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

Cees Pierik

Frank S. de Boer

institute of information and computing sciences, utrecht university

technical report UU-CS-2003-010

www.cs.uu.nl

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

Cees Pierik and Frank S. de Boer
Institute of Information and Computing Sciences
Utrecht University, The Netherlands
{cees, frankb}@cs.uu.nl

version: 1.1[1]

17 June 2003

[1]version 1.0: 28 March 2003; version 1.1: 17 June 2003

Abstract

This report presents 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.

Chapter 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 report 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.

Our syntax-directed approach 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 [13]. Moreover, our weakest precondition calculus 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 [9]. A *semantical* variant would be $\{P[\sigma\{x := e\}/\sigma]\}\ x := e\ \{P\}$, 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 $\sigma(y) = 0 \rightarrow \sigma\{x := 0\}(y) = 0$. 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 the semantics of the programming language in the theorem prover.

This report is organized as follows. In Chapter 2 and 3 we introduce the programming language and the assertion language. In Chapter 4 we describe the weakest precondition calculus for assignments and object creation and we give Hoare rules for reasoning about method calls. The completeness proof can be found in Chapter 5. Related research is discussed in the last chapter.

Chapter 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 2.1.

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. Every expression has a (declared) static type. By $[\![e]\!]$ we denote the 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 local variables and instance variables, respectively. The variable 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 direct 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 $\text{IVar}_{[\![e]\!]}$, 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 this latter 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

Table 2.1: 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	$::= y = e\ ; \mid y = s\ ; \mid S\ S \mid \texttt{if}\ (e)\ \{\ S\ \}\ \texttt{else}\ \{\ S\} \mid \texttt{while}\ (e)\ \{\ S\ \}$
$meth \in$ Meth	$::= m(u_1, \ldots, u_n)\ \{\ S\ \texttt{return}\ e\ ;\ \}$
$main \in$ Main	$::= \texttt{main}()\ \{\ S\ \}$
$exts \in$ Exts	$::= \epsilon \mid \texttt{extends}\ C$
$class \in$ Class	$::= \texttt{class}\ C\ exts\ \{\ meth^*\ \}$
$\pi \in$ Prog	$::= class^*\ main$

upward search starting in class C. This might be a variable different from `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 our language does not include constructor methods. 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 only be denoted by a local variable. We made this assumption for technical convenience only. We assume that all methods are public. An assignment $y = u.m(e_1, \ldots, e_n)$ denotes a call of the 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 $y =$ `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 $[\![$`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 [11]. 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.' [8, p. 305]

2.1 Semantics

In this section, we will only describe the overall functionality of the semantics of the presented language because this suffices to understand the rest of the report. 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 subtype 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. It is undefined if C has no superclass. We chose not to assume that every class is a subclass of the class Object, as in Java, because that would correspond to a particular instance of the more generic proof system described in this report. 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 $\mathtt{subs}(C)$ be the set $\{C' \in \mathcal{C} | C' \preceq C\}$. By $\mathrm{dom}(C)$ we denote the set $(\bigcup_{C' \in \mathtt{subs}(C)} O^{C'}) \cup \{\bot\}$. Here $\bot$ is the value of `null`. In general, $\bot$ stands for 'undefined'. For $t =$ `int`, `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 $\mathtt{supers}(C)$ 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 (nested) cartesian product

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

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

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

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

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

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 will be $\perp$ 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 $\langle \perp, f \rangle$, 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 l determines to which field an expression $l.x$ refers. We introduce a function $\mathsf{resolve}$ that yields the class of the field to which the expression $l.x$ refers given $[\![l]\!]$ 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 evaluating an expression e is denoted by $\mathcal{E}(e)(\sigma, \tau)$. The $\preceq$ relation is not updated in the definitions below and is therefore omitted as an argument of $\mathcal{E}$. We can distinguish the following cases.

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

Given a set of class definitions the semantics of statements is given by the (strict) function:

$$\mathcal{S} : \mathrm{Stat} \to (\Sigma \times \mathrm{T})_{\perp} \to (\Sigma \times \mathrm{T})_{\perp},$$

such that $\mathcal{S}(S)(\sigma,\tau) = (\sigma',\tau')$ indicates that the execution of S in the state (σ,τ) terminates in the state (σ',τ') (note that as such S is in fact executed by $\tau(\texttt{this})$ and that $\tau'(\texttt{this}) = \tau(\texttt{this})$). Divergence is denoted by $\perp$. A compositional characterization of $\mathcal{S}$ (which is fairly standard) can be given following [4].

Chapter 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.

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

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. 4.3. We sometimes omit the type 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 also have type t^*, for some $t \in \mathcal{T}$. This means that its value is a finite sequence of elements of $\text{dom}(t)$. To reason about sequences we assume the presence of notations to express the length of a sequence (denoted by $|l|$) and the selection of an element of a sequence (denoted by $l[n]$, where n is an integer expression). More precisely, we assume in this report that the elements of a sequence are indexed by $1, \ldots, n$, for some natural number $n \geq 0$ (the sequence is of zero length, i.e., empty, if $n = 0$). Accessing a sequence with an index which is out of its bounds results in the value $\perp$.

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`. All other operators are strict. They simply pass the result of an (otherwise) undefined expression to the enclosing equality operator. An alternative solution which is employed in JML [12] 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 $\omega \in \prod_{z \in \text{LVar}} \text{dom}([\![z]\!])$, which assigns values to the logical variables. The logical environment is restricted similar to a local context: no logical variable points 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. We can distinguish the following cases.

$$
\begin{aligned}
\mathcal{L}(\texttt{null})(\sigma,\tau,\omega) &= \bot \\
\mathcal{L}(\texttt{this})(\sigma,\tau,\omega) &= \tau(\texttt{this}) \\
\mathcal{L}(u)(\sigma,\tau,\omega) &= \tau(u) \\
\mathcal{L}(z)(\sigma,\tau,\omega) &= \omega(z) \\
\mathcal{L}(l.x)(\sigma,\tau,\omega) &= \begin{cases} \bot & \text{if } l' = \bot \\ \sigma(l')(\mathsf{resolve}([\![l]\!],x))(x) & \text{otherwise} \end{cases} \\
&\quad \text{where } l' = \mathcal{L}(l)(\sigma,\tau,\omega) \\
\mathcal{L}(\ (C)l\ )(\sigma,\tau,\omega) &= \begin{cases} \bot & \text{if } \mathcal{L}(l)(\sigma,\tau,\omega) = (C',n) \text{ and } C' \not\preceq C \\ \mathcal{L}(l)(\sigma,\tau,\omega) & \text{otherwise} \end{cases} \\
\mathcal{L}(l_1 = l_2)(\sigma,\tau,\omega) &= \begin{cases} \mathsf{true} & \text{if } \mathcal{L}(l_1)(\sigma,\tau,\omega) = \mathcal{L}(l_2)(\sigma,\tau,\omega) \\ \mathsf{false} & \text{otherwise} \end{cases} \\
\mathcal{L}(l \ \texttt{instanceof}\ C)(\sigma,\tau,\omega) &= \begin{cases} \bot & \text{if } \mathcal{L}(l)(\sigma,\tau,\omega) = \bot \\ C' \preceq C & \text{if } \mathcal{L}(l)(\sigma,\tau,\omega) = (C',n) \end{cases} \\
\mathcal{L}(\texttt{op}(l_1,\ldots,l_n))(\sigma,\tau,\omega) &= \begin{cases} \bot & \text{if } l'_i = \bot \text{ for some } i = 1,\ldots,n \\ \bar{\texttt{op}}(l'_1,\ldots,l'_n) & \text{otherwise,} \end{cases} \\
&\quad \text{where } \bar{\texttt{op}} \text{ denotes the fixed interpretation of } \texttt{op}, \text{ and} \\
&\quad l'_i = \mathcal{L}(l_i)(\sigma,\tau,\omega), \text{ for } i = 1,\ldots,n. \\
\mathcal{L}(\texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi})(\sigma,\tau,\omega) &= \begin{cases} \bot & \text{if } \mathcal{L}(l_l)(\sigma,\tau,\omega) = \bot \\ \mathcal{L}(l_2)(\sigma,\tau,\omega) & \text{if } \mathcal{L}(l_1)(\sigma,\tau,\omega) = \mathsf{true} \\ \mathcal{L}(l_3)(\sigma,\tau,\omega) & \text{if } \mathcal{L}(l_1)(\sigma,\tau,\omega) = \mathsf{false} \end{cases}
\end{aligned}
$$

The evaluation of a formula P can be defined similar to the evaluation of a logical expression. The resulting value is denoted by $\mathcal{A}(P)(\sigma,\tau,\omega)$. As already explained above, 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 $t \in \{\texttt{int}, \texttt{boolean}, \texttt{int}^*, \texttt{boolean}^*\}$. We then define the meaning of formulas as follows.

$$
\begin{aligned}
\mathcal{A}(l_1 = l_2)(\sigma,\tau,\omega) &= \begin{cases} \mathsf{true} & \text{if } \mathcal{L}(l_1)(\sigma,\tau,\omega) = \mathcal{L}(l_2)(\sigma,\tau,\omega) \\ \mathsf{false} & \text{otherwise} \end{cases} \\
\mathcal{A}(\neg P)(\sigma,\tau,\omega) &= \begin{cases} \mathsf{true} & \text{if } \mathcal{A}(P)(\sigma,\tau,\omega) = \mathsf{false} \\ \mathsf{false} & \text{otherwise} \end{cases} \\
\mathcal{A}(P \wedge Q)(\sigma,\tau,\omega) &= \begin{cases} \mathsf{true} & \text{if } \mathcal{A}(P)(\sigma,\tau,\omega) = \mathsf{true} \text{ and } \mathcal{A}(Q)(\sigma,\tau,\omega) = \mathsf{true} \\ \mathsf{false} & \text{otherwise} \end{cases} \\
\mathcal{A}(\exists z : t(P))(\sigma,\tau,\omega) &= \begin{cases} \mathsf{true} & \text{if } \mathcal{A}(P)(\sigma,\tau,\omega\{\alpha/z\}) = \mathsf{true} \text{ for some } \alpha \in \mathrm{qdom}(t,\sigma) \\ \mathsf{false} & \text{otherwise} \end{cases}
\end{aligned}
$$

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, whereas $\models P$ denotes that $\sigma,\tau,\omega \models P$ for any mutually consistent σ,τ, and ω.

