A Rule of Adaptation for OO

Cees Pierik

Frank S. de Boer

institute of information and computing sciences,
utrecht university

technical report UU-CS-2003-032
version 3.0 (23 November 2004)

www.cs.uu.nl

A Rule of Adaptation for OO

Cees Pierik[1] and Frank S. de Boer[1,2,3]

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

Abstract. This paper presents a new rule for reasoning about method calls in object-oriented programs. It concerns an optimized adaptation of Hoare's rule of adaptation to the object-oriented paradigm. The new rule contributes in various ways to the modularity of the specification. We also argue that our rule of adaptation is the missing link between Hoare logics and proof outlines for object-oriented programs.

1 Introduction

It is often argued that encapsulation is one of the strong features of the object-oriented paradigm. Encapsulation is obtained by forbidding direct access to the internal state of other objects. Thus one can only query or alter the state of an object by invoking a method. Therefore method calls are the main computational mechanism in object-oriented programs.

Different calls to a particular method may occur in completely different contexts. Preferably, a method has one fixed pre/post specification that acts as a contract between the designer of the method and its clients. This contract should enable a client to predict the behavior of a call to this method. Essential for this scenario is that the reasoning rule for method calls is able to adapt the method specification to the required specification of the call. This can be achieved for procedural languages with global variables by means of the rule of adaptation. This rule was introduced by Hoare [1], and improved by Olderog [2].

The above mentioned rules suffice in a context where only simple global variables occur. However, the set of variables of modern object-oriented programs is more complex since each existing object has its own internal state. Moreover, the state can be *extended* during the execution of a method by object creation. A rule of adaptation for object-oriented programs has to take such state extensions into account.

The main contribution of this paper is a rule of adaptation for object-oriented programs. The rule enables in various ways a modular specification of object-oriented programs. For example, it results in an important separation of concerns for local variables in proof outlines of object-oriented programs.

The rule that we present in this paper integrates frame conditions and information about the objects that a method creates in a natural way in the programming logic. These optimizations of the rule also contribute to the modularity of the specification. By integrating frame conditions we ensure that the specification of the call only needs to specify its effect on the fields that are actually modified in the implementation of the method. Thus one does not have to revise the specification whenever a field is added to a class. In a similar way the rule ensures that the specification only has to reflect the creation of new objects of a class if the implementation actually allocates new object of that such.

Our adaptation rule has been integrated in a tool that computes the verification conditions of proof outlines for a sequential subset of Java. The verification conditions are passed to a theorem prover. The tool is a successor of the tool described in [3]. The adaptation rule is particularly suitable for proof outline logics since it describes the verification condition that results from the specification of a call and the corresponding method. Thus it enables a shift from Hoare logics (where Hoare triples are embedded and manipulated in a theorem prover) to proof outlines for object-oriented languages.

This paper is organized as follows. In Sect. 2 we discuss related work on proof methods for object-oriented programming. Section 3 introduces a small sequential subset of Java that is used to illustrate the context of the adaptation rule. In Sect. 4 we describe the assertion language for object-oriented programs to which the new rule is tailored. Section 5 defines the *static effect* of a method. This information will be used to optimize the rule of adaptation in Sect. 6. The following section contains an example. In Sect. 8 we prove that the new rule preserves the relative completeness result in [4]. The paper ends with conclusions and some remarks about future work.

2 Related Work

The past decade has seen a large interest in proof methods for object-oriented programming. Several Hoare logics for sequential object-oriented languages have been proposed [5–9].

Kleymann [10] has proposed a new rule of consequence for sequential programs. It enables one to adapt the values of logical variables in specifications. The outline of the rule resembles the outline of the rule that we propose. However, it solves none of the typical object-oriented problems like the state extensions that are due to object creation.

One particular advantage of the adaptation rule we present in this paper is its treatment of local variables. The values of the local variables of the caller are not changed during a method invocation. A common technique that is used to reflect this fact is to allow the use of a substitution rule (see, e.g., [5, 8, 9]). Such a rule replaces a logical variable by a local variable in both the precondition and the postcondition of a method call:

$$\frac{\{P\}\ \mathsf{call}\ \{Q\}}{\{P[u/z]\}\ \mathsf{call}\ \{Q[u/z]\}}\ .$$

In this paper we use the convention that u always denotes a local variable. In the above rule z is a logical variable. Applying this rule is the only way to prove, for example, that the value of a local variable of a caller does not change.

Example 1. Let $m()\{$ `skip` $\}$ be the declaration of method m with body `skip`. Obviously, the Hoare triple $\{u = 1\}\ m();\ \{u = 1\}$ is valid. To prove this we must first derive $\{z = 1\}$ `skip` $\{z = 1\}$. A rule for reasoning about method calls then typically lets us infer $\{z = 1\}\ m();\ \{z = 1\}$. An application of the substitution rule with $[u/z]$ then yields the desired specification. Observe that the introduction of the logical variable z is necessary because the local variable u is out of scope in the body of the method.

The above example reveals an important drawback of this technique. It turns out that to prove a property of the local variables of the caller one has to express this property in the proof outline of the procedure in terms of logical variables. This clearly violates the modularity of the method specification. Our rule of adaptation provides a means to prove such facts by expressing it in the proof outline of the caller only.

3 Object-Oriented Programs

The rule of adaptation that we propose in this paper is suited for reasoning about method calls in object-oriented programs. An adaptation rule has to take into account all changes to the program state that can result from execution of a method. For this reason it cannot be stated independent of the rest of the programming language. Therefore we outline in this section a sequential subset of Java to which our adaptation rule will be tailored. The chosen subset illustrates the main issues in object-oriented languages. At the same time it leaves out some features that would merely complicate the definitions.

3.1 The Syntax

The syntax of the programming language is summarized in Fig. 1. A program π consists of a set of classes. The declaration of a class specifies the fields (or instance variables) of the class (denoted by the sequence $\bar{x}$), and a set of methods. A clause `extends` c' indicates that the class is an extension of another class c'. In that case, the class is said to be a subclass of class c'. We assume that a class extends the root class Object if the clause is omitted. A class inherits all fields and methods of its superclass. We impose no restrictions concerning the fields or methods that a class declares in relation to the fields and methods that it inherits. That is, we allow both field shadowing and method overriding.

A method m specifies a sequence of formal parameters $u_1, \ldots, u_n$, a sequence of additional local variables $\bar{v}$, a statement S and the return value e. For brevity, we leave the return type of the method implicit. The formal parameters and the sequence $\bar{v}$ make up the local variables of the method. The scope of a local variable is the method to which is belongs. We use u as a typical element of the

$$
\begin{array}{rcl}
\texttt{op} \in \text{Oper} & & \text{an arbitrary operator on elements of a primitive type} \\
y \in \text{Loc} & ::= & u \mid e.x \\
e \in \text{Expr} & ::= & \texttt{null} \mid \texttt{this} \mid y \mid (c)e \mid e \; \texttt{instanceof} \; c \mid \texttt{op}(e_1, \ldots, e_n) \\
S \in \text{Stat} & ::= & y \ = \ e \ ; \mid S \; S \mid u \ = \texttt{new} \; c() \ ; \mid u \ = e_0.m(e_1, \ldots, e_n) \ ; \\
 & & \mid \texttt{if} \; (e) \; \{ \; S \; \} \; \texttt{else} \; \{ \; S \; \} \mid \texttt{while} \; (e) \; \{ \; S \; \} \\
\mathit{meth} \in \text{Meth} & ::= & m(u_1, \ldots, u_n) \; \{ \; \bar{v} \; S \; \texttt{return} \; e \; \} \\
\mathit{class} \in \text{Class} & ::= & \texttt{class} \; c \; (\epsilon \mid \texttt{extends} \; c) \; \{ \; \bar{x} \; \mathit{meth}^* \} \\
\pi \in \text{Prog} & ::= & \mathit{class}^*
\end{array}
$$

Fig. 1. The syntax of the programming language

set of local variables of a method. It denotes either a formal parameter or an additional local variable from $\bar{v}$.

Assignments are divided in two kinds. Assignments to local variables have the form $u \ = \ e$, where e denotes an expression that has no side effects. Thus this kind of assignments has no permanent effect on the program state (the values of local variables are discarded when the method terminates). Assignments to instance variables have the form $e.x \ = \ e'$. Execution of such a statement results in the assignment of the value of e' to field x of object e.

A statement $u \ = \ \texttt{new} \; c()$ involves the creation of a new object of class c. A reference to this new object is assigned to the local variable u. Method invocations are denoted by $e_0.m(e_1, \ldots, e_n)$. Here e_0 is the object of which method m is invoked. The expressions $e_1, \ldots, e_n$ are the actual parameters of the invocation.

The expressions listed are a minimal subset that suffices for our present purposes. The `instanceof` operator is used to handle dynamic binding. An expression e `instanceof` c is true if e denotes an instance of (some subclass of) class c. Casts of the form $(c)e$ can be used to cope with field shadowing [9]. They change the type of the expression to c.

Finally some words on types. We only consider two primitive types in this paper: `int` and `boolean`. Variables in the program may also have a reference type. In that case the type of the variable is one of the classes declared in the program. We will tacitly assume that all programs are well-typed. We refer to the type of an expression e by $[\![e]\!]$. The variable t ranges over the set of types.

3.2 Structural Operational Semantics

In this subsection we present a simple structural operational semantics for the statements in the languages. The formal semantics of method calls is, however, reserved for the next section. The semantics should be helpful for a proper understanding of the rest of this paper. Moreover, it is an essential requirement for the soundness proof of the adaptation rule.

We start our description with some auxiliary definitions for a program π. The program π determines among others the subclass relation. By $c \preceq c'$ we denote that c is a subclass of c'. This relation is reflexive and transitive. We assume that $F_{\lhd}(c)$ denotes the direct superclass of a class c. It is undefined if c has no superclass. We say that c' is a *proper* subclass of c if $c' \preceq c$ and $c' \neq c$.

Let Var denote a set of variables. We assume that at least all local variables that are used in π and the special-purpose variable `this` are elements of Var. Let IVar_c denote the set of instance variables declared in class c. Due to inheritance an object can have several fields with the same identifier. An expression $e.x$ always corresponds to the first declaration of a instance variable x as found by an upward search that starts in class $[\![e]\!]$. The upward search is formalized by the function origin, which is declared as follows.

$$\mathsf{origin}(c, x) = \begin{cases} c & \text{if } x \in \mathrm{IVar}_c \\ \mathsf{origin}(F_{\lhd}(C), x) & \text{otherwise} \end{cases}$$

We represent objects as follows. Each object has its own identity and belongs to a certain class. Let $\mathcal{C}$ be the set of classes declared in π. For each class $c \in \mathcal{C}$ we introduce the infinite set $O^c = \{c\} \times \mathbb{N}$ of objects of class c (here $\mathbb{N}$ denotes the set of natural numbers).

The domain of a variable of type t is denoted by $\mathrm{dom}(t)$. The domains $\mathrm{dom}(\texttt{boolean})$ and $\mathrm{dom}(\texttt{int})$ are the set of boolean and integer values, respectively. Due to subtyping a variable of some reference type c can also refer to an object of some subclass of c. Let $\texttt{subs}(c)$ be the set of all subclasses of class c. Then $\mathrm{dom}(c)$ is the set $(\bigcup_{c' \in \texttt{subs}(c)} O^{c'}) \cup \{\nu\}$. Here ν is the value of `null`.

A state (σ, τ) of a program π consists of a heap σ and an environment τ. An environment $\tau \in \mathrm{T}$ assigns values to the local variables. Formally, T is the set $\prod_{z \in \mathrm{Var}} \mathrm{dom}([\![z]\!])$. Note that by $\prod_{a \in A}(P(a))$ we mean a (generalized) cartesian product. An element of this set is a function that assigns to every element $a \in A$ an element of the set $P(a)$.

