cases}$$

This example shows that we can find an equivalent expression even if the substitution is undefined for l. The instance variables of a new object have their default values after creation. These values depend on the types of the variables, which is reflected by the above substitution. The case where $l \equiv (C')u$ is similar, but requires additionally checking if $C \preceq C'$ to predict if the cast will succeed. If the cast fails, one can return `null`.

The other special case is that of a conditional expression. Suppose that $[\![(\texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi}).x]\!] = C'$. Then we define

$$\begin{aligned} &(\texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi}).x[\texttt{new}\ C/u] \\ &\equiv \texttt{if}\ l_1[\texttt{new}\ C/u]\ \texttt{then}\ ((C')l_2).x[\texttt{new}\ C/u]\ \texttt{else}\ ((C')l_3).x[\texttt{new}\ C/u]\ \texttt{fi}\ . \end{aligned}$$

In all other cases we have $l.x[\texttt{new}\ C/u] \equiv (l[\texttt{new}\ C/u]).x$.

The changing scope of a bound occurrence of a variable z ranging over objects, as induced by the creation of a new object, is captured as follows. We define $(\exists z : C'(P))[\texttt{new}\ C/u]$

$$\equiv \begin{cases} (\exists z : C'(P[\texttt{new}\ C/u])) \vee (P[(C')u/z][\texttt{new}\ C/u]) & \text{if } C \preceq C' \\ \exists z : C'(P[\texttt{new}\ C/u]) & \text{otherwise.} \end{cases}$$

The idea of the application of $[\texttt{new}\ C/u]$ to $(\exists z P)$ is that the first disjunct $\exists z(P[\texttt{new}\ C/u])$ represents the case that P holds for an old object whereas the second disjunct $P[(C')u/z][\texttt{new}\ C/u]$ represents the case that the new object itself satisfies P. The substitution $[(C')u/z]$ consists of simply replacing every occurrence of z by $(C')u$. The other cases of $l[\texttt{new}\ C/u]$ and $P[\texttt{new}\ C/u]$ can be found in [3].

The following theorem states that $P[\texttt{new}\ C/u]$ indeed calculates the weakest precondition of P (with respect to the assignment $u = \texttt{new}\ C()$).

Theorem 3. *Let $[\![u]\!] \preceq C$. Then we have*

$$\sigma, \tau, \omega \models P[\texttt{new}\ C/u] \text{ if and only if } \sigma', \tau', \omega \models P \ ,$$

where σ' is obtained from σ by extending the domain of σ with a new object $o = (C, n) \notin \mathrm{qdom}(C, \sigma)$ and setting its instance variables at their default values. Furthermore, the resulting local context τ' is obtained from τ by assigning o to the variable u.

6 Method Invocations

In this section we discuss the rules for method invocations. In particular, we will analyze reasoning about dynamically bound method calls like in the statement $S \equiv y = u.m(e_1, \ldots, e_n)$. A correctness formula $\{P\}S\{Q\}$ implies that Q holds after the call independent of which implementation is executed. Therefore we must consider all implementations of m that are defined in (a subclass of) $[\![u]\!]$, and the implementation that is inherited by class $[\![u]\!]$ if it does not contain an implementation of method m itself.

The challenge in this section is to show that our assertion language is able to express the conditions under which an implementation is bound to a particular call given the restriction imposed by the abstraction level. That is, by using only expressions from the programming language. Secondly, we aim to define and present the rules in such a way that their translation to proof obligations in proof outlines for object-oriented programs is straightforward. For both these reasons we cannot adopt the virtual methods approach as proposed in [1].