Chapter 4

The Proof System

In this chapter we introduce step-by-step a Hoare logic that covers all language constructs of an object-oriented language with inheritance, subtyping and late-binding. For statements that are not discussed in this chapter, the standard Hoare rules suffice. These rules can be found in, for example, [2].

Given a set of class definitions, correctness formulas have the usual form $\{P\}S\{Q\}$, where P and Q are assertions and S is a statement. We say that a correctness formula $\{P\}S\{Q\}$ is true w.r.t. a logical environment ω and a configuration (σ, τ), written as $\omega, \sigma, \tau \models \{P\}S\{Q\}$, if $\sigma, \tau, \omega \models P$ and $\mathcal{S}(S)(\sigma, \tau) = (\sigma', \tau')$ implies $\sigma', \tau', \omega \models Q$. This corresponds with the standard *partial* correctness interpretation of pre/postcondition specifications.

By $\models \{P\}S\{Q\}$, i.e., the correctness formula $\{P\}S\{Q\}$ is valid, we denote that $\omega, \sigma, \tau \models \{P\}S\{Q\}$, for every logical environment ω, and state (σ, τ) that are consistent with respect to the existing objects of σ. By $\vdash \{P\}S\{Q\}$ we denote that $\{P\}S\{Q\}$ has been derived by applying a finite number of rules and axioms of the logic that is presented in this chapter.

4.1 Assignments and Aliasing

In this section we show how we can model assignments involving aliasing in the assertion language by means of substitutions. The basic underlying idea as originally introduced in [9] is that the assertion resulting from the application of a 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 $u = e$, with u a local variable, and a *postcondition* P, the assertion $P[e/u]$ obtained from P by replacing every occurrence of u by e does *not* have the same meaning as the unmodified assertion P in the state after the assignment. Subtyping combined with dereferencing is the cause of this phenomenon. For an assignment of the form $u = e$, subtyping implies that u and e need not have the same type. The only restriction is that $[\![e]\!] \preceq [\![u]\!]$. That is, the type of e is a subtype of $[\![u]\!]$. This implies that the substitution $[e/u]$ might change the type of an assertion P.

To see where things go wrong, consider the following case. Suppose we have two classes C_1 and C_2 such that $C_2 \preceq C_1$. 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.

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

Is this specification valid? Clearly, we have that $u_1.x = 3[u_2/u_1] \equiv u_2.x = 3$. 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. We have to change the result of the substitution $[e/u]$ in such a way that the type remains unchanged. This can be done by changing the following case (syntactic equality is denoted by $\equiv$).

$$u[e/u] \equiv \mathsf{cast}?([\![u]\!], e)$$

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 report 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 $u = e$. It justifies the axiom $\{P[e/u]\}\ u = e\ \{P\}$.

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.

Proof Note that the clause $[\![e]\!] \preceq [\![u]\!]$ ensures that τ' is a valid local state. Observe that it suffices to prove, by structural induction on l, that

$$\mathcal{L}(l[e/u])(\sigma, \tau, \omega) = \mathcal{L}(l)(\sigma, \tau', \omega) \text{ and } [\![l[e/u]]\!] = [\![l]\!].$$

We deal with the three most interesting cases here. First, let $l \equiv u$. If $[\![u]\!]$ is a primitive type, it must be the case that $[\![e]\!] = [\![u]\!]$ and hence $[\![u[e/u]]\!] = [\![\mathsf{cast}?([\![u]\!], e)]\!] = [\![e]\!] = [\![u]\!]$. Otherwise, we infer $[\![u[e/u]]\!] = [\![\mathsf{cast}?([\![u]\!], e)]\!] = [\![([\![u]\!])e]\!] = [\![u]\!]$. Furthermore, we have

$$\begin{array}{rcl} \mathcal{L}(u[e/u])(\sigma, \tau, \omega) & = & \mathcal{L}(\mathsf{cast}?([\![u]\!], e))(\sigma, \tau, \omega) \\ & = & \mathcal{L}(e)(\sigma, \tau, \omega) \\ & = & \tau'(u) \\ & = & \mathcal{L}(u)(\sigma, \tau', \omega). \end{array}$$

Observe that the second step is valid because $[\![e]\!] \preceq [\![u]\!]$ holds. This excludes the possibility that $\mathcal{L}(e)(\sigma, \tau, \omega)$ is an object that cannot be cast to type $[\![u]\!]$. Secondly, let $l \equiv (C)l'$. We can prove type preservation in this case by observing that $[\![((C)l')[e/u]]\!] = [\![(C)(l'[e/u])]\!] = C = [\![(C)l']\!]$. By the induction hypothesis we have $\mathcal{L}(l'[e/u])(\sigma, \tau, \omega) = \mathcal{L}(l')(\sigma, \tau', \omega)$. A few calculation steps then prove this case.

$$\begin{array}{rcl} \mathcal{L}(((C)l')[e/u])(\sigma, \tau, \omega) & = & \mathcal{L}((C)(l'[e/u]))(\sigma, \tau, \omega) \\ & = & \mathcal{L}((C)l')(\sigma, \tau', \omega) \end{array}$$

Actually, the last step requires a trivial case split. Finally, let $l \equiv l'.x$. By the induction hypothesis we have $[\![l'[e/u]]\!] = [\![l']\!]$. But then $l'[e/u].x$ and $l'.x$ denote the same instance variable and we obtain $[\![l'.x[e/u]]\!] = [\![l'[e/u].x]\!] = [\![l'.x]\!]$. We must also prove that

$$\mathcal{L}(l'.x[e/u])(\sigma, \tau, \omega) = \mathcal{L}(l'.x)(\sigma, \tau', \omega).$$

This requires us to consider two cases. First we assume that $\mathcal{L}(l'[e/u])(\sigma, \tau, \omega) = \bot$. By the induction hypothesis we infer that $\mathcal{L}(l')(\sigma, \tau', \omega) = \bot$. We then calculate as follows.

$$\mathcal{L}(l'.x[e/u])(\sigma, \tau, \omega) = \mathcal{L}(l'[e/u].x)(\sigma, \tau, \omega) = \bot = \mathcal{L}(l'.x)(\sigma, \tau', \omega)$$

In the second case we assume that $\mathcal{L}(l'[e/u])(\sigma,\tau,\omega) \neq \bot$. By the induction hypothesis this implies that $\mathcal{L}(l')(\sigma,\tau',\omega) \neq \bot$. The proof then proceeds as follows.

$$\begin{array}{rcl}
\mathcal{L}(l'.x[e/u])(\sigma,\tau,\omega) & = & \mathcal{L}(l'[e/u].x)(\sigma,\tau,\omega) \\
& = & \sigma(\mathcal{L}(l'[e/u])(\sigma,\tau,\omega))(\mathsf{resolve}([\![l'[e/u]]\!],x))(x) \\
& = & \sigma(\mathcal{L}(l')(\sigma,\tau',\omega))(\mathsf{resolve}([\![l'[e/u]]\!],x))(x) \\
& = & \sigma(\mathcal{L}(l')(\sigma,\tau',\omega))(\mathsf{resolve}([\![l']\!],x))(x) \\
& = & \mathcal{L}(l'.x)(\sigma,\tau',\omega)
\end{array}$$

The other cases are straightforward. □

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), so that $l.x$ denotes the same location as $e.x$ and should be substituted by e'. It is also possible that l does not refer to the object e, and in this case 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.