A heap consists of the existing objects. Each object in turn has its own internal state (the values of its instance variables). The internal state of an object $o \in O^c$ is a total function that maps the instance variables of class c and its superclass to their values. Let $\texttt{supers}(c)$ be the set $\{c' \in \mathcal{C} | c \preceq c'\}$. The internal state of an instance of class c is an element of the set $\mathrm{internal}(c)$, which is the set

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

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

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

In the sequel, we will write $\sigma(o)$ for some object $o = (c, n)$ as shorthand for $\sigma(c)(n)$. In this way, $\sigma(o)$ denotes the internal state of an object. Note that $\sigma(c)$ is not defined for objects that do not exist in a particular heap σ. Thus σ specifies the set of existing objects. We will only consider states that are *consistent.* A heap is consistent if all instance variable of existing objects refer to existing objects or ν. Similarly, an environment is consistent with a heap if all variables refer to existing objects or ν.

The state of a program is then simply a pair (σ, τ) that consists of a consistent heap σ, and an environment τ that is consistent with the heap σ.

Expressions are evaluated relative to a program π, and a state (σ, τ). The result of the evaluation of an expression e is denoted by $\mathcal{E}(e)(\sigma, \tau)$. We leave the program π implicit. The definition of $\mathcal{E}(e)(\sigma, \tau)$ can be found in Figure 3.2.

$$
\begin{aligned}
\mathcal{E}(\texttt{null})(\sigma,\tau) &= \nu \\
\mathcal{E}(\texttt{this})(\sigma,\tau) &= \tau(\texttt{this}) \\
\mathcal{E}(u)(\sigma,\tau) &= \tau(u) \\
\mathcal{E}(e.x)(\sigma,\tau) &= \begin{cases} \bot & \text{if } \mathcal{E}(e)(\sigma,\tau) \in \{\nu, \bot\} \\ \sigma(\mathcal{E}(e)(\sigma,\tau))(\mathsf{origin}([\![e]\!], x))(x) & \text{otherwise} \end{cases} \\
\mathcal{E}(\ (c)e\ )(\sigma,\tau) &= \begin{cases} \bot & \text{if } \mathcal{E}(e)(\sigma,\tau) = (c', n) \text{ and } c' \not\preceq c \\ \mathcal{E}(e)(\sigma,\tau) & \text{otherwise} \end{cases} \\
\mathcal{E}(e \ \texttt{instanceof}\ c)(\sigma,\tau) &= \begin{cases} \bot & \text{if } \mathcal{E}(e)(\sigma,\tau) = \bot \\ \mathsf{false} & \text{if } \mathcal{E}(e)(\sigma,\tau) = \nu \\ c' \preceq c & \text{if } \mathcal{E}(e)(\sigma,\tau) = (c', n) \end{cases} \\
\mathcal{E}(\mathsf{op}(e_1, \ldots, e_n))(\sigma,\tau) &= \begin{cases} \bot & \text{if } \mathcal{E}(e_i)(\sigma,\tau) = \bot \\ \bar{\mathsf{op}}(\mathcal{E}(e_1)(\sigma,\tau), \ldots, \mathcal{E}(e_n)(\sigma,\tau)) & \text{otherwise,} \end{cases} \\
& \text{where } \bar{\mathsf{op}} \text{ denotes the fixed interpretation of } \mathsf{op}.
\end{aligned}
$$

Fig. 2. Semantics of Expressions

The presented language is not very large. So it should be possible to give a succinct formal semantics to the statements. We will try not to become overly formal about simple things. Therefore we start with a list of properties that we will merely assume (without presenting a formal system to derive them).

Programs should be unambiguous and acyclic. The first conditions means that classes, fields and methods should be unique. More precisely, we require that no two classes have the same identifier, and that fields and methods are unique in their defining class. In particular, we do not allow overloading of methods. A class can only declare a method that it also inherits if the new method has the same argument types and return type as the overridden method. This latter requirement is not essential but simplifies the semantics. A program is acyclic if the subclass hierarchy is acyclic. Finally, we assume that the program is well-typed.

By $\langle S, (\sigma, \tau)\rangle \rightarrow (\sigma', \tau')$ we denote that a computation of S that starts in a state (σ, τ) ends in a state (σ', τ').

We will discuss some interesting cases in detail. We start with an assignment to a local variable. Such an assignment only succeeds if the value of the expression

e is defined.

$$\frac{\begin{array}{c}\mathcal{E}(e)(\sigma,\tau) \neq \bot \\ \tau' = \tau[u \mapsto \mathcal{E}(e)(\sigma,\tau)]\end{array}}{\langle u \ = \ e \ ; \ ,(\sigma,\tau)\rangle \to (\sigma,\tau')} \tag{L}$$

Successful termination of an assignment $e.x \ = \ e'$ requires that the values of both expressions are defined. Moreover, e should denote an object. The rule assumes that field x is defined in class c'. Then the new heap σ' is obtained from σ by assigning the value of e' to field x of class c' of this object. The local state remains unchanged.

$$\frac{\begin{array}{c}\mathcal{E}(e)(\sigma,\tau) = (c,n) \\ \mathcal{E}(e')(\sigma,\tau) \neq \bot \\ \mathsf{origin}([\![e]\!],x) = c' \\ \sigma' = \sigma[(c)(n)(c')(x) \mapsto \mathcal{E}(e')(\sigma,\tau)]\end{array}}{\langle e.x \ = \ e' \ ; \ ,(\sigma,\tau)\rangle \to (\sigma',\tau)} \tag{I}$$

An assignment $u \ = \ \texttt{new}\, c()$ involves the creation of a new object (an object that does not exists in the heap σ). All instance variables of the new objects are initialized with their default values. The default value of a variable depends on its type. We assume that $\mathsf{def}(t)$ denotes the default value of a type t. We have $\mathsf{def}(c) = \nu$ for every reference type c.

$$\frac{\begin{array}{c}\sigma(c)(n) \text{ is undefined} \\ \forall c' \in \texttt{supers}(c) \forall x \in \mathrm{IVar}_{c'} (f(c')(x) = \mathsf{def}([\![x]\!])) \\ \sigma' = \sigma[(c)(n) \mapsto f] \\ \tau' = \tau[u \mapsto (c,n)]\end{array}}{\langle u \ = \ \texttt{new}\, c() \ ; \ ,(\sigma,\tau)\rangle \to (\sigma',\tau')} \tag{C}$$

The other cases are straightforward. They are listed in Figure 3.2

3.3 Method calls

Our rule of adaptation is designed for reasoning about method calls in object-oriented languages. In this section we discuss the structural operational semantics of such method calls.

Recall that a call to method m of object e_0 is of the form $u = e_0.m(e_1,\ldots,e_n)$. We assume that all methods are public. This implies that we have to deal with dynamic binding if a method is overridden in a subclass. However, we will first describe the case of a method that is not overridden. This implies that we can simply lookup the implementation of method m in class $[\![e_0]\!]$ (we ignore method overloading). Let this implementation be

$$m(u_1,\ldots,u_n)\{ \ \bar{v} \ S \ \texttt{return} \ e \ \} \ .$$

The call starts with the context switch in which the value of e_0 is assigned to `this` and the values of the actual parameters $\bar{e} = e_1,\ldots,e_n$ are assigned to the

$$\frac{\begin{array}{c}\langle S_1, (\sigma, \tau)\rangle \to (\sigma'', \tau'') \\ \langle S_2, (\sigma'', \tau'')\rangle \to (\sigma', \tau')\end{array}}{\langle S_1\ S_2, (\sigma, \tau)\rangle \to (\sigma', \tau')} \qquad \text{(S)}$$

$$\frac{\begin{array}{c}\mathcal{E}(e)(\sigma, \tau) = \mathsf{true} \\ \langle S_1, (\sigma, \tau)\rangle \to (\sigma', \tau')\end{array}}{\langle \texttt{if}\ (e)\ \{\ S_1\ \}\ \texttt{else}\ \{\ S_2\ \}\ , (\sigma, \tau)\rangle \to (\sigma', \tau')} \qquad \text{(IF1)}$$

$$\frac{\begin{array}{c}\mathcal{E}(e)(\sigma, \tau) = \mathsf{false} \\ \langle S_2, (\sigma, \tau)\rangle \to (\sigma', \tau')\end{array}}{\langle \texttt{if}\ (e)\ \{\ S_1\ \}\ \texttt{else}\ \{\ S_2\ \}\ , (\sigma, \tau)\rangle \to (\sigma', \tau')} \qquad \text{(IF2)}$$

$$\frac{\begin{array}{c}\mathcal{E}(e)(\sigma, \tau) = \mathsf{true} \\ \langle S, (\sigma, \tau)\rangle \to (\sigma'', \tau'') \\ \langle \texttt{while}\ (e)\ \{\ S\ \}, (\sigma'', \tau'')\rangle \to (\sigma', \tau')\end{array}}{\langle \texttt{while}\ (e)\ \{\ S\ \}, (\sigma, \tau)\rangle \to (\sigma', \tau')} \qquad \text{(W1)}$$

$$\frac{\mathcal{E}(e)(\sigma, \tau) = \mathsf{false}}{\langle \texttt{while}\ (e)\ \{\ S\ \}, (\sigma, \tau)\rangle \to (\sigma, \tau)} \qquad \text{(W2)}$$

Fig. 3. Structural operational semantics: additional rules

formal parameters $\bar{u} = u_1, \ldots, u_n$. The additional local variables $\bar{v}$ initially have their default value. After execution of the body the value of e is assigned to u. This corresponds to the following rule.

$$\frac{\begin{array}{c}\langle S, (\sigma, \tau_i)\rangle \to (\sigma', \tau_i') \\ \tau_i = \tau[\texttt{this}, \bar{u}, \bar{v} \mapsto \mathcal{E}(e_0)(\sigma, \tau), \mathcal{E}(\bar{e})(\sigma, \tau), \mathsf{def}(\llbracket \bar{v} \rrbracket)] \\ \tau' = \tau[u \mapsto \mathcal{E}(e)(\sigma', \tau_i')]\end{array}}{\langle u = e_0.m(e_1, \ldots, e_n), (\sigma, \tau)\rangle \to (\sigma', \tau')}$$

Observe that the environment τ' only differs from τ in the value that is assigned to u. All other local variables are not changed by the method call. By contrast, the fields of the objects in σ may have different values in σ'. Moreover, the heap σ' possibly contains objects that did not exist in σ.

Evaluation of calls to public methods requires an evaluation of e_0 to decide to which implementation the call is bound. Let $\mathcal{E}(e_0)(\sigma, \tau) = (c, n)$. Then this call is bound to the implementation of method m that is found in class c or otherwise the first implementation of this method in a superclass of c. We denote this method implementation by $\mathsf{exec\_impl}(c, m)$. In all other aspects the evaluation of the call is similar to the rule above.

$$\frac{\begin{array}{c}\mathcal{E}(e_0)(\sigma,\tau) = (c,n)\\ \mathsf{exec\_impl}(c,m) = m(u_1,\ldots,u_n)\{\ \bar{v}\ S\ \texttt{return}\ e\ \}\\ \langle S,(\sigma,\tau_i)\rangle \rightarrow (\sigma',\tau_i')\\ \tau_i = \tau[\texttt{this},\bar{u},\bar{v} \mapsto \mathcal{E}(e_0)(\sigma,\tau),\mathcal{E}(\bar{e})(\sigma,\tau),\mathsf{def}(\llbracket\bar{v}\rrbracket)]\\ \tau' = \tau[u \mapsto \mathcal{E}(e)(\sigma',\tau_i')]\end{array}}{\langle u = e_0.m(e_1,\ldots,e_n),(\sigma,\tau)\rangle \rightarrow (\sigma',\tau')} \quad \text{(D)}$$

4 The Assertion Language

The proof rule that we propose is tailored to a specific assertion language called AsO (Assertion language for Object structures). We will describe the syntax and semantics of AsO in this section. Moreover, we will explain some of the design decisions behind the language.