We first consider a statement of the form $y = \texttt{super}.m(e_1, \ldots, e_n)$, because this allows us to explain many features of our approach while postponing the complexity of late binding. Suppose that the statement occurred somewhere in the definition of a class C. Assume that searching for the definition of m starting in the superclass of C ends in class C' with the following implementation $m(u_1, \ldots, u_n)\ \{\ S\ \texttt{return}\ e\ \}$. Then the invocation $\texttt{super}.m(e_1, \ldots, e_n)$

is bound to this particular implementation. The following rule for overridden method invocations (OMI) allows the derivation of a correctness specification for $y = \texttt{super}.m(e_1, \ldots, e_n)$ from a correctness specification of the body S of the implementation of m.

$$\frac{\{P' \wedge I\}S\{Q'[e/\texttt{return}]\} \quad Q'[(C')\texttt{this}/\texttt{this}][\bar{f}/\bar{z}] \rightarrow Q[\texttt{return}/y]}{\{P'[(C')\texttt{this}, \bar{e}/\texttt{this}, \bar{u}][\bar{f}/\bar{z}]\}\ y = \texttt{super}.m(e_1, \ldots, e_n)\ \{Q\}} \quad \text{(OMI)}$$

The precondition P' and postcondition Q' of S are transformed into corresponding conditions of the call by the substitution $[(C')\texttt{this}/\texttt{this}]$. This substitution reflects the context switch. The active object is the same in both contexts, but its type differs. The substitution corrects this. It corresponds to the standard notion of structural substitution, but should take place simultaneously with the (also simultaneous) substitutions $[\bar{e}/\bar{u}]$. These substitutions model the assignment of the actual parameters $\bar{e} = e_1, \ldots, e_n$ to the formal parameters $\bar{u} = u_1, \ldots, u_n$. Note that we have for every formal parameter u_i and corresponding actual parameter e_i that $[\![e_i]\!] \preceq [\![u_i]\!]$. So the simultaneous substitution we mean here is the generalization of $[e/u]$ as defined in Sect. 4. Except for the formal parameters $u_1, \ldots, u_n$, no other local variables are allowed in P'. We do not allow local variables in Q'.

We cannot simply substitute y by the result value e in Q, because e may refer to local variables of S that might clash with local variables of the caller. Therefore we introduce a logical variable `return`. The substitution $[e/\texttt{return}]$ applied to the postcondition Q' of S in the first premise models a (virtual) assignment of the result value to the logical variable `return`, which must not occur in the assertion Q. The related substitution $[\texttt{return}/y]$ applied to the postcondition Q of the call models the actual assignment of the return value to y. The substitution corresponds to one of the enhanced notions of substitution as defined in Sect. 4.

The assertion I in the precondition of S specifies the initial values of the local variables of m (excluding its formal parameters): In cOOre we have $u = \mathsf{false}$, in case of a boolean local variable, $u = 0$, in case of an integer variable, and $u = \texttt{null}$, for a reference variable.

Next we observe that the local state of the caller is not affected by the execution of S by the receiver. For this reason we know that an expression that only refers to local variables of the caller or the keyword `this` is constant during a method call. Such an expression f is generated by the following abstract syntax.

$$f ::= \texttt{null} \mid \texttt{this} \mid u \mid (C)f \mid f_1 = f_n \mid f\ \texttt{instanceof}\ C \mid \texttt{op}(f_1, \ldots, f_n)$$
$$\mid \texttt{if}\ f_1\ \texttt{then}\ f_2\ \texttt{else}\ f_3\ \texttt{fi}$$

A sequence of such expressions $\bar{f}$ can be substituted for a corresponding sequence of logical variables $\bar{z}$ of exactly the same type in the specification of the body S. Without these substitutions one cannot prove anything about the local state of the caller after the method invocation.

Next, we analyze reasoning about method invocations that are dynamically bound to an implementation like in the statement $S \equiv y = u.m(e_1, \ldots, e_n)$. For this purpose we first define some abbreviations.