We first list the simple cases of the substitution operation $[e'/e.x]$.

$$\begin{array}{rcl}
l[e'/e.x] & \equiv & l, \text{ for } l \in \{\texttt{null}, \texttt{this}, u, z\} \\
(C)l[e'/e.x] & \equiv & (C)l[e'/e.x] \\
(l_1 = l_2)[e'/e.x] & \equiv & l_1[e'/e.x] = l_2[e'/e.x] \\
(l \texttt{ instanceof } C)[e'/e.x] & \equiv & (l[e'/e.x]) \texttt{ instanceof } C \\
\texttt{op}(l_1,\ldots,l_n)[e'/e.x] & \equiv & (\texttt{op}(l_1[e'/e.x]),\ldots,l_n[e'/e.x]) \\
(\texttt{if } l_1 \texttt{ then } l_2 \texttt{ else } l_3 \texttt{ fi})[e'/e.x] & \equiv & \texttt{if } l_1[e'/e.x] \texttt{ then } l_2[e'/e.x] \texttt{ else } l_3[e'/e.x] \texttt{ fi}
\end{array}$$

The first clause deals with the base cases of the keywords `null` and `this`, a temporary variable u, and a logical variable z. In the penultimate clause we have that $\texttt{op}[e'/e.x] \equiv \texttt{op}$, in case $n = 0$ (i.e., in case of a constant). Next, we consider the cases of this substitution where we have to account for possible aliases.

$$(l.x)[e'/e.x] \equiv \begin{cases} \texttt{if } l' = e \texttt{ then } \mathsf{cast}?([\![l.x]\!], e') \texttt{ else } l'.x \texttt{ fi} & \text{if } [\![l]\!] \preceq [\![e]\!] \text{ or } [\![e]\!] \preceq [\![l]\!] \text{ and} \\ & \mathsf{resolve}([\![l]\!],x) = \mathsf{resolve}([\![e]\!],x) \\ l'.x & \text{otherwise} \end{cases}$$

where $l' \equiv (l[e'/e.x])$

$$(l.y)[e'/e.x] \equiv (l[e'/e.x]).y$$

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 $[\![e]\!]$ and $[\![l]\!]$ are disjoint. The clause $\mathsf{resolve}([\![l]\!],x) = \mathsf{resolve}([\![e]\!],x)$ checks if the two occurrences of x denote the same instance variable. In the second clause we assume that the instance variables x and y are distinct. 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 $e.x = e'$.

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

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

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'.

Proof Observe that $[\![e']\!] \preceq [\![e.x]\!]$ implies that $\sigma' \in \Sigma$. It suffices to prove by induction on the complexity of l that

$$\mathcal{L}(l[e'/e.x])(\sigma,\tau,\omega) = \mathcal{L}(l)(\sigma',\tau,\omega) \text{ and } [\![l[e'/e.x]]\!] = [\![l]\!].$$

Only one case turns out to be interesting. Naturally, that is the case where $l \equiv l'.x$. Therefore, we will show

$$\mathcal{L}(l'.x[e'/e.x])(\sigma,\tau,\omega) = \mathcal{L}(l'.x)(\sigma',\tau,\omega) \text{ and } [\![l'.x[e'/e.x]]\!] = [\![l'.x]\!].$$

Let us first prove the latter clause. The induction hypothesis allows us to assume that $[\![l'[e'/e.x]]\!] = [\![l']\!]$. That implies $[\![l'[e'/e.x].x]\!] = [\![l'.x]\!]$. The definition of $l'.x[e'/e.x]$ now forces us to make a case split. Fortunately, the second case follows immediately from our previous observation. The first case occurs when $[\![l']\!] \preceq [\![e]\!]$ or $[\![e]\!] \preceq [\![l']\!]$ and $\mathsf{resolve}([\![l']\!],x) = \mathsf{resolve}([\![e]\!],x)$. The latter clause implies that $[\![l'.x]\!] = [\![e.x]\!]$, since both expressions denote the same instance variable. We have to show that

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

Observe that it suffices to prove that $[\![\mathsf{cast}?([\![l'.x]\!],e')]\!] = [\![l'[e'/e.x].x]\!]$. If $[\![l'.x]\!]$ is a primitive type, we reason as follows.

$$[\![\mathsf{cast}?([\![l'.x]\!],e')]\!] = [\![e']\!] = [\![e.x]\!] = [\![l'.x]\!]$$

If $[\![l'.x]\!]$ is not a primitive type, we can do the following steps.

$$[\![\mathsf{cast}?([\![l'.x]\!],e')]\!] = [\![([\![l'.x]\!])e']\!] = [\![l.x]\!]$$

This proves that $[\![l'.x[e'/e.x]]\!] = [\![l'.x]\!]$. We now turn to the claim that

$$\mathcal{L}(l'.x[e'/e.x])(\sigma,\tau,\omega) = \mathcal{L}(l'.x)(\sigma',\tau,\omega).$$

By the induction hypothesis we have $\mathcal{L}(l'[e'/e.x])(\sigma,\tau,\omega) = \mathcal{L}(l')(\sigma',\tau,\omega)$. We first consider the case where $[\![l']\!] \preceq [\![e]\!]$ or $[\![e]\!] \preceq [\![l']\!]$ and $\mathsf{resolve}([\![l']\!],x) = \mathsf{resolve}([\![e]\!],x)$. So we have

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

We have to distinguish two cases. First let

$$\mathcal{L}(l'[e'/e.x])(\sigma,\tau,\omega) = \mathcal{L}(e)(\sigma,\tau,\omega).$$

We then calculate as follows.

$$\begin{array}{ll} & \mathcal{L}(\texttt{if } l'[e'/e.x] = e \texttt{ then } \mathsf{cast}?([\![l'.x]\!],e') \texttt{ else } l'[e'/e.x].x \texttt{ fi})(\omega,\sigma,\tau) \\ = & \mathcal{L}(e')(\sigma,\tau,\omega) \\ = & \sigma'(\mathcal{L}(e)(\sigma,\tau,\omega))(\mathsf{resolve}([\![e]\!],x))(x) \\ = & \sigma'(\mathcal{L}(l'[e'/e.x])(\sigma,\tau,\omega))(\mathsf{resolve}([\![e]\!],x))(x) \\ = & \sigma'(\mathcal{L}(l')(\sigma',\tau,\omega))(\mathsf{resolve}([\![e]\!],x))(x) \\ = & \sigma'(\mathcal{L}(l')(\sigma',\tau,\omega))(\mathsf{resolve}([\![l']\!],x))(x) \\ = & \mathcal{L}(l'.x)(\sigma',\tau,\omega) \end{array}$$

Observe that the second step is valid because $[\![e']\!] \preceq [\![e.x]\!] = [\![l'.x]\!]$. Next, let

$$\mathcal{L}(l'[e'/e.x])(\sigma,\tau,\omega) \neq \mathcal{L}(e)(\sigma,\tau,\omega).$$

By construction of σ' it follows that

$$\sigma(\mathcal{L}(l'[e'/e.x])(\sigma,\tau,\omega))(\mathsf{resolve}([\![l']\!],x))(x) = \sigma'(\mathcal{L}(l'[e'/e.x])(\sigma,\tau,\omega))(\mathsf{resolve}([\![l']\!],x))(x).$$

Then the proof proceeds as follows (assuming that $\mathcal{L}(l'[e'/e.x])(\sigma,\tau,\omega) \neq \bot$).

$$\begin{array}{ll} & \mathcal{L}(\texttt{if } l'[e'/e.x] = e \texttt{ then } \mathsf{cast}?([\![l'.x]\!], e') \texttt{ else } l'[e'/e.x].x \texttt{ fi})(\omega,\sigma,\tau) \\ = & \mathcal{L}(l'[e'/e.x].x)(\sigma,\tau,\omega) \\ = & \mathcal{L}(l.x)(\sigma',\tau,\omega) \end{array}$$

This finishes the first half of the proof. The other case occurs when $[\![l']\!] \not\preceq [\![e]\!]$ and $[\![e]\!] \not\preceq [\![l']\!]$ or $\mathsf{resolve}([\![l']\!],x) \neq \mathsf{resolve}([\![e]\!],x)$. This implies that $(l'.x)[e'/e.x] \equiv l'[e'/e.x].x$. For the second time we have to prove that $\mathcal{L}(l'[e'/e.x].x)(\sigma,\tau,\omega) = \mathcal{L}(l'.x)(\sigma',\tau,\omega)$. However, the assumptions are somewhat different here. Observe that from $[\![l']\!] \not\preceq [\![e]\!]$ and $[\![e]\!] \not\preceq [\![l']\!]$ it follows that the values of l' and e are different in any state, since $\mathrm{dom}([\![l']\!]) \cap \mathrm{dom}([\![e]\!]) = \emptyset$. In particular, we have $\mathcal{L}(l')(\sigma',\tau,\omega) \neq \mathcal{L}(e)(\sigma,\tau,\omega)$. Therefore both $[\![l']\!] \not\preceq [\![e]\!]$ together with $[\![e]\!] \not\preceq [\![l]\!]$ and $\mathsf{resolve}([\![l']\!],x) \neq \mathsf{resolve}([\![e]\!],x)$ imply (by construction of σ'):

$$\sigma(\mathcal{L}(l')(\sigma',\tau,\omega))(\mathsf{resolve}([\![l']\!],x))(x) = \sigma'(\mathcal{L}(l')(\sigma',\tau,\omega))(\mathsf{resolve}([\![l']\!],x))(x).$$

At this point we can refer to previous steps in the proof to finish this branch, and indeed the entire proof. □

4.2 Object creation

Next we consider the creation of objects. We want to define the substitution $[\texttt{new } C/u]$, which models the creation of a new instance of class C and its assignment to the local variable u. That is, it models logically 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 temporary variable which does not occur in P and e.

As with the usual notions of substitution we want the formula after substitution to have the same meaning before the assignment as the unmodified formula after the assignment. However, due to the creation of a new object, there are certain logical expressions for which this is not possible, because they refer to the new object, and there is no expression that has this value in the state before its creation, because it does not exist yet. Therefore the result of the substitution must be left undefined in some cases.