Any Hoare logic is tailored to a specific assertion language in which the assertions are expressed. The assertion language decides the abstraction level of the logic. An important design decision behind AsO is to keep its abstraction level as close as possible to the programming language. In other words, we refrain as much as possible from introducing constructs that do not occur in the programming language. This makes it easier for programmers to annotate their programs.

The set of expressions in AsO is obtained by extending the set of program expressions with the following clauses.

$$l \in \text{LExpr} \ ::= \ \ldots \mid z \mid l_1 = l_2 \mid \texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi} \mid l[l'] \mid l.\texttt{length}$$

The variable z denotes a logical variable. A logical variable is simply a placeholder for an arbitrary value. Logical variables are usually used to refer to the old value of a variable in the postcondition of a method. Conditional expressions like $\texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi}$ are usually also present in the programming language. We list them here explicitly because they are needed for reasoning about aliases [9]. The same remark applies to expressions of the form $l_1 = l_2$.

The only real addition are the two operations on *finite sequences*. Logical variables can have type t^* for some type t of the programming language. This means that its value is a finite sequence of elements from the domain of t. Finite sequences can be used to specify properties of object structures. For example, it enables us to express that there exists a sequence of objects in which each objects has a pointer to its successor in the sequence. In fact, finite sequences are also essential for our adaptation rule as will become clear in the rest of this paper. We write $l[l']$ to select the element at position l' in the sequence l. The length of a sequence l is denoted by $l.\texttt{length}$.

Formulas in AsO are built from expressions in the usual way.

$$P, Q \in \text{Ass} ::= l_1 = l_2 \mid \neg P \mid P \wedge Q \mid \exists z : t(P)$$

A formula $\exists z : c(P)$ means that P holds for an *existing* object of (a subclass of) class c or `null`. A formula $\exists z : c^*(P)$ indicates that P holds for a sequence of such objects. We sometimes omit the type in $\exists z : t(P)$ if it is clear from the context.

The standard abbreviations like $p \vee q$ for $\neg(\neg p \wedge \neg q)$ and $\forall z(P)$ for $\neg\exists z \neg(P)$ are valid. We also use two other useful abbreviations. A formula $z \in z'$ will stand for $\exists i(0 \leq i < z'.\mathsf{length} \wedge z = z'[i])$, and $z \subseteq z'$ abbreviates the formula $\forall i(0 \leq i < z.\mathsf{length} \rightarrow z[i] \in z')$.

Note that the validity of assertions may also be affected by the creation of new objects. We can, for example, express in the assertion language that there exist no objects of some class c by means of the formula $\forall z : c(z = \texttt{null})$. This formula clearly does not hold in the state that results from executing $u = \texttt{new}\ c()$.

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 (in examples) the value is implicitly compared to `true`. An alternative solution that is employed in JML [11] is to return an arbitrary element of the underlying domain.

4.1 Semantics

The evaluation of expressions of AsO is very similar to that of expressions of the programming language. Therefore we focus in this section on the elements of AsO that are not present in the programming language.

As described above, the most essential addition are the finite sequences. The domain of logical variables of some type t^* is straightforward. Formally, $\mathrm{dom}(t^*)$ is the set

$$\{(n, f) \mid n \in \mathbb{N} \text{ and } f \in \prod_{i=0}^{n-1} \mathrm{dom}(t)\} \ .$$

That is, we see finite sequences as a pair of a natural number that denotes the length of the sequences and a function that assigns to each valid index an element of the component type t of the sequence.

Logical expressions are evaluated relative to a program π, and a state (σ, τ). An assertion $\exists z : c(P)$ expresses 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 heap. Let $\mathrm{qdom}(t, \sigma)$ denote the quantification domain of a variable of type t in heap σ. We define $\mathrm{qdom}(c, \sigma) = \{o \in \mathrm{dom}(c) \mid \sigma(o) \text{ is defined } \} \cup \{\perp\}$. 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 $t \in \{\texttt{int}, \texttt{boolean}, \texttt{int}^*, \texttt{boolean}^*\}$.

$$
\begin{aligned}
\mathcal{L}(z)(\sigma,\tau) &= \tau(z) \\
\mathcal{L}(l_1 = l_2)(\sigma,\tau) &= \begin{cases} \mathsf{true} & \text{if } \mathcal{L}(l_1)(\sigma,\tau) = \mathcal{L}(l_2)(\sigma,\tau) \\ \mathsf{false} & \text{otherwise} \end{cases} \\
\mathcal{L}(\mathtt{if}\ l_1\ \mathtt{then}\ l_2\ \mathtt{else}\ l_3\ \mathtt{fi})(\sigma,\tau) &= \begin{cases} \bot & \text{if } \mathcal{L}(l_l)(\sigma,\tau) \in \{\bot,\nu\} \\ \mathcal{L}(l_2)(\sigma,\tau) & \text{if } \mathcal{L}(l_1)(\sigma,\tau) = \mathsf{true} \\ \mathcal{L}(l_3)(\sigma,\tau) & \text{if } \mathcal{L}(l_1)(\sigma,\tau) = \mathsf{false} \end{cases} \\
\mathcal{L}(l[l'])(\sigma,\tau) &= \begin{cases} \bot & \text{if } \mathcal{L}(l')(\sigma,\tau) = \bot \\ f(i) & \text{if } \sigma(\mathcal{L}(l)(\sigma,\tau)) = (n,f) \\ & \quad \text{and } \mathcal{L}(l')(\sigma,\tau) = i,\ 0 \le i < n \\ \bot & \text{otherwise} \end{cases} \\
\mathcal{L}(l.\mathtt{length})(\sigma,\tau) &= n, \ \text{ where } \sigma(\mathcal{L}(l)(\sigma,\tau)) = (n,f)
\end{aligned}
$$

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

Fig. 4. Semantics of AsO

The result of the evaluation of an expression l is denoted by $\mathcal{L}(l)(\sigma,\tau)$. We leave the program π implicit. Similarly, the value of an assertion P is denoted by $\mathcal{A}(P)(\sigma,\tau)$. The definition of $\mathcal{L}$ and $\mathcal{A}$ can be found in Figure 4.

The statement $\sigma,\tau \models P$ means that $\mathcal{A}(\mathrm{P})(\sigma,\tau)$ yields true. $\models P$ indicates that $\sigma,\tau \models P$ for every state (σ,τ).

5 The Static Effect of a Method

The rule of adaptation that we present in the next section integrates frame conditions in a natural way in the verification conditions of method calls. Frame conditions specify what variables are altered by a method. Thus they implicitly contain information about which variables remain unchanged by a method. We approximate the set of variables that is assigned by a method in the sense that we do not attempt to solve the aliasing problem in frame conditions. We only analyze which fields are (possibly) assigned. To which object the field belongs can be specified in the method specification. The same decision is reported [12] to result in an efficient checker of frame conditions.

Similarly, one can statically collect information about which objects are (possibly) created by a class. This information can in turn by used to guarantee that a method creates no objects of a particular class. The creation of objects can also affect the validity of assertions as explained in Sect. 4. Surprisingly, a specification language for Java such as JML [11] seems to have no clause that allows the designer of a method to express such properties. The adaptation rule in this paper reveals that such information is equally important as the well-known frame conditions.

By the *static effect* of a method we mean a pair that consists of the set of classes of which objects are possibly created by the method, and the set of fields that are possibly assigned by the method. A field is described by a pair that consists of the class in which the field is declared and its identifier. We denote effects by ψ or by its pair of constituents (cs, fs). In the rest of this section we give a formal definition of the effect of a method.

The effect of a method implementation *meth* is given by $\mathsf{sef}(meth)(\mathsf{ms})$. The second parameter is the set of methods that have already been considered. This parameter prevents a circular definition in case of recursive methods. The effect of a method implementation in some class c is the effect of its body. Let $\mathrm{meth} \equiv m(u_1, \ldots, u_n)\{\ \bar{v}\ S\ \mathtt{return}\ e\ \}$. Then

$$\mathsf{sef}(\mathrm{meth})(ms) = \begin{cases} \mathsf{sef}(S)(ms \cup \{(c, m)\}) & \text{if } (c, m) \notin ms \\ (\emptyset, \emptyset) & \text{otherwise} \end{cases} .$$

The following cases list the effects of basic statements.

$$\begin{array}{rcl} \mathsf{sef}(u\ =\ e)(ms) & = & (\emptyset, \emptyset) \\ \mathsf{sef}(e.x\ =\ e')(ms) & = & (\emptyset, \{(\mathsf{origin}([\![e]\!], x), x)\}) \\ \mathsf{sef}(S_1\ S_2)(ms) & = & \mathsf{sef}(S_1)(ms) \cup \mathsf{sef}(S_2)(ms) \\ \mathsf{sef}(u\ =\ \mathtt{new}\ c())(ms) & = & (\{c\}, \emptyset) \\ \mathsf{sef}(\mathtt{if}\ (e)\ \{\ S_1\ \}\ \mathtt{else}\ \{\ S_2\ \})(ms) & = & \mathsf{sef}(S_1)(ms) \cup \mathsf{sef}(S_2)(ms) \\ \mathsf{sef}(\mathtt{while}\ (e)\ \{\ S\ \})(ms) & = & \mathsf{sef}(S)(ms) \end{array}$$

The union of two effects is defined as follows:

$$(cs_1, fs_1) \cup (cs_2, fs_2) = (cs_1 \cup cs_2, fs_1 \cup fs_2) \ .$$

Finally, we will define $\mathsf{sef}(u\ =\ e_0.m(e_1, \ldots, e_n))(ms)$. Due to dynamic binding we cannot (in general) statically decide which implementation will be bound to this call. Therefore we consider all possibilities. We denote the set of classes that provide an implementation for this call by $\mathsf{impls}([\![e_0]\!], m)$. More precisely, $\mathsf{impls}(c, m)$ denotes the set of classes that contains

- class c if it provides an implementation of method m or otherwise the set from which c inherits the implementation of method m;
- All subclasses of class c that provide an implementation of method m.

Let $\mathsf{impl}(c, m)$ denote the implementation of method m in class c. The definition of the final clause is then as follows.

$$\mathsf{sef}(u\ =\ e_0.m(e_1, \ldots, e_n))(ms) = \bigcup\nolimits_{c \in \mathsf{impls}([\![e_0]\!], m)} (\mathsf{sef}(\mathsf{impl}(c, m))(ms))$$

As explained above, we use the effect of a method implementation to guarantee that is does not alter a certain field. In other words, if a field does not occur in the set of fields in the effect, then this field is not assigned to during the call in any object. The following lemmas links these claims to the semantics of statements.

Lemma 1. *Let* $\langle S, (\sigma, \tau)\rangle \to (\sigma', \tau')$. *Let* $\mathsf{sef}(S)(\emptyset) = (cs, fs)$. *Let* x *be in arbitrary field of some class* c *such that* $(c, x) \notin fs$. *Let* o *be an arbitrary object of (some subclass of) class* c *such that* $\sigma(o)$ *is defined. Then* $\sigma(o)(c)(x) = \sigma'(o)(c)(x)$.

Proof. By induction on the length of the derivation.

Lemma 2. *Let* $\langle S), (\sigma, \tau)\rangle \to (\sigma', \tau')$. *Let* $\mathsf{sef}(S)(\emptyset) = (cs, fs)$. *Let* c *be a class such that* $\bigwedge_{c' \in cs}(c' \not\preceq c)$. *Then* $\mathrm{qdom}(c, \sigma) = \mathrm{qdom}(c, \sigma')$.