Firstly, we formalize the set of classes that provide an implementation of a particular method. Assume that $\mathsf{methods}(C)$ denotes the set of method identifiers for which an implementation is given in class C. The function impl yields the class that provides the implementation of a method m for objects of a particular class. It is defined as follows.

$$\mathsf{impl}(C)(m) = \begin{cases} C & \text{if } m \in \mathsf{methods}(C) \\ \mathsf{impl}(F_{\lhd}(C))(m) & \text{otherwise} \end{cases}$$

We can generalize the above definition to get all implementations that are relevant to a particular domain. This results in the following definition.

$$\mathsf{impls}(C)(m) = \{C' \in \mathcal{C} | \mathsf{impl}(C'')(m) = C' \text{ for some class } C'' \text{ with } C'' \preceq C\} \ .$$

Thus the set $\mathsf{impls}([\![u]\!])(m)$ contains all classes that provide an implementation of method m that might be bound to the call $u.m(e_1, \ldots, e_n)$.

Another important issue when reasoning about methods calls is which classes inherit a particular implementation of a method. For that reason we consider the subclasses of a class C that override the implementation given in class C. We denote this set by $\mathsf{overrides}(C)(m)$. We have $C' \in \mathsf{overrides}(C)(m)$ if C' is a proper subclasses of C with $m \in \mathsf{methods}(C')$ and there does not exist another proper subclass C'' of C such that $C' \prec C''$ and $m \in \mathsf{methods}(C'')$. With this definition we can formulate the condition for an implementation of m in class C to be bound to a method call $u.m(e_1, \ldots, e_n)$. It is

$$u \ \texttt{instanceof} \ C \wedge \neg u \in \mathsf{overrides}(C)(m) \ ,$$

where the latter clause abbreviates the conjunction

$$\bigwedge_{C' \in \mathsf{overrides}(C)(m)} \neg(u \ \texttt{instanceof} \ C') \ .$$

We now have all building blocks for reasoning about a specification of the form $\{P\}\ y = u.m(e_1, \ldots, e_n)\ \{Q\}$. Assume that $\mathsf{impls}([\![u]\!])(m) = \{C_1, \ldots, C_k\}$. Let $\{\ S_i \ \texttt{return}\ e_i\ \}$ be the body of the implementation of method m in class C_i, for $i = 1, \ldots, k$, and let $\bar{u_i}$ be its formal parameters. To derive a specification for $y = u.m(e_1, \ldots, e_n)$ we have to prove that for each implementation S_i a specification $B_i \equiv \{P_i \wedge I_i\} S_i \{Q_i[e_i/\texttt{return}_i]\}$ holds. Moreover, this specification should satisfy certain restrictions. First of all, the assertions P_i and Q_i must satisfy the same conditions as the assertions P and Q in the rule OMI. The assertion I_i is similar to the assertion I in that rule. Secondly, the preconditions of the implementations must be implied by the precondition of the call. That is, we must prove the following implications.

$$P \wedge u \ \texttt{instanceof} \ C_i \wedge \neg u \in \mathsf{overrides}(C_i)(m) \\ \rightarrow P_i[(C_i)u, \bar{e}/\texttt{this}, \bar{u}_i][\bar{f}/\bar{z}] \qquad (\vec{P}_i)$$

Similarly, we have to check wether the postconditions of the implementations imply the postcondition of the call. This requires proving the following formulas.

$$Q_i[(C_i)u/\mathtt{this}][\bar{f}/\bar{z}] \to Q[\mathtt{return}_i/y] \qquad (\vec{Q}_i)$$

The rule for dynamically-bound method invocations (DMI) then simply says that all given implications should hold and, moreover, we have to derive the specifications of the bodies.

$$\frac{\vec{P}_1, \ldots, \vec{P}_k \qquad B_1, \ldots, B_k \qquad \vec{Q}_1, \ldots, \vec{Q}_k}{\{P\}\ y = u.m(e_1, \ldots, e_n)\ \{Q\}} \qquad \text{(DMI)}$$

The generalization of the rule for non-recursive method invocations to one for recursive and even mutually recursive method invocations is a variant of the classical recursion rule. The idea behind the classical rule is to prove correctness of the specification of the body of the call on the assumption that the method call satisfies its specification. Our rule for mutually recursive method invocations (MRMI) allows both dynamically bound method invocations and calls to overridden methods in the recursion chain. To enable this it combines the rules (OMI) and (DMI). The outline of the rule is as follows.