However, we *are* able to carry out the substitution on any boolean logical expression and therefore on any formula because a temporary variable u referring to the new object can only occur in a context where we can statically predict the value of the surrounding expression. Therefore we are able to construct an expression that has the same value in the state before the creation without having to refer to the new object.

Let us first consider the cases of the substitution which must be left undefined. Clearly, $u[\texttt{new } C/u]$ is undefined. For this reason, the substitution is also undefined for expressions of the form $\texttt{if } l_1 \texttt{ then } l_2 \texttt{ else } l_3 \texttt{ fi}$, if the substitution is undefined for l_2 or l_3. We will show in the remainder of this section that we can define the substitution on all boolean expressions, so $l_1[\texttt{new } C/u]$ is always defined. Possibly we are also unable to define the substitution $[\texttt{new } C/u]$ for expressions of the form $(C')l$. This can be the case if $l \equiv u$ or if l is a conditional expression. Actually, we can define $(C')u[\texttt{new } C/u]$ if $C \not\preceq C'$, because the value of $(C')u$ is $\bot$ in this case.

To simplify the definition of $[\texttt{new } C/u]$ we will first rewrite formulas into a normal form. Firstly, we will remove all occurrences of casts of conditional expressions by observing that the following equivalence holds.

$$\mathcal{L}((C)\texttt{if } l_1 \texttt{ then } l_2 \texttt{ else } l_3 \texttt{ fi})(\sigma,\tau,\omega) = \mathcal{L}(\texttt{if } l_1 \texttt{ then } (C)l_2 \texttt{ else } (C)l_3 \texttt{ fi})(\sigma,\tau,\omega)$$

Moreover, we have

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

We introduce the operator $\downdownarrows$ for this rewriting operation. Its only characteristic case is

$$((C)\texttt{if } l_1 \texttt{ then } l_2 \texttt{ else } l_3 \texttt{ fi}) \downdownarrows \equiv \texttt{if } l_1 \downdownarrows \texttt{ then } (C)l_2 \downdownarrows \texttt{ else } (C)l_3 \downdownarrows \texttt{ fi}.$$

Secondly, we want to 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 $C \preceq C''$. The operator $[u \boxslash C]$ performs this operation. Its only interesting case is the following.

$$(C')l[u \boxslash C] \equiv \begin{cases} (C')u & \text{if } l[u \boxslash C] \equiv (C'')u \text{ and } C \preceq C'' \\ (C')\texttt{null} & \text{if } l[u \boxslash C] \equiv (C'')u \text{ and } C \not\preceq C'' \\ (C')(l[u \boxslash C]) & \text{otherwise} \end{cases}$$

It is easy to see that, for every logical expression l, $[\![l[u \boxslash C]]\!] = [\![l]\!]$ holds and, moreover, we have $\mathcal{L}(l)(\sigma, \tau, \omega) = \mathcal{L}(l[u \boxslash C])(\sigma, \tau, \omega)$ for every state (σ, τ) that arises by executing $u = \texttt{new } C()$. We cannot replace all occurrences of $(C)u$ by u or $\texttt{null}$, because such an operation does not preserve the type of an expression.

First applying $\downdownarrows$ and $[u \boxslash C]$ makes the definition of $[\texttt{new } C/u]$ much simpler, because they ensure that $(C)l[\texttt{new } C/u]$ can be defined for every $l \not\equiv u$. The definition of $[\texttt{new } C/u]$ is straightforward in the following cases.

$$l[\texttt{new } C/u] \equiv l, \text{for } l \equiv \texttt{null}, \texttt{this}, v \text{ or } z,$$

where v is a temporary variable distinct from u, and z is a logical variable. Observe that we use the fact that u is the unique reference to the new object after its creation.

The (instance) variables of a new object have their default values after creation. These values depend on the types of the variables. We have the following cases.

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

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

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

$$(l.x)[\texttt{new } C/u] \equiv (l[\texttt{new } C/u]).x$$

The last case is only valid if none of the other cases can be applied. For the third case to be valid it is essential that l_2 and l_3 have the same type. In that case, the conditional expression also has this type. Observe that this restriction can be easily met by the inclusion of the cast operator.

Our next case involves a cast. If $l[\texttt{new } C/u]$ is defined, we have

$$(C')l[\texttt{new } C/u] \equiv (C')(l[\texttt{new } C/u]).$$

Another context in which u may occur is that of an equality $l = l'$. If both the expressions l and l' equal u we obviously have

$$(u = u)[\texttt{new } C/u] \equiv \mathsf{true}.$$

If l or l' is an expression of the form $(C')u$ we first replace it by u or $\texttt{null}$ depending on the condition $C \preceq C'$ and then apply $[\texttt{new } C/u]$ to the resulting equality. If either l is u and l' is neither u nor a conditional expression (or vice versa) we have that after the substitution operation

l and l' cannot denote the same object (because one of them refers to the new object while the other one refers to an already existing object):

$$(l = l')[\texttt{new}\ C/u] \equiv \mathsf{false}.$$

If neither l nor l' is u or a conditional expression they cannot refer to the new object and we have

$$(l = l')[\texttt{new}\ C/u] \equiv (l[\texttt{new}\ C/u]) = (l'[\texttt{new}\ C/u]).$$

For l a conditional expression of the form $\texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi}$ we define

$$(l = l')[\texttt{new}\ C/u] \equiv \texttt{if}\ l_1[\texttt{new}\ C/u]\ \texttt{then}\ (l_2 = l')[\texttt{new}\ C/u]\ \texttt{else}\ (l_3 = l')[\texttt{new}\ C/u]\ \texttt{fi},$$

and similar if l' is a conditional expression. Expressions of the form $l\ \texttt{instanceof}\ C'$ also require a case split. We distinguish the following cases.

$$(u\ \texttt{instanceof}\ C')[\texttt{new}\ C/u] \equiv \begin{cases} \mathsf{true} & \text{if } C \preceq C' \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$((C'')u\ \texttt{instanceof}\ C')[\texttt{new}\ C/u] \equiv \begin{cases} \mathsf{true} & \text{if } C \preceq C' \text{ and } C \preceq C'' \\ \mathsf{false} & \text{otherwise} \end{cases}$$

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

If none of the above cases can be applied we have

$$(l\ \texttt{instanceof}\ C')[\texttt{new}\ C/u] \equiv (l[\texttt{new}\ C/u])\ \texttt{instanceof}\ C'.$$

Finally, we have the following two cases.

$$\begin{aligned}(\texttt{op}(l_1, \ldots, l_n))[\texttt{new}\ C/u] &\equiv \texttt{op}(l_1[\texttt{new}\ C/u], \ldots, l_n[\texttt{new}\ C/u]) \\ \texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi}[\texttt{new}\ C/u] &\equiv \texttt{if}\ l_1[\texttt{new}\ C/u]\ \texttt{then}\ l_2[\texttt{new}\ C/u]\ \texttt{else}\ l_3[\texttt{new}\ C/u]\ \texttt{fi}\end{aligned}$$

Note that any operator $\texttt{op}$ is assumed to be an operator on primitive types, and secondly, that $[\texttt{new}\ C/u]$ is defined on any expression of a primitive type. The final case is only meant for conditional expressions where l_2 and l_3 have the same primitive type. We need not define this case if the conditional expression is of some reference type. The following lemma states in which cases $l[\texttt{new}\ C/u]$ is defined.

Lemma 1 The result of $l'[\texttt{new}\ C/u]$ is defined for all logical expressions $l' \equiv l \downarrow\downarrow [u \boxslash C]$ except if l' is of the form u, $(C')u$ or $\texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi}$ with $[\![l_2]\!] = [\![l_3]\!] \in \mathcal{C}$.

Proof First prove by induction on l that $l \downarrow\downarrow [u \boxslash C]$ never results in anything of the form $(C)(C')u$ or $(C)\texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi}$. Then one can prove the lemma by structural induction on l. Observe that the lemma implies that $l'[\texttt{new}\ C/u]$ is defined for all expressions $l' \equiv l \downarrow\downarrow [u \boxslash C]$ of a primitive type. □

The following lemma states that the value of $l[u \boxslash C][\texttt{new}\ C/u]$ before the creation of the new object, if defined, equals that of l after its creation.

Lemma 2 If $l'[\texttt{new}\ C/u]$ is defined, for some expression $l' \equiv l \downarrow\downarrow [u \boxslash C]$, we have

$$\mathcal{L}(l'[\texttt{new}\ C/u])(\sigma, \tau, \omega) = \mathcal{L}(l)(\sigma', \tau', \omega) \text{ and } [\![l'[\texttt{new}\ C/u]]\!] = [\![l]\!],$$

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.