Proof. By induction on the length of the derivation.

6 A Rule of Adaptation for OO

The Starting Point The rule of adaptation that we propose in this paper was inspired by the profound analysis by Olderog of several earlier rules of adaptation [2]. Olderog showed that the precondition of Hoare's original proposal was not the weakest possible precondition that fits in the adaptation rule. He described a way to actually *derive* a weaker precondition. His analysis also leads in a similar way to a rule of adaptation that is based on the strongest possible *postcondition*. This particular rule appears also in [13]. It has the following form.

$$\frac{\begin{array}{c}\{P'\}S\{Q'\}\\ P[\bar{y}/\bar{x}] \wedge \forall \bar{z}(P'[\bar{y}/\bar{x}] \to Q') \to Q\end{array}}{\{P\}S\{Q\}} \tag{1}$$

Here $\bar{y}$ denotes a sequence of fresh logical variables (not occurring free in P, P', Q or Q'). The sequence $\bar{x}$ contains all program variables that occur in the statement S, and $\bar{z}$ is a list of the logical variables that occur free in P' or Q'.

The logical variables in $\bar{y}$ are placeholders of the values of the program variables $\bar{x}$ in the initial state. The antecedent of the verification condition of this rule says that P holds in the initial state. Typically, this implies that P' also holds in the old state for particular values of its logical variables. For those values of the logical variables we then infer that Q' holds in the final state. And Q' must in turn imply Q.

One particular advantage of this adaptation rule is that it considers the old values of the variables (in the state preceding the call) instead of the new values after the call. This is important because of the state extensions in object-oriented programs. In the precondition variant we are forced to reason about objects that do not (yet) exist. But our assertion language only describes properties of *existing* objects. A postcondition variant of the adaptation rule only requires us to consider the objects that existed in the initial state from the perspective of the final state. That boils down to reasoning about subsets of the objects that exist in the state after the call.

Recall from Sect. 3.3 that the environment does not changes during a method call. This means that the local variables of the caller are not changed during a call. Therefore we do not have to distinguish between the old values of local variables of the caller and their new values. We only have to consider changes to the heap σ. More in particular, we have to deal with heap *modifications* and heap *extensions*.

Modelling the Old Heap To design an object-oriented variant of the above adaptation rule we have to analyze which parts of the state are modified by a method call. Recall from Sect. 3.3 that the environment of the caller (and hence every local variable of the caller) does not change during a method call. Therefore we do not have to distinguish between the old values of local variables of the caller and their new values. The heap however may change in two ways. We have to deal with both heap *modifications* and heap *extensions*.

Observe that the above rule introduces a sequence of logical variables $\bar{y}$ that represent the old values of the program variables. A first challenge for an object-oriented version of the rule of adaptation is to introduce logical variables that model the old heap. The heap comprises the existing objects and the values of their instance variables. We can model the old heap by means of a fresh logical variable μ of type Object^*. We will assume that this sequence contains the objects that existed in the old heap.

The internal state of an object consists of the values of its instance variables. For each instance variable x of some type t defined in some class c we introduce a fresh logical variable $\mu(x_c)$ of type t^*. The idea is that if an object o is stored at position i in the sequence μ then the value of $o.x$ is $\mu(x_c)[i]$. This straightforward model of the old heap presupposes that we have some way of finding the index of an object in the old heap. For this purpose we introduce a function f that yields the index in the old heap of every object.

This model of the old heap is based on the following assumptions.

- $\forall i(0 \leq i < \mu.\mathsf{length} \rightarrow \mathsf{f}(\mu(i)) = i)$,
  which states that the index function yields the index of each object in μ; It also implies that each object occurs at most once in the old heap;
- $\mu(x_c) \sqsubseteq \mu$,
  for every field x declared in some class c such that $[\![x]\!] \in \mathcal{C}$. This formula boils down to consistency of the old heap.

We denote the conjunction of these two assumptions by heap_1. This formula should be available in the theorem prover as an axiom for the verification conditions.

Bounded Quantification An important concept in our rule of adaptation is that of bounded quantification. Methods can not only modify the heap but also extend it by creating new objects. Thus a property that holds for all objects in the state before the call may not hold for all objects after the call. Therefore we sometimes have to restrict quantification to the objects that existed before the call. However, we only restrict quantification if the static effect of the method indicates that a object of this class might be created by the method.

Let μ be the sequence that models the objects in the old heap. Let the effect of the method that is called be ψ. Let $\psi = (\mathsf{cs}, \mathsf{fs})$. Then we define the bounded variant $\exists z^{\psi}_{\mu} : t(P)$ of an expression $\exists z : t(P)$ as follows.

$$\begin{aligned} \exists z^{\psi}_{\mu} : t(P) &\equiv \exists z : t(P) \text{ for } t \in \{\mathtt{int}(^*), \mathtt{boolean}(^*)\} \\ \exists z^{\psi}_{\mu} : c(P) &\equiv \begin{cases} \exists z : c(z \in \mu \wedge P) & \text{if } \bigvee_{c' \in cs}(c' \preceq c) \\ \exists z : c(P) & \text{otherwise} \end{cases} \\ \exists z^{\psi}_{\mu} : c^*(P) &\equiv \begin{cases} \exists z : c^*(z \subseteq \mu \wedge P) & \text{if } \bigvee_{c' \in cs}(c' \preceq c) \\ \exists z : c^*(P) & \text{otherwise} \end{cases} \end{aligned}$$

Note that quantification becomes bounded if the effect lists a subclass of the class that we consider. This is the right condition because the quantification domain of a class also contains the objects of subclasses.

The following lemma says that an object that exists in the final state of a call and moreovers occur in the sequence μ already existed in the initial state.

Lemma 3. *Let $\langle S, (\sigma, \tau)\rangle \to (\sigma', \tau')$. Let μ be a logical variable of sequence type* Object^*. *Then*

$$\begin{aligned} &\mathcal{A}(z \in \mu)(\sigma, \tau[z \mapsto \alpha]) = \mathsf{true} \textit{ and } \alpha \in \mathrm{qdom}(c, \sigma) \\ &\qquad\qquad\qquad \textit{if and only if} \\ &\qquad\qquad \mathcal{A}(z \in \mu)(\sigma', \tau'[z \mapsto \alpha]) = \mathsf{true} \textit{ and } \alpha \in \mathrm{qdom}(c, \sigma') \ . \end{aligned}$$

Proof. By induction on the length of the derivation.

Restricting Assertions to the Old Heap Another challenge is to find a counterpart of the substitution $[\bar{y}/\bar{x}]$. This substitution must do two things. It has to replace references to instance variables by the corresponding logical variables. And it must restrict quantification to the objects that existed in the old heap as argued above. We introduce the substitution $\downharpoonleft_{\psi}$ for this purpose. Note that this substitution takes the effect ψ of the method that is called into account. All cases of the substitution $\downharpoonleft_{\psi}$ are listed in Fig. 6.

The most interesting case of this substitution is $(l.x)\downharpoonleft_{\psi}$. It replaces $l.x$ by its value in the old heap as described above. Let $\psi = (cs, fs)$. Assume that $\mathsf{origin}([\![l]\!], x) = c$ (field x is defined in class c). Then we define this case as follows.

$$(l.x)\downharpoonleft_{\psi} \equiv \begin{cases} \mu(x_c)[\mathsf{f}(l\downharpoonleft_{\psi})] & \text{if } (c, x) \in fs \\ (l\downharpoonleft_{\psi}).x & \text{otherwise} \end{cases}$$

We only substitute the instance variable if the effect indicates that the method possibly changes its value.

$$\begin{array}{rcl}
\texttt{null}\downharpoonleft_\psi & \equiv & \texttt{null} \\
\texttt{this}\downharpoonleft_\psi & \equiv & \texttt{this} \\
z\downharpoonleft_\psi & \equiv & z \\
(l.x)\downharpoonleft_\psi & \equiv & \begin{cases} \mu(x_c)[\mathsf{f}(l\downharpoonleft_\psi)] & \text{if } (c,x)\in \mathit{fs} \\ (l\downharpoonleft_\psi).x & \text{otherwise} \end{cases} \\
(l[l'])\downharpoonleft_\psi & \equiv & (l\downharpoonleft_\psi)[l'\downharpoonleft_\psi] \\
(l.\texttt{length})\downharpoonleft_\psi & \equiv & (l\downharpoonleft_\psi).\texttt{length} \\
((c)l)\downharpoonleft_\psi & \equiv & (c)(l\downharpoonleft_\psi) \\
(l_1 = l_2)\downharpoonleft_\psi & \equiv & (l_1\downharpoonleft_\psi) = (l_2\downharpoonleft_\psi) \\
(l \texttt{ instanceof } c)\downharpoonleft_\psi & \equiv & (l\downharpoonleft_\psi) \texttt{ instanceof } c \\
\mathsf{op}(l_1,\ldots,l_n)\downharpoonleft_\psi & \equiv & \mathsf{op}(l_1\downharpoonleft_\psi,\ldots,l_n\downharpoonleft_\psi) \\
(\texttt{if } l_1 \texttt{ then } l_2 \texttt{ else } l_3 \texttt{ fi})\downharpoonleft_\psi & \equiv & \texttt{if } l_1\downharpoonleft_\psi \texttt{ then } l_2\downharpoonleft_\psi \texttt{ else } l_3\downharpoonleft_\psi \texttt{ fi} \\
\hline
(l_1 = l_2)\downharpoonleft_\psi & \equiv & (l_1\downharpoonleft_\psi) = (l_2\downharpoonleft_\psi) \\
(\neg P)\downharpoonleft_\psi & \equiv & \neg(P\downharpoonleft_\psi) \\
(P \wedge Q)\downharpoonleft_\psi & \equiv & (P\downharpoonleft_\psi) \wedge (Q\downharpoonleft_\psi) \\
(\exists z : t(P))\downharpoonleft_\psi & \equiv & \exists z^\psi_\mu : t(P\downharpoonleft_\psi))
\end{array}$$

Fig. 5. The substitution $\downharpoonleft_\psi$

Another interesting case is $(\exists z : t(P))\downharpoonleft_\psi$, which is defined as follows.

$$(\exists z : t(P))\downharpoonleft_\psi \equiv \exists z^\psi_{\bar\mu} : t(P\downharpoonleft_\psi))$$

Thus the substitution $\downharpoonleft_\psi$ restricts quantification to the objects that existed in the state before the call. The quantification is not bounded if the effect of the method guarantees that no objects of a class will be created.

The Adaptation Rule We can now state our rule of adaptation. We start with a simplified version of the rule for calls that have but one implementation. This is, for example, the case if a method is private, or if it is not overridden in some subclass. Let the statement $u = e_0.m(e_1,\ldots,e_n)$ involve a call to such a method. Let

$$\mathit{meth} \equiv m(u_1,\ldots,u_n)\{\ \bar v\ S\ \texttt{return}\ e\ \}$$

be the implementation of method m that is bound to the call. Let the effect ψ be $\mathsf{sef}(\mathit{meth})(\emptyset) = (\mathit{cs},\mathit{fs})$. Then we have the following rule of adaptation (2) for such calls.

$$\frac{\begin{array}{c}\{P'\}\ \bar v\ S\ \{Q'[e/\mathsf{result}]\} \\ \mathsf{heap}_2 \wedge P\downharpoonleft_\psi \wedge \forall \bar z^\psi_{\bar\mu}(P'[e_0,\bar e/\texttt{this},\bar u]\downharpoonleft_\psi \rightarrow \exists \bar v'(Q')) \rightarrow Q[\mathsf{result}/u]\end{array}}{\{P\}\ u = e_0.m(e_1,\ldots,e_n)\ \{Q\}} \qquad (2)$$

Observe that the rule has the same outline as the former rule except for the predicate heap_2. This assumption supplies additional information concerning the old heap. It consists of the following parts:

- $\bigwedge_{u \in U} u \in \mu$, where U is the set of all local variables that occur either in P, Q or e_i, for $i \in \{0 \ldots n\}$, and
- $\bigwedge_{i=0}^{n} e_i \downharpoonleft_{\psi} \in \mu$.