$$\frac{F_1, \ldots, F_r \vdash \bar{B}_1, \ldots, \bar{B}_r \qquad \bar{P}_1, \ldots, \bar{P}_r \qquad \bar{Q}_1, \ldots, \bar{Q}_r}{F_1} \qquad \text{(MRMI)}$$

The formulas $F_1, \ldots, F_r$ are the specifications of the calls that occur in the recursion chain. That is, we require that each F_j is a correctness formula about a method invocation. As a naming convention, we assume that each F_j is of the form

$$\{P_j\}\ y_j = u_j.m_j(e^j_1, \ldots, e^j_{n_j})\ \{Q_j\} \text{ or } \{P_j\}\ y_j = \mathtt{super}.m_j(e^j_1, \ldots, e^j_{n_j})\ \{Q_j\}\ .$$

The formulas $\bar{B}_j$, for $j = 1, \ldots, r$, denote sequences of correctness formulas about all possible implementations of the call in F_j. $\bar{B}_j$ contains only one element if F_j concerns a call to an overridden method. The sequences $\bar{P}_j$ and $\bar{Q}_j$ are the corresponding compatibility checks for the pre- and postconditions. Each element in $\bar{P}_j$ and $\bar{Q}_j$ corresponds to the element at the same position in F_j.

Let us first consider the case where F_j is of the form

$$\{P_j\}\ y_j = u_j.m_j(e^j_1, \ldots, e^j_{n_j})\ \{Q_j\}\ .$$

Assume that $\mathsf{impls}([\![u_j]\!])(m_j) = \{C_1, \ldots, C_{k_j}\}$. Let $\{\ S_i\ \mathtt{return}\ e_i\ \}$ be the body of the implementation of method m_j in class C_i, for $i = 1, \ldots, k_j$, and let $\bar{u}_i$ be its formal parameters. Then $\bar{B}_j$ is the sequence containing, for $i = 1, \ldots, k_j$, the correctness formulas $\{P'_i \wedge I_i\} S_i \{Q'_i[e_i/\mathtt{return}_i]\}$. The formula $\bar{P}_j$ is the sequence containing, for $i = 1, \ldots, k_j$, the implications

$$P_j \wedge u_j\ \mathtt{instanceof}\ C_i \wedge \neg u_j \in \mathsf{overrides}(C_i)(m_j) \to P'_i[(C_i)u_j, \bar{e}_j/\mathtt{this}, \bar{u}_i][\bar{f}/\bar{z}_i]\ .$$

And finally, $\bar{Q}_j$ is the sequence containing, for $i = 1, \ldots, k_j$, the implications

$$Q'_i[(C_i)u_j/\texttt{this}][\bar{f}/\bar{z}_i] \rightarrow Q_j[\texttt{return}_i/y_j] \ .$$

On the other hand, if F_j is of the form $\{P_j\}\ y_j = \texttt{super}.m_j(e^j_1, \ldots, e^j_{n_j})\ \{Q_j\}$ we can statically determine to which implementation this call is bound. Suppose that it is bound to the implementation $m_j(u_1, \ldots, u_n)\{\ S\ \texttt{return}\ e\}$ in class C. Then the sequence $\bar{B}_j$ contains only the formula $\{P'_j \wedge I_j\}\ S\ \{Q'_j[e/\texttt{return}_j]\}$. The compatibility check $\bar{P}_j$ is $P_j \rightarrow P'_j[(C)\texttt{this}, \bar{e}_j/\texttt{this}, \bar{u}_j][\bar{f}_j/\bar{z}_j]$ and $\bar{Q}_j$ is $Q'_j[(C)\texttt{this}/\texttt{this}][\bar{f}_j/\bar{z}_j] \rightarrow Q_j[\texttt{return}_j/y_j]$. In the formula $\bar{P}_j$, $\bar{e}_j = e^j_1, \ldots, e^j_{n_j}$ and $u_i = u_1, \ldots, u_n$.