Proof This lemma is proved by a straightforward induction on the complexity of l. Let us deal with one representative case: $l \equiv z.x$ (note that z is a logical variable and therefore distinct from the temporary variable u). Then $(z.x) \Downarrow [u \boxslash C][\texttt{new}\ C/u] \equiv z.x$. So we have to show that

$$\mathcal{L}(z.x)(\sigma, \tau, \omega) = \mathcal{L}(z.x)(\sigma', \tau', \omega),$$

where σ' and τ' are obtained as described above. Since $\omega(z)$ denotes an object existing in σ, we have that $\omega(z) \neq o$. It follows that $\sigma'(\omega(z))(\mathsf{resolve}([\![z]\!], x))(x) = \sigma(\omega(z))(\mathsf{resolve}([\![z]\!], x))(x)$. $\square$

Next we consider the corresponding substitution operation $[\texttt{new}\ C/u]$ on assertions. We define

$$\begin{array}{rcl} (P \wedge Q)[\texttt{new}\ C/u] & \equiv & P[\texttt{new}\ C/u] \wedge Q[\texttt{new}\ C/u] \\ (\neg P)[\texttt{new}\ C/u] & \equiv & \neg(P[\texttt{new}\ C/u]). \end{array}$$

The changing scope of a bound occurrence of a variable z ranging over objects which is induced by the creation of a new object is captured as follows.

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

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 (i.e. which already exists before the creation of the new 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 $[u/z]$ corresponds to the type-preserving substitution that is defined in the previous section. It is worthwhile to observe that we can derive the following clause for universal quantification.

$$(\forall z : C'(P))[\texttt{new}\ C/u] \equiv \begin{cases} (\forall z : C(P[\texttt{new}\ C/u])) \wedge (P[u/z] \Downarrow [u \boxslash C][\texttt{new}\ C/u]) & \text{if } C \preceq C' \\ \forall z : C'(P[\texttt{new}\ C/u]) & \text{otherwise} \end{cases}$$

As a simple example, we compute the weakest precondition of the assertion $\forall z : C(u = z \vee \texttt{this} = z)$, which states that the set of existing objects consist only of the object denoted by the temporary variable u and the active object.

$$\begin{array}{rcl} & & \Big(\forall z : C(u = z \vee \texttt{this} = z)\Big)[\texttt{new}\ C/u] \\ & \equiv & \forall z : C\Big((u = z \vee \texttt{this} = z)[\texttt{new}\ C/u]\Big) \wedge (u = (C)u \vee \texttt{this} = u)[\texttt{new}\ C/u] \\ & \equiv & \forall z : C(\mathsf{false} \vee \texttt{this} = z) \wedge (\mathsf{true} \vee \mathsf{false}) \end{array}$$

where the last assertion obviously reduces to $\forall z : C(\texttt{this} = z)$. This assertion states that `this` is the only existing object of class C, which indeed is the weakest precondition of the assertion $\forall z : C(u = z \vee \texttt{this} = z)$ with respect to $u = \texttt{new}\ C()$.

Next we consider the case of an occurrence of a bound variable z which ranges over *sequences* of objects. Recall that we assume that in the assertion language the operations on sequences are limited to $|l|$, i.e. the length of the sequence l, and $l[n]$, i.e. the operation which yields the nth element of l. So we do not have, for example, equality on sequences as a primitive operation in the assertion language. Given this assumption, let z' be a (fresh) logical variable ranging over sequences of *boolean* values. The variables z and z' together will code a sequence of objects possibly including the new object: at the places where z' yields true the value of the coded sequence is the new object. Where z' yields false the value of the coded sequence is the same as the value of z. This encoding is described by the substitution operation $[z', u/z]$, of which the main characteristic cases are:

$$\begin{array}{rcl} z[z', u/z] & & \text{is undefined} \\ (|z|)[z', u/z] & \equiv & |z| \\ (z[l])[z', u/z] & \equiv & \texttt{if}\ z'(l')\ \texttt{then}\ ([\![z[l]]\!])u\ \texttt{else}\ z(l')\ \texttt{fi},\ \text{where } l' \equiv l[z', u/z]. \end{array}$$

This substitution operation $[z', u/z]$ is defined for the remaining expressions and extended to assertions in the standard way (its application to a compound expression is defined only if its application to its constituents is defined). Given the above restriction on the kind of operations on sequences, it is easy to see that this substitution is defined for boolean expressions and assertions.

The following lemma states the correctness of this substitution operation.

Lemma 3 Fix some $C \in \mathcal{C}$. Given a configuration σ, let ω be a logical environment with $\omega(z : C^*) \in \mathrm{qdom}(C^*, \sigma)$ (a sequence of existing objects of (a subclass of) class C) and $\omega(z')$ an equally long sequence of boolean values. Let $\alpha \in \mathrm{qdom}(C^*, \sigma)$ be a sequence of objects in σ such that the sequences $\omega(z)$ and α have equal length. Furthermore, for some existing object $o = (C, n)$ in σ, we have, for all i, if $\omega(z')[i] = \mathsf{true}$ then $\alpha[i] = o$ else $\alpha[i] = \omega(z)[i]$. We have

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

($\tau\{o/u\}$ results from τ by assigning o to u).

Proof Straightforward induction on the complexity of l and P. We treat the only (slightly) non-trivial case of an expression of the form $z[l]$. Let $\omega' = \omega\{\alpha/z\}$, where α is a sequence of objects such that for all i, if $\omega(z')[i] = \mathsf{true}$ then $\alpha[i] = o$ else $\alpha[i] = \omega(z)[i]$. Moreover, let $\tau' = \tau\{o/u\}$.

By the induction hypothesis we have

$$\mathcal{L}(l[z', u/z])(\sigma, \tau', \omega) = \mathcal{L}(l)(\sigma, \tau', \omega').$$

Let $l' \equiv l[z', u/z]$. We then calculate as follows.

$$\begin{array}{ll} & \mathcal{L}((z[l])[z', u/z])(\sigma, \tau', \omega) \\ = & \mathcal{L}(\mathtt{if}\ z'(l')\ \mathtt{then}\ (\llbracket z[l] \rrbracket)u\ \mathtt{else}\ z(l')\ \mathtt{fi})(\sigma, \tau', \omega) \\ = & \text{if } \omega(z')[\mathcal{L}(l')(\sigma, \tau', \omega)] = \mathsf{true} \text{ then } o \text{ else } \omega(z)[\mathcal{L}(l')(\sigma, \tau', \omega)] \\ = & \text{if } \omega(z')[\mathcal{L}(l)(\sigma, \tau', \omega')] = \mathsf{true} \text{ then } o \text{ else } \omega(z)[\mathcal{L}(l)(\sigma, \tau', \omega')] \\ = & \omega'(z)[\mathcal{L}(l)(\sigma, \tau', \omega')] \\ = & \mathcal{L}(z[l])(\sigma, \tau', \omega'). \end{array}$$

□

Given this encoding we can now define

$$(\exists z : C'^*(P))[\mathtt{new}\ C/u] \equiv \begin{cases} \exists z\ \exists z'\ (|z| = |z'| \wedge (P[z', u/z] \downdownarrows [u \boxslash C][\mathtt{new}\ C/u])) & \text{if } C \preceq C' \\ \exists z : C'^*(P[\mathtt{new}\ C/u]) & \text{otherwise} \end{cases}$$

As an example, consider the following assertion

$$\exists z_1 : C^*\Big(|z_1| = n \wedge \forall z_2 : C(\exists i\ (z_1[i] = z_2))\Big).$$

This assertion states that there exist at most n instances of class C. We will compute the weakest precondition of this formula w.r.t. the statement $u = \mathtt{new}\ C()$. An application of the substitution $[z_1', u/z_1]$ to the assertion $|z_1| = n \wedge \forall z_2 \exists i\ (z_1[i] = z_2)$ results in the assertion

$$|z_1| = n \wedge \forall z_2 \exists i\ \Big(\mathtt{if}\ z'[i]\ \mathtt{then}\ (C)u\ \mathtt{else}\ z_1[i]\ \mathtt{fi} = z_2\Big).$$

An application of $[\mathtt{new}\ C/u]$ to this latter assertion results in the formula

$$(|z_1| = n) \wedge \forall z_2 \Big(\begin{array}{c} \exists i\ (\mathtt{if}\ z'[i] = \mathsf{true}\ \mathtt{then}\ \mathsf{false}\ \mathtt{else}\ z_1[i] = z_2\ \mathtt{fi}) \\ \wedge\ \exists i(\mathtt{if}\ z'[i] = \mathsf{true}\ \mathtt{then}\ \mathsf{true}\ \mathtt{else}\ \mathsf{false}\ \mathtt{fi}) \end{array} \Big).$$

The last clause rules out one of the n locations as a possible location of an existing object. Therefore, this assertion is logically equivalent to the assertion

$$(|z_1| = n) \wedge \forall z_2 \Big(\exists i\ (\neg z'[i] \wedge z_1[i] = z_2) \wedge \exists i(z'[i])\Big).$$

The complete weakest precondition of our example formula then is

$$\exists z_1 \exists z' \Big(|z_1| = |z'| \wedge |z_1| = n \wedge \forall z_2 \Big(\exists i \, (\neg z'[i] \wedge z_1[i] = z_2) \wedge \exists i \, (z'[i]) \Big) \Big)$$

This latter assertion clearly is logically equivalent to the assertion

$$\exists z_1 \Big(|z_1| = n - 1 \wedge \forall z_2 \exists i \, (z_1[i] = z_2) \Big)$$

which indeed corresponds with our intuition of the weakest precondition of the assertion which states that there exists at most n objects after the creation of a new object.

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

Theorem 3 We have

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

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.