The first clause states that the local variables of the caller reference objects in the old heap; the second clause says the same about the actual parameters of the call. Local variables and parameters of a primitive type must be omitted.

The list $\bar{z}$ again contains all logical variables that occur free in P' or Q' (except the special-purpose logical variable result). The sequence $\bar{v}'$ is a sequence of all local variables that occur free in Q' including `this`. We quantify over these local variables of the callee to prevent confusion with local variables of the caller in P or Q. The precondition of the method P' may only mention the formal parameters and `this`. All other local variables are out of scope in P'.

The simultaneous substitution $[e_0, \bar{e}/\mathtt{this}, \bar{u}]$ models the context switch. It replaces `this` by e_0, and the formal parameters $\bar{u} = u_1, \ldots, u_n$ by the actual parameters $\bar{e} = e_1, \ldots, e_n$. To prevent problems with field shadowing these substitutions should preserve the types of the expressions by introducing casts if necessary (see [9] for further details).

The special-purpose logical variable result denotes the result value. It is only allowed in postconditions of methods. The substitution $[e/\mathsf{result}]$ models a virtual assignment of the result value to result. Similarly, the substitution $[\mathsf{result}/u]$ models an assignment of this value to u.

Observe that the second premiss of the rule is the verification condition of a call with annotation $\{P\}\ u = e_0.m(e_1, \ldots, e_n)\ \{Q\}$ in an object-oriented program where the corresponding implementation of method m has precondition P' and postcondition Q'. To be able to express the verification condition we extended the assertion language by allowing the function symbol f in assertions. However, these symbols will only occur in the verification conditions. The proof outlines retain their desired abstraction level.

The rule looks rather complex because it has to account for all possible contexts. In concrete contexts the resulting verification condition is often simpler. We give an example in Sect. 7.

Next, we analyze reasoning about method invocations that are dynamically bound to an implementation. In such cases we have to consider all implementations of method m that might possibly be bound to the call. Suppose again that we consider a call $u = e_0.m(e_1, \ldots, e_n)$. Recall that $\mathsf{impls}([\![e_0]\!], m)$ denotes the set of classes that provide an implementation for this call. This set tells us how many implementations we must consider.

A second consideration concerns the set of classes that inherit the particular implementation in some class $c \in \mathsf{impls}([\![e_0]\!], m)$. A class inherits the implementation in class c if it is a subclass of class c and it is not a subclass of some class that overrides the implementation in class c. We denote the set of classes that override the implementation of method m in class c by $\mathsf{overrides}(c)(m)$. We have $c' \in \mathsf{overrides}(c)(m)$ if

- c' is a proper subclass of c that provides an implementation of method m, and

– there does not exist another proper subclass c'' of c such that c' is a proper subclass of c'' and c'' also provides an implementation of method m.

With this definition we can formulate in what circumstances the implementation of m in class c is bound to a method call $e_0.m(e_1, \ldots, e_n)$. That occurs when e_0 is an instance of a subclass of c and it is not an instance of a class that overrides this implementation. These conditions are stated in the assertion language by the following formula: e_0 $\mathtt{instanceof}$ $c \wedge \bigwedge_{c' \in \mathsf{overrides}(c)(m)} \neg(e_0 \ \mathtt{instanceof} \ c')$. We denote this formula by $\mathsf{boundto}(e_0, c, m)$.

Let $\mathsf{impls}([\![e_0]\!], m) = \{c_1, \ldots, c_k\}$. Let $m(u^i_1, \ldots, u^i_n)\{\ \bar{v}_i\ S_i\ \mathtt{return}\ e_i\ \}$ be the implementation of method m in class c_i, for $i = 1, \ldots, k$ with effect ψ_i. We assume that the implementation of method m in class c_i has precondition P'_i and postcondition Q'_i. Let B_i denote $\{P'_i\}\ \bar{v}_i\ S_i\ \{Q'_i[e_i/\mathsf{result}]\}$. That is, B_i describes the specification of the implementation in class c_i. The verification condition V_i for the implementation in class c_i is the implication

$$\begin{aligned}&\mathsf{heap}_2 \wedge P\!\downarrow_{\psi_i} \wedge \mathsf{boundto}(e_0, c_i, m)\!\downarrow_{\psi_i} \wedge \\ &\qquad\qquad \forall \bar{z}^{\psi_i}_{\bar{\mu}}(P'_i[e_0, \bar{e}/\mathtt{this}, \bar{u}]\!\downarrow_{\psi_i} \rightarrow \exists \bar{v}'(Q'_i)) \rightarrow Q[\mathsf{result}/u] \ . \qquad (V_i)\end{aligned}$$

Note that we have strengthened the antecedent with the clause that implies that the receiver is an object of a class that inherits the implementation of class c_i.

The rule of adaptation for dynamically-bound method calls (RoA) then simply checks if all specifications of implementations that might possibly be bound to the call can be adapted to the specification of the call. The rule results in one verification condition for each implementation.

$$\frac{B_1, \ldots, B_k \quad V_1, \ldots, V_k}{\{P\}\ u = e_0.m(e_1, \ldots, e_n)\ \{Q\}} \qquad \text{(RoA)}$$

Lemma 4. *Let P be an arbitrary assertion. Let μ be a logical variable of type Object^*. Let $\mu(x_c)$ be a logical variable of type $[\![x]\!]^*$, for every instance variable x declared in some class c. All the introduced logical variables should be fresh (not occurring in P) and distinct.*

Let $\langle S, (\sigma, \tau')\rangle \rightarrow (\sigma', \tau'')$. Let $\psi = \mathsf{sef}(S)(\emptyset) = (cs, fs)$. Let τ be an environment such that $\sigma', \tau \models \mathsf{heap}_1$. Let $\sigma', \tau \models \mathsf{heap}_2$ (for all variables and fields in P). Let

$$\sigma, \tau \models \forall o : \mathsf{Object}(o \in \mu) \qquad (3)$$

if $cs \neq \emptyset$. Finally, assume that

$$\sigma, \tau \models \forall o : c(o.x = \mu(x_c)[f(o)]) \qquad (4)$$

for every instance variable x declared in some class c such that $(c, x) \in fs$. Then

$$\sigma, \tau \models P \text{ if and only if } \sigma', \tau \models P\!\downarrow_\psi \text{ and } [\![P]\!] = [\![P\!\downarrow_\psi]\!] \ .$$

Proof. By structural induction on P. We first consider the most interesting case of an expression $l.x$ and prove

$$\mathcal{L}(l.x)(\sigma, \tau) = \mathcal{L}(l.x\!\downarrow_\psi)(\sigma', \tau) \ .$$

By the induction hypothesis we have $\mathcal{L}(l)(\sigma, \tau) = \mathcal{L}(l{\downharpoonleft_{\psi}})(\sigma', \tau)$ and $\llbracket l \rrbracket = \llbracket l{\downharpoonleft_{\psi}} \rrbracket$. Let $\mathsf{origin}(\llbracket l \rrbracket, x) = c$. Let us first consider the case where $(c, x) \in \mathit{fs}$. We calculate this case as follows.

$$
\begin{array}{ll}
\mathcal{L}(l.x{\downharpoonleft_{\psi}})(\sigma', \tau) & \{ \text{ def. } {\downharpoonleft_{\psi}} \ \} \\
= \mathcal{L}(\mu(x_c)[\mathsf{f}(l{\downharpoonleft_{\psi}})])(\sigma', \tau) & \{ \text{ def. } \mathcal{L}, \text{ where } \tau(\mu(x_c)) = (n, g) \ \} \\
= \text{if } 0 \leq \mathsf{f}(\mathcal{L}(l{\downharpoonleft_{\psi}})(\sigma', \tau)) < n & \\
\quad \text{then } g(\mathsf{f}(\mathcal{L}(l{\downharpoonleft_{\psi}})(\sigma', \tau))) \text{ else } \bot & \{ \text{ ind.hyp. } \} \\
= \text{if } 0 \leq \mathsf{f}(\mathcal{L}(l)(\sigma, \tau)) < n & \\
\quad \text{then } g(\mathsf{f}(\mathcal{L}(l)(\sigma, \tau))) \text{ else } \bot & \{ \text{ def. } \mathcal{L} \ \} \\
= \mathcal{L}(\mu(x_c)[\mathsf{f}(l)])(\sigma, \tau) & \{ \text{ (4) and def. } \mathcal{L} \ \} \\
= \mathcal{L}(l.x)(\sigma, \tau) &
\end{array}
$$

The final step of the proof above also uses the assumption $\sigma, \tau \models \mathsf{heap}_1$ (the first clause of heap_1 implies that $\mathsf{f}(\bot)$ denotes an index that is out of bounds for μ).

Secondly, we prove the case where $(c, x) \notin \mathit{fs}$. This case follows directly from Lemma 1.

$$
\begin{array}{ll}
\mathcal{L}(l.x{\downharpoonleft_{\psi}})(\sigma', \tau) & \{ \text{ def. } {\downharpoonleft_{\psi}} \ \} \\
= \mathcal{L}((l{\downharpoonleft_{\psi}}).x])(\sigma', \tau) & \{ \text{ def. } \mathcal{L} \ \} \\
= \text{if } \mathcal{L}(l{\downharpoonleft_{\psi}})(\sigma', \tau) \in \{\nu, \bot\} \text{ then } \bot & \\
\quad \text{else } \sigma'(\mathcal{L}(l{\downharpoonleft_{\psi}})(\sigma', \tau))(\mathsf{origin}(\llbracket l{\downharpoonleft_{\psi}} \rrbracket, x))(x) & \{ \text{ ind. hyp. } \} \\
= \text{if } \mathcal{L}(l)(\sigma, \tau) \in \{\nu, \bot\} \text{ then } \bot & \\
\quad \text{else } \sigma'(\mathcal{L}(l)(\sigma, \tau))(\mathsf{origin}(\llbracket l \rrbracket, x))(x) & \{ \text{ Lemma 1 } \} \\
= \text{if } \mathcal{L}(l)(\sigma, \tau) \in \{\nu, \bot\} \text{ then } \bot & \\
\quad \text{else } \sigma(\mathcal{L}(l)(\sigma, \tau))(\mathsf{origin}(\llbracket l \rrbracket, x))(x) & \{ \text{ def. } \mathcal{L} \ \} \\
= \mathcal{L}(l.x)(\sigma, \tau) &
\end{array}
$$

Another interesting case is the assertion $\exists z : c(P)$. We must show that

$$\mathcal{A}(\exists z : c(P))(\sigma, \tau) = \mathcal{A}(\exists z : c(P){\downharpoonleft_{\psi}})(\sigma', \tau) \ .$$

Let us first consider the case where $\bigvee_{c' \in cs}(c' \preceq c)$.