7 Conclusions

The main result of this paper is a syntax-directed Hoare logic for a language that has all standard object-oriented features. The logic extends our work as presented in [7, 8] by covering inheritance, subtyping and dynamic binding. We prove that the proposed Hoare logic is (relatively) complete in the full paper [3].

In recent years, several Hoare logics for (sequential) fragments of object-oriented languages, in particular Java, were proposed. However, the formal justification of existing Hoare logics for object-oriented languages is still under investigation. For example, completeness is still un open question for many Hoare logics in this area (see, e.g., [1, 9]).

However, in [10], a Hoare logic for a substantial sequential subset of Java is proved complete. This Hoare logic formalizes correctness proofs directly in terms of a semantics of the subset of Java in Isabelle/HOL. Higher order logic is used as specification language. As observed by the author this results in a serious discrepancy between the abstraction level of the Hoare logic and the programming language.

We are currently putting the finishing touch to a tool that enables the application of our logic to larger test-cases. It supports the annotation of programs, fully automatically computes the resulting verification conditions, and feeds them to a theorem prover. We aim to make this tool publicly available soon.

Checking the verification conditions is in general not decidable. However, we plan to investigate the isolation of a decidable subset of the present assertion language which would still allow, for example, aliasing analysis. Future work also includes the integration of related work on reasoning about abrupt termination [11] and concurrency in an object-oriented setting [12].

Finally, we would like to give a compositional formulation of the logic presented in this paper which will be based on invariants that specify the externally observable behavior of the objects in terms of the send and received messages. We envisage the use of temporal logics as described in [13] for the formulation of such invariants.

References

1. Poetzsch-Heffter, A., Müller, P.: A programming logic for sequential Java. In Swierstra, S.D., ed.: ESOP '99. Volume 1576 of LNCS. (1999) 162–176
2. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12** (1969) 576–580
3. Pierik, C., de Boer, F.S.: A syntax-directed Hoare logic for object-oriented programming concepts. Technical Report UU-CS-2003-010, Institute of Information and Computing Sciences, Utrecht University, The Netherlands (2003) Available at http://www.cs.uu.nl/research/techreps/UU-CS-2003-010.html.
4. Kowaltowski, T.: Axiomatic approach to side effects and general jumps. Acta Informatica **7** (1977) 357–360
5. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison-Wesley (1996)
6. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Department of Computer Science, Iowa State University (2003)
7. de Boer, F.S.: A wp-calculus for OO. In Thomas, W., ed.: FoSSaCS '99. Volume 1578 of LNCS. (1999) 135–149
8. de Boer, F., Pierik, C.: Computer-aided specification and verification of annotated object-oriented programs. In Jacobs, B., Rensink, A., eds.: FMOODS V, Kluwer Academic Publishers (2002) 163–177
9. Reus, B., Wirsing, M., Hennicker, R.: A Hoare calculus for verifying Java realizations of OCL-constrained design models. In Hussmann, H., ed.: FASE 2001. Volume 2029 of LNCS. (2001) 300–317
10. von Oheimb, D.: Hoare logic for Java in Isabelle/HOL. Concurrency and Computation: Practice and Experience **13** (2001) 1173–1214
11. Huisman, M., Jacobs, B.: Java program verification via a Hoare logic with abrupt termination. In Maibaum, T., ed.: FASE 2000. Volume 1783 of LNCS. (2000) 284–303
12. Abraham-Mumm, E., de Boer, F., de Roever, W., Steffen, M.: Verification for Java's reentrant multithreading concept. In: FoSSaCS '02. Volume 2303 of LNCS. (2002) 5–20
13. Distefano, D., Katoen, J.P., Rensink, A.: On a temporal logic for object-based systems. In Smith, S.F., Talcott, C.L., eds.: FMOODS III, Kluwer Academic Publishers (2000) 305–326