Proof The proof proceeds by induction on the complexity of P. Again, we treat only the most interesting case of an assertion $\exists z : C^*(P)$, where z is a logical variable ranging over sequences of instances of C. We calculate as follows. By definition of the substitution operation $[\texttt{new}\ C/u]$ we have

$$\sigma, \tau, \omega \models (\exists z P)[\texttt{new}\ C/u] \text{ iff } \sigma, \tau, \omega \models \exists z \exists z' (|z| = |z'| \wedge P[z', u/z][\texttt{new}\ C/u]).$$

So, assuming that $\sigma, \tau, \omega \models (\exists z P)[\texttt{new}\ C/u]$, there exists a sequence $\alpha \in \mathrm{qdom}(C^*, \sigma,)$ and a sequence β of boolean values, with α and β of equal length, such that for $\omega' = \omega\{\alpha/z, \beta/z'\}$ we have

$$\sigma, \tau, \omega' \models P[z', u/z][\texttt{new}\ C/u].$$

By the induction hypothesis (measuring the complexity in terms of the number of quantifiers and propositional connectives) we next derive that

$$\sigma, \tau, \omega' \models P[z', u/z][\texttt{new}\ C/u] \text{ iff } \sigma', \tau', \omega' \models P[z', u/z],$$

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.

Let α' be a sequence of objects existing in σ' of the same length as α such that for all i, if $\beta[i] = \mathsf{true}$ then $\alpha'[i] = \tau'(u) = o$ else $\alpha'[i] = \alpha[i]$. Let $\omega'' = \omega'\{\alpha'/z\}$. It follows from lemma 3 that

$$\omega', \sigma', \tau' \models P[z', u/z] \text{ iff } \sigma', \tau', \omega'' \models P.$$

Finally, we observe that $\sigma', \tau', \omega'' \models P$ implies $\sigma', \tau', \omega \models \exists z P$ (the logical variable z' is assumed not to occur in P).

Conversely, let σ, τ and ω be such that τ and ω only involve objects existing in σ and

$$\sigma', \tau', \omega \models \exists z : C^*(P),$$

where σ' and τ' are defined as above. So there exists a sequence $\alpha \in \mathrm{qdom}(C^*, \sigma')$ such that

$$\sigma', \tau', \omega\{\alpha/z\} \models P.$$

Let β be a a sequence of boolean values, with α and β of equal length, and for all i, if $\alpha[i] = \tau'(u)$ then $\beta[i] = \mathsf{true}$ and else $\beta[i] = \mathsf{false}$. It follows that

$$\sigma', \tau', \omega'' \models P,$$

where $\omega'' = \omega\{\alpha/z, \beta/z'\}$ (the logical variable z' is assumed not to occur in P). Now let $\alpha' \in \mathrm{qdom}(C^*, \sigma')$ be a sequence of the same length as α such that $\alpha'[i] = \alpha[i]$, if $\beta[i] = \mathsf{false}$. Let $\omega' = \omega\{\alpha'/z\}$. By lemma 3 it then follows that

$$\sigma', \tau', \omega'' \models P \text{ iff } \sigma', \tau', \omega' \models P[z', u/z].$$

By the induction hypothesis we have

$$\sigma', \tau', \omega' \models P[z', u/z] \text{ iff } \sigma, \tau, \omega' \models P[z', u/z][\texttt{new}\ C/u].$$

By construction of ω' we have that

$$\sigma, \tau, \omega' \models (|z| = |z'| \wedge P[z', u/z][\texttt{new}\ C/u]).$$

We conclude that

$$\sigma, \tau, \omega \models (\exists z P)[\texttt{new}\ C/u].$$

□

4.3 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 [13].

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 from 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 overwritten 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.

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

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 reflect the context switch. The active object is the same in both contexts, but its type differs. The substitution correct 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. 4.1. 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'.

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. 4.1.

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 a local expression f generated by the following abstract syntax

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

is not affected by the execution of S by the receiver. A sequence of such expressions $\bar{f}$ can be substituted by a corresponding sequence of logical variables $\bar{z}$ of exactly the same type in the specification of the body S. Thus the rule reflects the fact that such expressions are constant during execution of the call. 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 overwrite the implementation given in class C. We denote this set by $\mathsf{overwrites}(C)(m)$. We have $C' \in \mathsf{overwrites}(C)(m)$ if C' is a proper subclasses of C with $m \in \mathsf{methods}(C')$ and there does not exists 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 $u \texttt{ instanceof } C \wedge \neg u \in \mathsf{overwrites}(C)(m)$, where the latter clause abbreviates the conjunction $\bigwedge_{C' \in \mathsf{overwrites}(C)(m)} \neg(u \texttt{ instanceof } C')$.

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.

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

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

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

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 overwritten 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

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

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. However, $\bar{B}_j$ contains only one element if F_j concerns a call to an overwritten 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 $\{\ S_i\ \texttt{return}\ e_i\ \}$ 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 $\{P'_i \wedge I_i\} S_i \{Q'_i[e_i/\texttt{return}_i]\}$. The formula $\bar{P}_j$ is the sequence containing, for $i = 1, \ldots, k_j$, the implications

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

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

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

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 determines 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 $P_j \to P'_j[(C)\texttt{this}, \bar{e}_j/\texttt{this}, \bar{u}_j][\bar{f}_j/\bar{z}_j]$ and $\bar{Q}_j$ is $Q'_j[(C)\texttt{this}/\texttt{this}][\bar{f}_j/\bar{z}_j] \to Q_j[\texttt{return}_j/y_j]$. Here, $\bar{e}_j = e^j_1, \ldots, e^j_{n_j}$ and $u_i = u_1, \ldots, u_n$.

Chapter 5

Completeness

In this chapter, we will prove (relative) completeness [3]. That is, given a finite set of class definitions, we prove that $\models \{P\}S\{Q\}$ implies $\vdash \{P\}S\{Q\}$. Our completeness proof is based on the following standard semantic definition of the strongest postcondition $SP(S, P)$ as the set of triples (σ, τ, ω) such that for some initial configuration (σ', τ') we have $\mathcal{S}(S)(\sigma', \tau') = (\sigma, \tau)$ and $\sigma', \tau', \omega \models P$.

It can be shown in a straightforward although rather tedious manner that $SP(S, P)$ is expressible in the assertion language (see [5, 14]). The main idea followed in [5] is based on a standard arithmetical encoding of computations and a logical encoding of global configurations as described in the next section.

5.1 Freezing the Initial Configuration

One of the first problems is to freeze the initial configuration which consists of the values of the instance variables of a set of existing objects and the values of the local variables of the active object. Note that the set of existing objects is not statically given. Therefore we store the existing objects of $\mathrm{dom}(C)$ 'dynamically' in a logical variable seq_C of type C^*. For storing the values of instance variables, we introduce for each instance variable x defined in a certain class C a corresponding logical variable $\Theta(C, x)$ of type $[\![x]\!]^*$. We assign values to this sequence in such a way that the value of the instance variable x of an existing object that has this variable is stored in the sequence denoted by $\Theta(C, x)$ at the position of the object in seq_C. Observe that only inhabitants of $\mathrm{qdom}(C, \sigma)$ have this instance variable. Storing the initial values of the local variables in logical variables is straightforward. We simply introduce a logical variable $\Theta(u)$ of type $[\![t]\!]$, for each local variable u.

The above encoding of a configuration can be captured logically by a finite (we assume given a finite set of class definitions) conjunction of the following assertions:

- $\forall z : C(\exists i(z = seq_C[i]))$
  which states that the sequence seq_C stores all existing instances of class C and its subclasses.
- $\forall i(seq_C[i].x = \Theta(x, C)[i])$
  which states that for each position in seq_C the value of the instance variable x of the object at that position is stored in the sequence $\Theta(C, x)$ at the same position.
- $u = \Theta(u)$
  which simply states that the value of u is stored in $\Theta(u)$.

We denote the resulting conjunction by *init*.

Given the above encoding of a configuration we next extend the mapping Θ to assertions such that it replaces all references to program variables by their logical counterparts. Since in general

we cannot statically determine the position of an object of class C in the sequence seq_C we simply introduce for every class C a function $pos_C(l)$ which is defined by the assertion

$$\forall z : C(z = seq_C[pos(z)]).$$

Note that indeed in practice assertion languages should allow the introduction of user-defined functions and predicates. We have the following main case of the transformation Θ on logical expressions l (where $C = \mathsf{resolve}([\![l]\!], x)$):

$$(l.x)\Theta \equiv \Theta(C, x)[pos_C(l\Theta)]$$

Quantification requires additional care when extending Θ to assertions because the scope of the quantifiers can be affected by object creation. Therefore we introduce the following *bounded* form of quantification:

$$\begin{array}{rcl} (\exists z : C(P))\Theta & \equiv & \exists z : C(\exists i(z = seq_C[i]) \wedge (P\Theta)) \\ (\exists z : C^*(P))\Theta & \equiv & \exists z : C^*(\forall i \exists j(z[i] = seq_C[j]) \wedge (P\Theta)) \end{array}$$

Note that the assertion $\exists i(z = seq_C[i])$ states that the object denoted by z appears in the sequence denoted by seq_C. The assertion $\forall i \exists j(z[i] = seq_C[j])$ states that all objects stored in the sequence denoted by z are stored in the sequence denoted by seq_C. We thus restrict quantification to the sequence denoted by seq_C. For quantification over primitive types or sequences of a primitive type we simply have $(\exists z P)\Theta = \exists z(P\Theta)$.