$$
\begin{array}{ll}
\mathcal{A}(\exists z : c(P){\downharpoonleft_{\psi}})(\sigma', \tau) & \{ \text{ def. } {\downharpoonleft_{\psi}} \ \} \\
= \mathcal{A}(\exists z_{\bar{\mu}}^{\psi} : t(P{\downharpoonleft_{\psi}}))(\sigma', \tau) & \{ \text{ def. bnd. quantif. } \} \\
= \mathcal{A}(\exists z : c(z \in \mu \wedge (P{\downharpoonleft_{\psi}})))(\sigma', \tau) & \{ \text{ def. } \mathcal{A} \ \} \\
= \mathcal{A}(z \in \mu)(\sigma', \tau[z \mapsto \alpha]) = \mathsf{true} \text{ and} & \\
\quad \mathcal{A}(P{\downharpoonleft_{\psi}})(\sigma', \tau[z \mapsto \alpha]) = \mathsf{true} \text{ and } \alpha \in \mathrm{qdom}(c, \sigma') & \{ \text{ def } \tau' \ \} \\
= \mathcal{A}(z \in \mu)(\sigma', \tau'[z \mapsto \alpha]) = \mathsf{true} \text{ and} & \\
\quad \mathcal{A}(P{\downharpoonleft_{\psi}})(\sigma', \tau[z \mapsto \alpha]) = \mathsf{true} \text{ and } \alpha \in \mathrm{qdom}(c, \sigma') & \{ \text{ Lemma 3 } \} \\
= \mathcal{A}(z \in \mu)(\sigma, \tau[z \mapsto \alpha]) = \mathsf{true} \text{ and} & \\
\quad \mathcal{A}(P{\downharpoonleft_{\psi}})(\sigma', \tau[z \mapsto \alpha]) = \mathsf{true} \text{ and } \alpha \in \mathrm{qdom}(c, \sigma) & \{ \text{ ind. hyp. } \} \\
= \mathcal{A}(z \in \mu)(\sigma, \tau[z \mapsto \alpha]) = \mathsf{true} \text{ and} & \\
\quad \mathcal{A}(P)(\sigma, \tau[z \mapsto \alpha]) = \mathsf{true} \text{ and } \alpha \in \mathrm{qdom}(c, \sigma) & \{ \text{ (3) } \} \\
= \mathcal{A}(P)(\sigma, \tau[z \mapsto \alpha]) = \mathsf{true} \text{ and } \alpha \in \mathrm{qdom}(c, \sigma) & \{ \text{ def. } \mathcal{A} \ \} \\
= \mathcal{A}(\exists z : c(P))(\sigma, \tau) &
\end{array}
$$

The second and last cast occurs when $\bigvee_{c' \in cs}(c' \preceq c)$ does not hold. This case basically follows from Lemma 2.

$$\begin{array}{ll}
\mathcal{A}(\exists z : c(P)\downharpoonleft_{\psi})(\sigma', \tau) & \{\text{ def. } \downharpoonleft_{\psi}\ \} \\
= \mathcal{A}(\exists z^{\psi}_{\bar{\mu}} : t(P\downharpoonleft_{\psi}))(\sigma', \tau) & \{\text{ def. bnd. quantif. }\} \\
= \mathcal{A}(\exists z : c(P\downharpoonleft_{\psi}))(\sigma', \tau) & \{\text{ def. } \mathcal{A}\ \} \\
= \mathcal{A}(P\downharpoonleft_{\psi})(\sigma', \tau[z \mapsto \alpha]) = \mathsf{true} \text{ and } \alpha \in \mathrm{qdom}(c, \sigma') & \{\text{ ind. hyp. }\} \\
= \mathcal{A}(P)(\sigma, \tau[z \mapsto \alpha]) = \mathsf{true} \text{ and } \alpha \in \mathrm{qdom}(c, \sigma') & \{\text{ Lemma 2 }\} \\
= \mathcal{A}(P)(\sigma, \tau[z \mapsto \alpha]) = \mathsf{true} \text{ and } \alpha \in \mathrm{qdom}(c, \sigma) & \{\text{ def. } \mathcal{A}\ \} \\
= \mathcal{A}(\exists z : c(P))(\sigma, \tau) &
\end{array}$$

□

With this lemma we can prove that the adaptation rule (RoA) is sound.

Theorem 1. *Rule (RoA) is sound.*

Proof. Let $\langle u = e_0.m(e_1, \ldots, e_n), (\sigma, \tau)\rangle \rightarrow (\sigma', \tau')$ and let $\sigma, \tau \models P$. The former computation must be derived by rule D from a computation of the body

$$\langle S, (\sigma, \tau_i)\rangle \rightarrow (\sigma', \tau'_i) \quad , \tag{5}$$

given the following assumptions.

- $\mathcal{E}(e_0)(\sigma, \tau) = (c, n)$
- $\mathsf{exec\_impl}(c, m) = m(u_1, \ldots, u_n)\{\ \bar{v}\ S\ \mathtt{return}\ e\ \}$
- $\tau_i = \tau[\mathtt{this}, \bar{u}, \bar{v} \mapsto \mathcal{E}(e_0)(\sigma, \tau), \mathcal{E}(\bar{e})(\sigma, \tau), \mathsf{def}([\![\bar{v}]\!])]$
- $\tau' = \tau[u \mapsto \mathcal{E}(e)(\sigma', \tau'_i)]$

We must prove that $\sigma', \tau' \models Q$.

Let $\bar{v}$ be an arbitrary sequence without repetitions that contains ν and all objects that exist in σ. Let $\bar{v}_{x_c}$ be a sequence of the same length as $\bar{v}$ such that $\bar{v}_{x_c}[i] = \sigma(\bar{v}[i])(c)(x)$, for every valid index i where $\bar{v}[i]$ is an instance of (a subclass of) class c. Let f denote a function such that for every $o \in \bar{v}$ we have $\bar{v}[\mathsf{f}(o)] = o$. Let τ_1 be the environment that is obtained from τ by assigning $\bar{v}$ to μ, and $\bar{v}_{x_c}$ to $\mu(x_c)$, for every field x declared in some class c. Let $\tau_2 = \tau_1[\mathsf{result} \mapsto \mathcal{L}(e)(\sigma', \tau'_i)]$.

Recall that $\mathsf{impls}([\![e_0]\!], m)$ denotes the set of classes that provide an implementation that may be bound to this call. Let c_i be the class in which $\mathsf{exec\_impl}(c, m)$ with effect ψ_i is declared. The corresponding verification condition is

$$\begin{array}{l}
\mathsf{heap}_2 \wedge P\downharpoonleft_{\psi_i} \wedge \mathsf{boundto}(e_0, c_i, m)\downharpoonleft_{\psi_i} \wedge \\
\qquad\qquad \forall \bar{z}^{\psi_i}_{\bar{\mu}}(P'_i[e_0, \bar{e}/\mathtt{this}, \bar{u}]\downharpoonleft_{\psi_i} \rightarrow \exists \bar{v}'(Q'_i)) \rightarrow Q[\mathsf{result}/u] \quad .
\end{array} \tag{6}$$

Since this verification condition is assumed to hold in all states it holds in particular in the state (σ', τ_2). We will prove that (σ', τ_2) satisfies the antecedent of (6) to obtain $\sigma', \tau_2 \models Q[\mathsf{result}/u]$.

By the construction of τ_2 it follows that $\sigma', \tau_2 \models \mathsf{heap}_1$. It is also not difficult to see why $\sigma', \tau_2 \models \mathsf{heap}_2$ holds. The local variables of the caller refer to objects that exist in σ. By the construction of τ_2 they all occur in μ. The same can be said about the logical variables in P. The validity of the last clause of heap_2 follows from the construction of τ_2 and Lemma 1.

Since none of the variables in $\bar{\mu}$ occurs in P we can conclude from $\sigma, \tau \models P$ that $\sigma, \tau_1 \models P$ also holds. We then use Lemma 4 to obtain $\sigma', \tau_1 \models P{\downarrow_\psi}$. Then $\sigma', \tau_2 \models P{\downarrow_\psi}$ also holds because result does not occur in $P{\downarrow_\psi}$.

Because the call is bound to the implementation $\mathsf{exec\_impl}(c, m)$ it must be the case that $\sigma, \tau \models \mathsf{boundto}(e_0, c_i, m)$. By a similar line of reasoning as above this leads to $\sigma', \tau_2 \models \mathsf{boundto}(e_0, c_i, m){\downarrow_{\psi_i}}$.

Let $\bar{\alpha}$ be an arbitrary sequence of values for the variables in $\bar{z}$ such that each value exists in σ'. If the quantification is bounded we assume that $\alpha \in \mu_{\llbracket z \rrbracket}$. Finally, assume that

$$\sigma', \tau_2[\bar{z} \mapsto \bar{\alpha}] \models P'[e_0, \bar{e}/\mathtt{this}, \bar{u}]{\downarrow_\psi} \quad . \qquad (7)$$

Since result does not occur in $P'[e_0, \bar{e}/\mathtt{this}, \bar{u}]{\downarrow_\psi}$ this also implies that (7) holds if we replace τ_2 by τ_1. By Lemma 4 (observe that $\bar{\mu} \cap \bar{z} = \emptyset$) we then infer $\sigma, \tau_1[\bar{z} \mapsto \bar{\alpha}] \models P'[e_0, \bar{e}/\mathtt{this}, \bar{u}]$. The variables in $\bar{\mu}$ do not occur in the formula $P'[e_0, \bar{e}/\mathtt{this}, \bar{u}]$. Therefore $\sigma, \tau[\bar{z} \mapsto \bar{\alpha}] \models P'[e_0, \bar{e}/\mathtt{this}, \bar{u}]$. The validity of $\sigma, \tau_i[\bar{z} \mapsto \bar{\alpha}] \models P'$ then follows from the construction of τ_i.

By the computation in (5) and the specification of the method implementation we conclude $\sigma', \tau_i'[\bar{z} \mapsto \bar{\alpha}] \models Q'[e/\mathsf{result}]$. This is equivalent to

$$\sigma', \tau_i'[\mathsf{result} \mapsto \mathcal{L}(e)(\sigma', \tau_i')][\bar{z} \mapsto \bar{\alpha}] \models Q' \ ,$$

which in turn implies

$$\sigma', \tau_i'[\mathsf{result} \mapsto \mathcal{L}(e)(\sigma', \tau_i')][\bar{z} \mapsto \bar{\alpha}] \models \exists \bar{v}'(Q') \ .$$

The latter formula contains no free occurrences of local variables. All its logical variables (except result) occur in $\bar{z}$. Therefore $\sigma', \tau_2[\bar{z} \mapsto \bar{\alpha}] \models \exists \bar{v}'(Q')$ holds also, which was the final part of the antecedent of the verification condition.

We now have proved that $\sigma', \tau_2 \models Q[\mathsf{result}/u]$. By the construction of τ_2 this is equal to $\sigma', \tau_1[\mathsf{result} \mapsto \mathcal{L}(e)(\sigma', \tau_i')] \models Q[\mathsf{result}/u]$. The variables in $\bar{\mu}$ do not occur in Q so $\sigma', \tau[\mathsf{result} \mapsto \mathcal{L}(e)(\sigma', \tau_i')] \models Q[\mathsf{result}/u]$ also holds. By the construction of τ' and the observation that result does not occur in Q this implies $\sigma', \tau' \models Q$. □

7 An Example

The rule as given above is certainly more complex that most other well-known Hoare rules. It can be conveniently used in proof outlines however if the verification condition is computed automatically. We will give a small example of a program and the resulting verification condition in this section. The example mainly illustrates the elegant way in which the rule handles local variables.

Table 1. An example proof outline

```
class c {
    int x;

    // precondition u == U && this == O
    // postcondition O.x==U
    public void set(int u) {
        this.x=u;
    }

    // precondition u == U && this.x==X
    // postcondition this.x==U && \result==(X==U)
    public boolean testAndSet(int u) {
        boolean b = (this.x==u);
        /*{  u==U && b ==(X==U) }*/
        this.set(u);
        return b; }
}
```

The example program is listed in Table 7. The code defines a simple Java class with two methods. Each method is preceded by a precondition and a postcondition. The capital letters denote logical variables. In the annotation we use the usual Java operators `==` and `&&` for equality and conjunction, respectively.