The transformation Θ is truth-preserving in the following manner.

Theorem 4 For every assertion P, and any logical environment ω and state (σ, τ) that are consistent, we have that $\sigma, \tau, \omega \models init$ implies $\sigma, \tau, \omega \models P$ iff $\omega, \sigma, \tau \models P\Theta$.

Proof By structural induction on P. □

Next we proof that $P\Theta$ is invariant over any statement. In general, correctness proofs of a statement S involve the set $M(S)$ of its method calls. Let c be some statement of the form $y = u.m(e_1, \ldots, e_n)$ or $y = \mathtt{super}.m(e_1, \ldots, e_n)$. Then $M(S)$ is defined as the smallest set which satisfies:

- $c \in M(S)$ if the call c occurs in S
- if $y = \mathtt{super}.m(e_1, \ldots, e_n) \in M(S)$ and S' is the body of the implementation of method m in class $C = \mathsf{impl}(F_{\lhd}(\mathtt{this}))(m)$ then $c \in M(S)$, for every call c that occurs in S'.
- if $y = u.m(e_1, \ldots, e_n) \in M(S)$ and S' is the body of an implementation of method m in a certain class $C \in \mathsf{impls}([\![u]\!])(m)$ then $c \in M(S)$, for every call c that occurs in S'.

Now we can prove the following theorem.

Theorem 5 For any statement S and assertion P we have $\vdash \{P\Theta\}\ S\ \{P\Theta\}$.

Proof We prove by structural induction on S that for every assertion P we have

$$\vdash \{P\Theta[z/\mathtt{this}]\}\ S\ \{P\Theta[z/\mathtt{this}]\}.$$

The case of a method call $S \equiv y = u.m(e_1, \ldots, e_n)$ is handled as follows. Let $C(S, P)$ denote the set of correctness specifications

$$\{P\Theta[z/\mathtt{this}]\} S_j \{P\Theta[z/\mathtt{this}]\},$$

with $j = 1, \ldots, r$ and $M(S) = \{S_1, \ldots, S_r\}$. It is easy to see that we can prove

$$C(S, P) \vdash \{P\Theta[z/\mathtt{this}]\}\ S'_j\ \{P\Theta[z/\mathtt{this}]\},$$

where S'_j denotes an arbitrary body of some implementation of a call $S_j \in M(S)$. By an application of the rule MRMI (observe that the substitutions which model the context switch, parameter passing, and assigning the return value, do not affect $P\Theta[z/\texttt{this}]$) we then obtain

$$\vdash \{P\Theta[z/\texttt{this}]\}\ S\ \{P\Theta[z/\texttt{this}]\}.$$

Finally, an application of a substitution rule which allows to replace z by `this` in the precondition and the postcondition finishes the proof. □

5.2 The Most General Correctness Formula

Our definition of the most general correctness formulas $\{init\}S\{SP(S, init)\}$ leads to the following main theorem.

Theorem 6 For every valid correctness formula $\{P\}S\{Q\}$ we have

$$F_1, \ldots, F_r \vdash \{P\}S\{Q\},$$

where $F_j = \{init\}S_j\{SP(S_j, init)\}$, for $j = 1, \ldots, r$, and $M(S) \subseteq \{S_1, \ldots, S_r\}$.

The proof proceeds by structural induction on S. The following lemma describes the most interesting case of a method call.

Lemma 4 For every call $S \equiv y = u.m(e_1, \ldots, e_n)$ we have

$$\models \{P\}S\{Q\} \text{ implies } \{init\}S\{SP(S, init)\} \vdash \{P\}S\{Q\}$$

Proof For technical convenience only we assume that P and Q do not contain free occurrences of the logical variables $\Theta(u)$ and $\Theta(x)$ (otherwise we first have to rename them). By Theorem 5 we have $\vdash \{P\Theta\}S\{P\Theta\}$. An application of the conjunction rule gives us the correctness formula

$$\vdash \{P\Theta \wedge init\}S\{P\Theta \wedge SP(S, init)\}.$$

Our next step is to prove

$$\models P\Theta \wedge SP(S, init) \rightarrow Q.$$

Let $\sigma, \tau, \omega \models P\Theta \wedge SP(S, init)$. By the definition of SP there exist an initial state (σ_0, τ_0) such that $\sigma_0, \tau_0, \omega \models init$ and $\mathcal{S}(S)(\sigma_0, \tau_0) = (\sigma, \tau)$. Since $\models \{P\Theta\}S\{P\Theta\}$ (this follows from Theorem 5 and the soundness of the proof system) we derive that $\sigma_0, \tau_0, \omega \models P\Theta$. By Theorem 4 we arrive at $\sigma_0, \tau_0, \omega \models P$. Since $\models \{P\}S\{Q\}$ we conclude that $\sigma, \tau, \omega \models Q$.

So we can proceed by an application of the consequence rule and obtain

$$\vdash \{P\Theta \wedge init\}S\{Q\}.$$

Next we can apply the standard rules which allow one to replace in the precondition every logical variable $\Theta(u)$ by the local variable u and existentially quantifying all the logical variables $\Theta(C, x)$ (x an arbitrary instance variable defined in class C). It is not difficult to prove that the assertion P logically implies the resulting precondition. Therefore an application of the consequence rule finishes the proof. □

5.3 The Context Switch

In this section, we prove the derivability of $\{init\}S\{SP(S, init)\}$, for an arbitrary method call $S \equiv y = u.m(e_1, \ldots, e_n)$. The derivability of the same specification for an arbitrary method call of the form $y = \texttt{super}.m(e_1, \ldots, e_n)$ is an obvious instantiation of the following proof. To obtain $\vdash \{init\}S\{SP(S, init)\}$ we will instantiate the rule for reasoning about mutually recursive method calls (MRMI), as defined in Section 4.3. The correctness formulas $F_1, \ldots, F_r$ in this rule will state that the set $M(S) = \{S_1, \ldots, S_r\}$ satisfies the MGF. Note that $M(S)$, in general, contains both calls to overwritten methods in superclasses and 'normal' method calls. For the sake of simplicity, we will first assume that each S_j, for $j = 1, \ldots, r$, is of the form $y_j = u_j.m_j(e^j_1, \ldots, e^j_{n_j})$. At the end, we will sketch how steps in the proof should be adapted if some S_j is a call to an overwritten method.

We use the following notational conventions in this section. Let $M(S) = \{S_1, \ldots, S_r\}$, with $S_j \equiv y_j = u_j.m_j(e^j_1, \ldots, e^j_{n_j})$, for $j = 1, \ldots, r$. Furthermore, let F_j denote the correctness formula $\{init\}S_j\{SP(S_j, init)\}$. We assume that Bs_j is defined as stated in the explanation of rule (MRMI). That is, BS_j contains a correctness formula for every implementation of the call in F_j. Which pre- and postcondition should be chosen is discussed later in this section. If $\mathsf{impls}([\![u_j]\!])(m_j) = \{C^j_1, \ldots, C^j_{k_j}\}$ then Bs_j contains k_j correctness formulas. Let $\{\ S^i_j\ \texttt{return}\ e^i_j\ \}$ be the implementation of the call S_j in class C^i_j, for $i = 1, \ldots, k_j$. Finally, we assume that u^i_j is the sequence of formal parameters of the implementation in class C^i_j.

So our goal is to prove $\vdash \{init\}S\{SP(S, init)\}$. By Theorem 6 and the rule (MRMI) it suffices to find, for $j = 1, \ldots, r$ and $i = 1, \ldots k_j$, valid correctness formulas

$$\{P^i_j \wedge I^i_j\}\ S^i_j\ \{Q^i_j[e^i_j/\texttt{return}^i_j]\}, \tag{5.1}$$

such that

$$\models init \wedge u_j\ \texttt{instanceof}\ C^i_j \wedge \neg u_j \in \mathsf{overwrites}(C^i_j)(m_j) \rightarrow P^i_j[(C^i_j)u_j, \bar{e}_j/\texttt{this}, \bar{u}^i_j][\bar{f}_j/\bar{z}^i_j] \tag{5.2}$$

and

$$\models Q^i_j[(C^i_j)u_j/\texttt{this}][\bar{f}_j/\bar{z}^i_j] \rightarrow SP(S_j, init)[\texttt{return}^i_j/y_j] \tag{5.3}$$

for some substitutions $[\bar{f}_j/\bar{z}^i_j]$ involving local expressions $\bar{f}_j$ and corresponding logical variables satisfying the conditions of rule (MRMI). The assertion I^i_j specifies the initial values of the local variables, excluding the formal parameters, as explained in the description of this rule.

To obtain the assertions P^i_j and Q^i_j, for $j = 1, \ldots, r$ and $i = 1, \ldots k_j$, we introduce renaming functions Φ_j which transforms `this` and any local variable u into (new) logical variables $\Phi_j(\texttt{this})$ and $\Phi_j(u)$ of exactly the same type. We introduce for every $j = 1, \ldots, r$, such a transformation Φ_j because the object denoted by `this` may belong to different classes in the context of a method call and therefore we need different logical variables to represent `this`. Applying Φ_j to any assertion (using Φ_j as a substitution) will *neutralize* the substitutions which model the context switch and the passing of the actual parameters. That is,

$$P\Phi_j[(C)u, e_1, \ldots, e_n/\texttt{this}, u_1, \ldots, u_n] \text{ equals } P\Phi_j,$$

for every assertion P, local variables $u, u_1, \ldots, u_n$ and expressions $e_1, \ldots, e_n$. So we can define P^i_j simply by $init\Phi_j$ and Q^i_j simply by $SP(S_j, init)[\texttt{return}^i_j/y_j]\Phi_j$. Using the inverse $(\Phi_j)^{-1}$ of Φ_j for $[\bar{f}_j/\bar{z}^i_j]$, we trivially obtain (5.2) and (5.3).