It may seem strange that we also introduce a logical variable O in the precondition of the `set` method to refer to the old value of `this` in its postcondition. This is done because the verification condition puts an existential quantifier around occurrences of `this` in the postcondition of the method to prevent confusion with occurrences of `this` in the specification of the caller. A possible refinement of the rule could make these steps superfluous by automatically adding the clause `this` $= z$ to the precondition of the method and replacing `this` in the postcondition by z (where z is a fresh logical variable).

The call `this.set(u)` is preceded by an intermediate assertion that is the precondition of the call. The postcondition of this call is obtained by substituting result in the postcondition of the method by b. We assume that the methods are not overridden in subclasses. Therefore this call has the following verification condition.

$$\begin{aligned}&u = U \wedge b = (X = U) \wedge \texttt{this instanceof } c \\ &\qquad\qquad \wedge \forall O : c(\forall U : \texttt{int}(u = U \wedge \texttt{this} = O \rightarrow O.x = U)) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \texttt{this}.x = U \wedge b = (X = U)\end{aligned}$$

The static effect of the `set`-method is $(\emptyset, \{(c, x)\})$. That is, it creates no new objects and only modifies field x in class c. That explains why quantification in this verification condition is not bounded. The predicate heap_2 is not needed to prove this particular verification condition and is therefore left out. It is not difficult to see that the above verification condition is valid if one chooses `this` for O. Our experience is that contemporary theorem provers can prove simple verification conditions like the one above completely automatically.

Observe that the clause $b = (X = U)$ in the precondition of the call can be used to prove the same clause in the postcondition. The specification of the `set` method is totally unrelated to the local variables of the caller.

8 Most General Formulas

In this section we investigate the strength of the rule of adaptation. The adaptation rule complements the (relatively) complete Hoare logic that is proposed in [9]. We prove that that logic is complete in [4].

It is a well-known fact that the rule of adaptation can replace the substitution rule, the invariance rule, and the elimination rule without compromising completeness [2]. In this section we prove that this is also the case for the rule presented here. This means that we will re-examine Lemma 4 of [4]. This Lemma states that we can prove any valid specification of a call if the corresponding implementation is specified with its Most General Formula (MGF). We must show that we can construct a similar proof with the adaptation rule without using the rules that it replaces.

For this purpose, we define an object-oriented counterpart of the well-known Most General Formula (MGF) of a method implementation in an object-oriented program. A MGF reflects the semantics of a statement. The final result of this chapter will be a theorem that states that we can derive any valid correctness formula concerning a method call in one step by means of our rule of adaptation if the corresponding method implementation is annotated similar to the MGF (which is a valid specification for any method implementation).

Our definition of the MGF will be the counterpart of the MGF in the context of simple procedural languages [14]. In such languages the MGF of a statement S is the Hoare triple

$$\{\bar{x} = \bar{y}\}S\{\mathrm{sp}(\bar{x} = \bar{y}, S)\} \enspace .$$

In this formula, $\bar{x}$ is a list of all program variables and $\bar{y}$ is a corresponding list of logical variables. By $\mathrm{sp}(P, S)$ we denote the strongest postcondition of a precondition P and a statement S.

The object-oriented MGF is based on a counterpart of the assertion $\bar{x} = \bar{y}$. This formula 'freezes' the initial values of the program variables in corresponding logical variables. We can do the same for object-oriented programs by applying the technique that is used for the adaptation rule. However, this time we should also store the initial values of the formal parameters and `this`.

Let θ be a fresh logical variable of type Object^*. This variable will store all objects that exist in the initial state. Let $\theta(x_c)$ be a fresh logical variable of type $[\![x]\!]^*$ for every instance variable x defined in some class c. Storing the initial values of the formal parameters in logical variables is straightforward. We simply introduce a logical variable $\theta(u)$ of type $[\![u]\!]$ for each formal parameter u, and similar for `this`. We will assume that $\bar{\theta}$ is a list that contains all the logical variables defined in this paragraph.

This encoding of the heap can be captured logically by a finite conjunction of the following assertions:

- $\forall z : \mathsf{Object}(z \in \theta)$
  which states that the sequence θ contains all existing objects.
- $\forall i(0 \leq i < \theta.\mathsf{length} \land \theta[i]\ \mathtt{instanceof}\ c \rightarrow ((c)\theta[i]).x = \theta(x_c)[i])$
  which states that for each position in θ that contains an instance of (a subclass of) class c the value of the instance variable x of that object can be found in the sequence $\theta(x_c)$ at the same position.
- $\mathtt{this} = \theta(\mathtt{this})$
  which states that the value of `this` is stored in $\theta(\mathtt{this})$.
- $u_i = \theta(u_i)$
  which states that the value of the formal parameter u_i is stored in $\theta(u_i)$.

We only have to freeze the values of fields that occur in the write effects of a statement. Therefore we will only include the second clause for fields that occur in the write effects of a method. We denote the resulting conjunction by $\textsc{init}_\psi$.

The MGF of an arbitrary implementation $m(u_1, \ldots, u_n)\{\bar{v}\ S\ \mathtt{return}\ e\}$ with static effect ψ is the Hoare triple $\{\textsc{init}\}\ \bar{v}\ S\ \{\mathrm{sp}(\textsc{init}_\psi, \bar{v}\ S) \land \mathsf{result} = e\}$. We use the following semantical definition of the strongest postcondition $\mathrm{sp}(P, \bar{v}\ S)$:

$$\{(\sigma, \tau) \mid \exists \sigma', \tau' \text{ such that } \tau'' = \tau[\bar{v} \mapsto \mathsf{def}([\![\bar{v}]\!])] \text{ and } \langle S, (\sigma', \tau'')\rangle \rightarrow (\sigma, \tau) \text{ and } \sigma', \tau'' \models P\}$$

It is not difficult to see that in this way

$$\{\textsc{init}\}\ \bar{v}\ S\ \{(\mathrm{sp}(\textsc{init}_\psi, \bar{v}\ S) \land \mathsf{result} = e)[e/\mathsf{result}]\}$$

always holds.

The following theorem states that we can derive any valid correctness formula concerning a call to a method in one step by means of our rule of adaptation if the corresponding method implementations are all annotated with their MGF.

Theorem 2. *Let $\{P\}\ u = e_0.m(e_1, \ldots, e_n)\ \{Q\}$ be a valid specification of the call. Let $\mathsf{impls}([\![e_0]\!], m) = \{c_1, \ldots, c_k\}$. Let $i \in \{1 \ldots k\}$. Let the implementation of method m in class c_i be $m_i(u^i_1, \ldots, u^i_n)\{\bar{v}_i\ S_i\ \mathtt{return}\ e_i\}$ with effect ψ_i. Let* $\textsc{init}$ *denote the formula defined above such that no variable in $\bar{\theta}$ occurs in P or Q. Then*

$$\mathsf{heap}_1 \land \mathsf{heap}_2 \land P\!\downharpoonleft_{\psi_i} \land \mathsf{boundto}(e_0, c_i, m)\!\downharpoonleft_{\psi_i} \land$$
$$\forall \bar{z}^{\psi_i}_{\mu}(\textsc{init}_{\psi_i}[e_0, \bar{e}/\mathtt{this}, \bar{u}_i]\!\downharpoonleft_{\psi_i} \rightarrow \exists \bar{v}'(\mathrm{sp}(\textsc{init}_{\psi_i}, \bar{v}_i\ S_i) \land \mathsf{result} = e) \rightarrow Q[\mathsf{result}/u].$$

Proof. Note that we must prove the verification condition for an arbitrary i that corresponds to a fixed implementation. Therefore we will drop all the subscript and superscript $i's$ in the following proof.

The outline of the proof is as follows. We will consider a state (σ, τ) that satisfies the antecedent of the implementation. Then we will modify the environment τ to obtain an environment τ' such that

$$\sigma, \tau' \models \textsc{init}_\psi[e_0, \bar{e}/\mathtt{this}, \bar{u}]\downarrow_\psi \ .$$

This implies that $\sigma, \tau' \models \exists\bar{v}(\mathrm{sp}(\textsc{init}_\psi, \bar{v}\ S) \wedge \mathsf{result} = e))$ also holds. This will enables us to construct a computation of the method implementation that starts in the state $(\sigma_0, \tau'[\mathtt{this}, \bar{u} \mapsto \mathcal{L}(e_0)(\sigma_0, \tau'), \mathcal{L}(\bar{e})(\sigma_0, \tau')])$. Next we show that we can extend this computation of the implementation to a computation of the call starting in the state (σ_0, τ'). We will prove that this state (σ_0, τ') satisfies P. Therefore we can conclude that Q holds in the final state of the latter computation. This will imply that $\sigma, \tau \models Q[\mathsf{result}/u]$ holds.

Let σ, τ be such that

$$\begin{aligned} \sigma, \tau \models \mathsf{heap}_1 \wedge \mathsf{heap}_2 \wedge P\downarrow_\psi \wedge \mathsf{boundto}(e_0, c, m)\downarrow_\psi \wedge \\ \forall\bar{z}^\psi_\mu(\textsc{init}_\psi[e_0, \bar{e}/\mathtt{this}, \bar{u}]\downarrow_\psi \rightarrow \exists\bar{v}'(\mathrm{sp}(\textsc{init}_\psi, \bar{v}\ S) \wedge \mathsf{result} = e) \ . \end{aligned} \quad (8)$$

Let $\psi = (cs, fs)$ be the effect of $\mathsf{exec\_impl}(c_i, m)$. Let τ' be the environment that results from τ by assigning $\tau(\mu)$ to θ if cs $\neq \emptyset$. Otherwise $\tau'(\theta)$ is a sequence that contains all objects that exist in σ. The value of $\tau'(\theta(x_c)) = \tau(\mu(x_c))$ for each $(c, x) \in fs$. Furthermore τ' should assign the value of $\mathcal{L}(e_0\downarrow_\psi)(\sigma, \tau)$ to $\theta(\mathtt{this})$, and $\mathcal{L}(e_i\downarrow_\psi)(\sigma, \tau)$ to $\theta(u_i)$, for $i \in \{1 \ldots n\}$. Finally, τ' should assign $\mathsf{def}(u)$ to every local variable u that does not occur in P or e_i, for $i \in \{0 \ldots n\}$. To all other variables τ' assigns the same values as τ. Observe that for τ' we also have

$$\sigma, \tau' \models \mathsf{heap}_1 \wedge \mathsf{heap}_2 \wedge P\downarrow_\psi \wedge \mathsf{boundto}(e_0, c, m)\downarrow_\psi$$

because the freeze variables do not occur in that formula. Moreover, we know that

$$\sigma, \tau' \models \forall\bar{z}^\psi_\mu(\textsc{init}_\psi[e_0, \bar{e}/\mathtt{this}, \bar{u}]\downarrow_\psi \rightarrow \exists\bar{v}'(\mathrm{sp}(\textsc{init}_\psi, \bar{v}\ S) \wedge \mathsf{result} = e)) \quad (9)$$

also holds because the freeze variables do not occur free in the formula. We now wish to drop the outermost quantifiers by instantiating the universally quantified variables from this formula with elements of the old heap μ. This requires us to prove that each value that is assigned to a freeze variable by τ' is a subsequence of $\tau'(\mu)$ if $cs \neq \emptyset$, which implies that quantification over θ is bounded.

Note that $\tau'(\theta) \sqsubseteq \tau'(\mu)$ if $cs \neq \emptyset$ because $\tau'(\theta) = \tau'(\mu)$ in that case. For each $(c, x) \in fs$ we have $\tau'(\theta(x_c) = \tau'(\mu(x_c)$, whereas the latter is a subsequence of the old heap according to $\sigma, \tau' \models \mathsf{heap}_1$. Finally, $\sigma, \tau \models \mathsf{heap}_2$ implies that $\theta(\mathtt{this})$ and $\theta(u)$ also reference objects in the old heap.

The above analysis shows that each logical variable of a class type $\bar{z}$ references an object in τ'. Therefore we may drop the outermost quantifiers in (9), which yields the following implication.

$$\sigma, \tau' \models \textsc{init}_\psi[e_0, \bar{e}/\mathtt{this}, \bar{u}]\downarrow_\psi \rightarrow \exists\bar{v}'(\mathrm{sp}(\textsc{init}_\psi, \bar{v};\ S) \wedge \mathsf{result} = e) \ . \quad (10)$$

Let's now focus at the antecedent $\textsc{init}_\psi[e_0, \bar{e}/\mathtt{this}, \bar{u}]\downharpoonleft_\psi$. If we compute the result of $\forall z : \mathsf{Object}(z \in \theta)[e_0, \bar{e}/\mathtt{this}, \bar{u}]\downharpoonleft_\psi$ we obtain $\forall z : \mathsf{Object}(z \in \mu \rightarrow z \in \theta)$ if cs $\neq \emptyset$, and $\forall z : \mathsf{Object}(z \in \theta)$ otherwise. Both formulas clearly hold in the state (σ, τ') due to the construction of τ'.

Another interesting part is the result of $[e_0, \bar{e}/\mathtt{this}, \bar{u}]\downharpoonleft_\psi$ on

$$\forall i(0 \leq i < \theta.\mathsf{length} \wedge \theta[i] \ \mathtt{instanceof}\ c \rightarrow ((c)\theta[i]).x = \theta(x_c)[i]) \ .$$

The following computation shows the outcome.

$$\begin{aligned}&\forall i(0 \leq i < \theta.\mathsf{length} \wedge \theta[i] \ \mathtt{instanceof}\ c \rightarrow ((c)\theta[i]).x = \theta(x_c)[i])[e_0, \bar{e}/\mathtt{this}, \bar{u}]\downharpoonleft_\psi \\ &\equiv \forall i(0 \leq i < \theta.\mathsf{length} \wedge \theta[i] \ \mathtt{instanceof}\ c \rightarrow ((c)\theta[i]).x = \theta(x_c)[i])\downharpoonleft_\psi \\ &\equiv \forall i(0 \leq i < \theta.\mathsf{length} \wedge \theta[i] \ \mathtt{instanceof}\ c \rightarrow \mu(x_c)[\mathsf{f}((c)\theta[i])] = \theta(x_c)[i]) \ ,\end{aligned}$$

This formula is valid in (σ, τ') by the construction of τ' and $\sigma, \tau' \models \mathsf{heap}_1$.

The rest of $\textsc{init}_\psi$ yields the following parts.

$$\begin{gathered}u_i = \theta(u_i)[e_0, \bar{e}/\mathtt{this}, \bar{u}]\downharpoonleft_\psi \equiv e_i\downharpoonleft_\psi = \theta(u_i) \\ \mathtt{this} = \theta(\mathtt{this})[e_0, \bar{e}/\mathtt{this}, \bar{u}]\downharpoonleft_\psi \equiv e_0\downharpoonleft_\psi = \theta(\mathtt{this})\end{gathered}$$

These clauses are valid in (σ, τ') because $\sigma, \tau' \models \mathsf{heap}_2$. Therefore we conclude $\sigma, \tau' \models \textsc{init}_\psi[e_0, \bar{e}/\mathtt{this}, \bar{u}]\downharpoonleft_\psi$. From (10) and the previous formula we may infer $\sigma, \tau' \models \exists \bar{v}'(\mathrm{sp}(\textsc{init}_\psi, \bar{v}\ S) \wedge \mathsf{result} = e)$. This formula guarantees the existence of a sequence of values $\bar{\alpha}$ such that

$$\sigma, \tau'[\bar{v}' \mapsto \bar{\alpha}] \models \mathrm{sp}(\textsc{init}_\psi, \bar{v}\ S) \wedge \mathsf{result} = e.$$

By the definition of the strongest postcondition sp this implies that there exists a state (σ_0, τ_0) such that

$$\langle S, (\sigma_0, \tau_0')\rangle \rightarrow (\sigma, \tau'[\bar{v}' \mapsto \bar{\alpha}]) \ , \qquad (11)$$

where $\tau_0' = \tau_0[\bar{v} \mapsto \mathsf{def}(\llbracket \bar{v} \rrbracket)]$ and $\sigma_0, \tau_0' \models \textsc{init}_\psi$.

It is easy to prove that τ_0' assigns the same values to the logical variables as $\tau'[\bar{v}' \mapsto \bar{\alpha}]$. Because $\bar{v}'$ is a sequence of local variables we also know that τ_0' assigns the same values to the logical variables as τ'. By $\sigma, \tau' \models \mathsf{heap}_2$ we know that every local variable of a reference type that occurs in P, Q or e_i, for $i \in \{0 \ldots n\}$ refers to a value that occurs in μ. All other local variables have their default value in τ'. Therefore τ' is consistent with σ_0.

From $\sigma, \tau' \models \mathsf{heap}_1 \wedge \mathsf{heap}_2$ it follows that $\sigma_0, \tau' \models \mathsf{heap}_1 \wedge \mathsf{heap}_2$. We also have $\sigma_0, \tau' \models \forall z : \mathsf{Object}(z \in \mu)$ if $cs \neq \emptyset$ by $\sigma_0, \tau_0' \models \textsc{init}_\psi$ and the construction of τ'. From $\sigma_0, \tau' \models \mathsf{heap}_1$ and $\sigma_0, \tau' \models \textsc{init}_\psi$ we can prove that

$$\sigma_0, \tau' \models \forall o : c(o.x = \mu(x_c)[f_c(o)])$$

holds for every instance variable x declared in some class c such that $(c, x) \in fs$. Therefore all conditions of Lemma 4 are met.

We want to extend the computation in (11) to a computation of the method call. Observe that

$$\begin{aligned}\tau_0'(\texttt{this}) = \tau_0'(\theta(\texttt{this})) = \tau'(\theta(\texttt{this})) &= \mathcal{L}(e_0\downharpoonleft_\psi)(\sigma,\tau)\\ &= \mathcal{L}(e_0\downharpoonleft_\psi)(\sigma,\tau') = \mathcal{L}(e_0)(\sigma_0,\tau') \ .\end{aligned}$$

The last step follows from an application of Lemma 4. In a similar way we obtain $\tau_i(u_i) = \mathcal{L}(e_i)(\sigma_0, \tau')$, for every $i \in \{1 \dots n\}$.

By the SOS-rule for method calls we then infer from the computation in (11) that

$$\langle u = e_0.m(e_1,\dots,e_n), (\sigma_0,\tau')\rangle \to (\sigma,\tau'') \ . \tag{12}$$

where $\tau'' = \tau'[u \mapsto \mathcal{L}(e)(\sigma, \tau'[\bar{v}' \mapsto \bar{\alpha}])]$.

We have observed that $\sigma, \tau' \models P\downharpoonleft_\psi$. By Lemma 4 this implies $\sigma_0, \tau' \models P$. By the run in (12) and the validity of the premiss of the rule we obtain $\sigma, \tau'' \models Q$. We repeat that $\sigma, \tau'[\bar{v}' \mapsto \bar{\alpha}] \models \mathsf{result} = e$. But then $\sigma, \tau' \models Q[\mathsf{result}/u]$. Then $\sigma, \tau \models Q[\mathsf{result}/u]$ holds by the construction of τ'. □

9 Conclusions

The main result of this paper is a (to the best of our knowledge) first rule of adaptation for the object-oriented paradigm. The rule not only enables us to adapt the values of logical variables but also results in an important separation of concerns for local variables in proof outlines for object-oriented programs. This improves the modularity of the specification. The rule uses a static approximation of the effect of a method to further improve the modularity of the specification.

The adaptation rule complements our earlier work on a (relatively) complete Hoare logic [9] for object-oriented programs. It is a well-known fact that the rule of adaptation can replace the substitution rule, the invariance rule, and the elimination rule without compromising completeness [2]. In the full version we show that this is also the case for the rule presented here.

Proof outlines are not only a compact representation of correctness proofs but are also useful for documentation purposes. The rule of adaptation describes the verification condition that results from the specification of a call and the corresponding method. That is precisely what is needed in a proof outline logic. Thus the rule enables a shift from Hoare logics to proof outlines for OO. It is unclear if and how the rules that it replaces can be used in proof outlines.

Future work includes an investigation of the sharpness [15, 16] of the proof rule. The present rule also seems a good starting point for an investigation of the benefits of more advanced type systems for alias control in object-oriented languages (cf. [17, 18]) in the context of method calls.

References

1. Hoare, C.A.R.: Procedures and parameters: An axiomatic approach. In Engeler, E., ed.: Symp. on Semantics of Algorithmic Languages. Volume 188 of Lecture Notes in Mathematics. (1971) 102–116
2. Olderog, E.R.: On the notion of expressiveness and the rule of adaptation. Theoretical Computer Science **24** (1983) 337–347
3. de Boer, F., Pierik, C.: Computer-aided specification and verification of annotated object-oriented programs. In: Proc. of Formal Methods for Open Object-Based Systems V (FMOODS 2002), Kluwer Academic Publishers (2002) 163–177
4. Pierik, C., de Boer, F.S.: A syntax-directed Hoare logic for object-oriented programming concepts. Technical Report UU-CS-2003-010, Institute of Information and Computing Sciences, Utrecht University, The Netherlands (2003) Available from http://www.cs.uu.nl/research/techreps/UU-CS-2003-010.html.
5. Poetzsch-Heffter, A., Müller, P.: A programming logic for sequential Java. In Swierstra, S.D., ed.: ESOP '99. Volume 1576 of LNCS. (1999) 162–176
6. von Oheimb, D.: Hoare logic for Java in Isabelle/HOL. Concurrency and Computation: Practice and Experience **13** (2001) 1173–1214
7. Huisman, M.: Reasoning about Java programs in higher order logic with PVS and Isabelle. PhD thesis, Katholieke Universiteit Nijmegen (2001)
8. Reus, B., Wirsing, M., Hennicker, R.: A Hoare calculus for verifying Java realizations of OCL-constrained design models. In Hussmann, H., ed.: FASE 2001. Volume 2029 of LNCS. (2001) 300–317
9. Pierik, C., de Boer, F.S.: A syntax-directed Hoare logic for object-oriented programming concepts. In: Proc. of Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003). Volume 2884 of LNCS. (2003) 64–78
10. Kleymann, T.: Hoare logic and auxiliary variables. Formal Aspects of Computing **11** (1999) 541–566
11. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06y, Department of Computer Science, Iowa State University (2004)
12. Cataño, N., Huisman, M.: Chase: a static checker for JML's assignable clause. In: Verification, Model Checking and Abstract Interpretation (VMCAI '03). Volume 2575 of LNCS. (2003) 26–40
13. Zwiers, J., Hannemann, U., Lakhnech, Y., de Roever, W.P., Stomp, F.: Modular completeness: Integrating the reuse of specified software in top-down program development. In: Formal Methods Europe, FME'96 Symposium. Volume 1051 of LNCS. (1996) 595–608
14. Gorelick, G.: A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report 75, Dep. Computer Science, Univ. Toronto (1975)
15. Bijlsma, A., Matthews, P., Wiltink, J.: A sharp proof rule for procedures in wp semantics. Acta Informatica **26** (1989) 409–419
16. Naumann, D.A.: Calculating sharp adaptation rules. Information Processing Letters **7** (2000) 201–208
17. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: Proc. of OOPSLA 2002. Volume 37 of ACM SIGPLAN Notices. (2002) 292–310
18. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. Volume 2262 of LNCS. Springer (2002)