However, to prove the validity of the correctness formulas in (5.1) we have to strengthen P_j with additional information about the callee and the actual parameters specified by the call S_j. This information is given by the conjunction of the clauses $\texttt{this} = \Phi_j(u_j)$, $\texttt{this}\ \texttt{instanceof}\ C^i_j$, $\neg\texttt{this} \in \mathsf{overwrites}(m_j)(C^i_j)$ and $u^i_j = (e^i_j\Phi_j)$, $i = 1, \ldots, n_j$. Observe that (5.2) still holds because

$$(\texttt{this} = \Phi_j(u_j))[u_j, \bar{e}^i_j/\texttt{this}, \bar{u}^i_j]\Phi^{-1}_i \text{ gives rise to } u_j = u_j$$

and

$$(u_j^i = (e_j^i \Phi_j))[u_j, \bar{e}_j^i/\texttt{this}, \bar{u}_j^i]\Phi_j^{-1} \text{ yields } e_j^i = e_j^i.$$

The other clauses yield $(C_j^i)u_j$ `instanceof` C_j^i and $\neg(C_j^i)u_j \in \mathsf{overwrites}(m_j)(C_j^i)$, respectively. These clauses also trivially follow from the precondition of the call.

To prove the validity of the correctness formula

$$\{P_j^i \wedge I_j^i\}\ S_j^i\ \{Q_j^i[e_j^i/\texttt{return}_j^i]\},$$

assume that $\sigma, \tau, \omega \models P_j^i \wedge I_j^i$ and $\mathcal{S}(S_j^i)(\sigma, \tau) = (\sigma', \tau')$. We can extend this computation of the callee to one of the caller as follows. We define the initial local context τ_0 of the caller by setting the active object to $\omega(\Phi_j(\texttt{this}))$ and assigning $\omega(\Phi_j(u))$ to every local variable u. Since $\sigma, \tau, \omega \models P_j^i$ it follows that $\sigma, \tau_0, \omega \models \mathit{init}$. Furthermore we have that $\tau_0(u_j) = \tau(\texttt{this})$, i.e., the callee is indeed called by the caller, and the value of the actual parameter e_i^j, with $i = 1, \ldots, n_j$, in the state (σ, τ_0) of the caller equals the value of the corresponding formal parameter u_i^j in the configuration (σ, τ) of the callee. On the other hand, we define the final configuration of the caller by (σ'', τ_0'), where *either* $\tau_0' = \tau_0$ and σ'' is obtained from σ' by assigning the value of the return expression e_j in the state (σ', τ') to the *instance* variable y_j of the caller $\tau_0(\texttt{this})$ *or* σ'' equals σ' and τ_0' is obtained from τ_0 by assigning the value of the return expression e_j to the *local* variable y_j.

Observe that S_j^i is indeed the body of the implementation that is bound to the call S_j in the state (σ, τ_0), since

$$\sigma, \tau_0, \omega \models u_j \texttt{ instanceof } C_j^i \wedge \neg u_j \in \mathsf{overwrites}(C_j^i)(m_j),$$

holds by construction of τ_0 and the definition of P_j^i. Therefore it follows that $\mathcal{S}(S_j)(\sigma, \tau_0) = (\sigma'', \tau_0')$. Since $\sigma, \tau_0, \omega \models \mathit{init}$ we thus have $\sigma'', \tau_0', \omega \models \mathit{SP}(S_j, \mathit{init})$. Next let ω' be obtained from ω by assigning the result value, i.e., the value of y_j in the state (σ'', τ_0') of the caller, to the logical variable $\texttt{return}_j^i$. It follows that $\sigma'', \tau_0', \omega' \models \mathit{SP}(S_j, \mathit{init})[\texttt{return}_j^i/y_j]$. Next we observe that the truth of the assertion $\mathit{SP}(S_j, \mathit{init})[\texttt{return}_j^i/y_j]\Phi_j$ does not depend any more on the local context or the variable y_j. So we have $\sigma', \tau', \omega' \models \mathit{SP}(S_j, \mathit{init})[\texttt{return}_j^i/y_j]\Phi_j$. Finally, because of the definition of ω' we conclude

$$\sigma', \tau', \omega \models \mathit{SP}(S_j, \mathit{init})[\texttt{return}_j^i/y_j]\Phi_j[e_j^i/\texttt{return}_j^i].$$

Finally, let us consider the question what has to be changed to the above reasoning pattern if S_j is a call to an overwritten method in a superclass. That is, if S_j is of the form $y_j = \texttt{super}.m_j(e_1^j, \ldots, e_{n_j}^j)$. We again assume that F_j is the correctness formula $\{\mathit{init}\}S_j\{\mathit{SP}(S_j, \mathit{init})\}$. Furthermore, let S_j' denote the body of the corresponding implementation of $\texttt{super}.m_j(e_1^j, \ldots, e_{n_j}^j)$ in some class C and let e_j be its return expression. By $\bar{e}_j$ we again denote the sequence $e_1^j, \ldots, e_{n_j}^j$, and we assume that $\bar{u}_j$ are the formal parameters $u_1^j, \ldots, u_{n_j}^j$ of the implementation. Recall that Bs_j is a singleton set in this case.

Proceeding with the proof as above we must find assertions P_j and Q_j such that we obtain a valid correctness formula

$$\{P_j \wedge I_j\}\ S_j'\ \{Q_j[e_j/\texttt{return}_j]\}, \tag{5.4}$$

such that

$$\models \mathit{init} \rightarrow P_j[(C)\texttt{this}, \bar{e}_j/\texttt{this}, \bar{u}_j][\bar{f}_j/\bar{z}_j] \tag{5.5}$$

and

$$\models Q_j[(C)\texttt{this}/\texttt{this}][\bar{f}_j/\bar{z}_j] \rightarrow \mathit{SP}(S_j, \mathit{init})[\texttt{return}_j/y_j] \tag{5.6}$$

for some substitution $[\bar{f}_j/\bar{z}_j]$ similar to the ones above.

One can easily check in a way similar to the proof above that these conditions are met if we choose for P_j our previous definition of P_j^i without the clauses `this instanceof` C_j^i and $\neg\texttt{this} \in \mathsf{overwrites}(m_j)(C_j^i)$. The substitution $[\bar{f}_j/\bar{z}_j]$ is again the inverse of Φ_j.

Chapter 6

Conclusions

The main result of this paper is a syntax-directed Hoare logic for a language that contains all standard object-oriented features. The logic extends our work as presented in [6] by covering inheritance, subtyping and dynamic binding. We also proved that the logic is (relatively) complete.

In recent years, several Hoare logics for (sequential) fragments of object-oriented languages, notably Java, were proposed. However, the formal justification of existing Hoare logics for object-oriented languages is still under investigation. Notably the problem of completeness until now defied clear solutions.

In [15], a Hoare logic for a large sequential subset of Java is proved complete. However, this Hoare logic formalizes correctness proofs directly in terms of a semantics of the subset of Java in Isabelle/HOL. As observed by the author this results in a serious discrepancy between the abstraction level of the Hoare logic and the programming language, which makes the logic hard to use in practice and only suited for meta-theory.

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 this year.

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 [10] and concurrency in an object-oriented setting [1].

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 [7] for the formulation of such invariants.

Bibliography

[1] E. Abraham-Mumm, F.S. de Boer, W.P. de Roever, and M. Steffen. Verification for java's reentrant multithreading concept. In *Foundations of Software Science and Computation Structures (FOSSACS)*, number 2303 in LNCS, pages 5–20, 2002.

[2] Krzysztof R. Apt. Ten Years of Hoare's Logic: A Survey - Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[3] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *Siam Journal of Computing*, 7(1):70–90, February 1978.

[4] J.W. de Bakker. *Mathematical theory of program correctness*. Prentice-Hall, 1980.

[5] F.S. de Boer. *Reasoning about dynamically evolving process structures*. PhD thesis, Vrije Universiteit, 1991.

[6] F.S. de Boer and C. Pierik. Computer-aided specification and verification of annotated object-oriented programs. In Bart Jacobs and Arend Rensink, editors, *Formal Methods for Open Object-Based Distributed Systems V*, pages 163–177. Kluwer Academic Publishers, 2002.

[7] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. On a temporal logic for object-based systems. In S. F. Smith and C. L. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 305–326. Kluwer Academic Publishers, 2000.

[8] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[10] Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In Tom Maibaum, editor, *FASE 2000*, volume 1783 of *LNCS*, pages 284–303. Springer, 2000.

[11] Tomasz Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7:357–360, 1977.

[12] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Department of Computer Science, Iowa State University, April 2003.

[13] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *LNCS*, pages 162–176. Springer, 1999.

[14] J.V. Tucker and J.I Zucker. *Program correctness over abstract data types with error-state semantics*. North-Holland, 1988.

[15] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.