Universitetet i Oslo
Institutt for informatikk

Lazy Behavioral Subtyping

Johan Dovland,
Einar B. Johnsen,
Olaf Owe, and
Martin Steffen

Research Report 368
ISBN 82-7368-326-5
(Revised version)

November 2007
Revised March 2008

Lazy Behavioral Subtyping

Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen

Dept. of Informatics, University of Oslo, Norway
{johand,einarj,olaf,msteffen}@ifi.uio.no

Abstract. Late binding allows flexible code reuse but complicates formal reasoning significantly, as a method call's receiver class is not statically known. This is especially true when programs are incrementally developed by extending class hierarchies. This report develops a novel method to reason about late bound method calls. In contrast to traditional behavioral subtyping, reverification is avoided without restricting method overriding to fully behavior-preserving redefinition. The approach ensures that when analyzing the methods of a class, it suffices to consider that class and its superclasses. Thus, the full class hierarchy is not needed, and *incremental* reasoning is supported. We formalize this approach as a calculus which lazily imposes context-dependent subtyping constraints on method definitions. The calculus ensures that all method specifications required by late bound calls remain satisfied when new classes extend a class hierarchy. The calculus does not depend on a specific program logic, but the examples in the report use a Hoare-style proof system. We show soundness of the analysis method.

1 Introduction

Late binding of method calls is a central feature in object-oriented languages and contributes to flexible code reuse. A class may extend its superclasses with new methods, possibly overriding the existing ones. This flexibility comes at a price: It significantly complicates reasoning about method calls as the binding of a method call to code cannot be statically determined; i.e., the binding at run-time depends on the actual class of the called object. In addition, object-oriented programs are often designed under an *open world assumption*: Class hierarchies are extended over time as subclasses are gradually developed and added. In general, a class hierarchy may be extended with new subclasses in the future, which will lead to new potential bindings for overridden methods.

To control this flexibility, existing reasoning and verification strategies impose restrictions on inheritance and redefinition. One strategy is to ignore openness and assume a "closed world"; i.e., the proof rules assume that the complete inheritance tree is available at reasoning time (e.g., [25]). This severely restricts the applicability of the proof strategy; for example, libraries are designed to be extended. Moreover, the closed world assumption contradicts inheritance as an object-oriented design principle, which is intended to support incremental development and analysis. If the reasoning relies on the world being closed, extending the class hierarchy requires a costly reverification.

An alternative strategy is to reflect in the verification system that the world is open, but to constrain how methods may be redefined. The general idea is that to avoid reveri-

$$
\begin{array}{ll}
P ::= \overline{L}\ \{t\} & L ::= \texttt{class}\ C\ \texttt{extends}\ C\ \{\overline{f}\ \overline{M}\} \\
M ::= m\ (\overline{x})\{t\} & t\ ::= v := \texttt{new}\ C() \mid v := e.m(\overline{e}) \mid v := e \\
v\ ::= f \mid \texttt{return} & \qquad \mid \texttt{skip} \mid \texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ t\ \texttt{fi} \mid t;t
\end{array}
$$

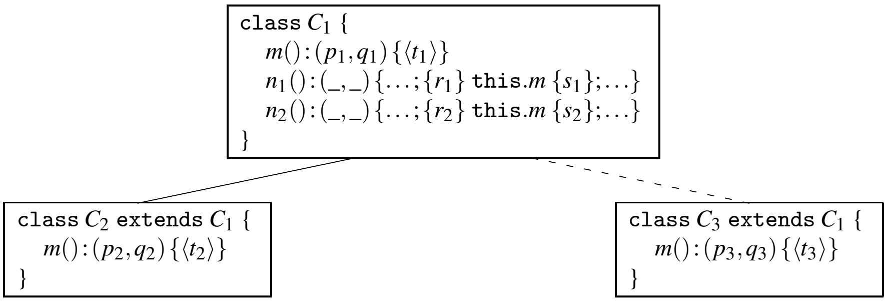
Fig. 1. The language syntax, where C and m are class and method names (of types *Cid* and *Mid*, respectively). Expressions e include declared fields f, the reserved variables `this` and `return`, and Boolean expressions b. Vector notation denotes lists; e.g., a list of expressions is written $\overline{e}$.

fication, any redefinition of a method through overriding must *preserve* certain properties of the method being redefined. An important part of the properties to be preserved is the method's contract; i.e., the pre- and postconditions for its body. The contract can be seen as a description of the promised behavior of all implementations of the method as part of its interface description, the method's *commitment*. Best known as *behavioral subtyping* (e.g, [2, 19, 20, 26]), this strategy achieves incremental reasoning by limiting the possibilities for code reuse. Once a method has committed to a contract, this commitment may not change in later redefinitions. That is overly restrictive and often violated in practice [27]; e.g., it is not respected by the standard Java library definitions.

This report relaxes the property preservation restriction of behavioral subtyping, while embracing the open world assumption of incremental program development. The basic idea is as follows: given a method m declared with p and q as the method's pre- and postcondition, there is no need to restrict the behavior of methods overriding m and require that these adhere to that specification. Instead it suffices to preserve the "part" of p and q actually *used to verify* the program at the current stage. Specifically, if m is used in the program in the form of a method call $\{r\}\ e.m(\ldots)\ \{s\}$, the pre- and postconditions r and s at that call-site constitute m's *required* behavior and it is those weaker conditions that need to be preserved to avoid reverification. We call the corresponding analysis strategy *lazy behavioral subtyping*. This strategy may serve as a blueprint for integrating a flexible system for program verification of late bound method calls into object-oriented program development and analysis tools environments [5–7].

The report formalizes this analysis strategy using an object-oriented kernel language, based on Featherweight Java [15], and using Hoare-style proof outlines. Formalized as a syntax-driven inference system, class analysis is done in the context of a *proof environment* constructed during the analysis. The environment keeps track of the context-dependent requirements on method definitions, derived from late bound calls. The strategy is incremental; for the analysis of a class C, only knowledge of C and its superclasses is needed. We show the soundness of the proposed method.

Report overview. Sect. 2 introduces the problem of reasoning about late binding, Sect. 3 presents the approach taken in this report, and Sect. 4 gives the details of the inference system. Related work is discussed in Sect. 5 and Sect. 6 concludes the report.



`class` C_1 {
$m() : (p_1, q_1)\,\{\langle t_1 \rangle\}$
$n_1() : (\_,\_)\,\{\ldots;\{r_1\}$ `this`$.m$ $\{s_1\};\ldots\}$
$n_2() : (\_,\_)\,\{\ldots;\{r_2\}$ `this`$.m$ $\{s_2\};\ldots\}$
}

`class` C_2 `extends` C_1 {
$m() : (p_2, q_2)\,\{\langle t_2 \rangle\}$
}

`class` C_3 `extends` C_1 {
$m() : (p_3, q_3)\,\{\langle t_3 \rangle\}$
}

Fig. 2. A class hierarchy with proof outlines for overridden methods.

2 Late Bound Method Calls

2.1 Syntax for an Object-Oriented Kernel Language

To succinctly explain late binding and our analysis strategy, we use an object-oriented kernel language (Fig. 1) with a standard operational semantics (e.g., [15]) . We assume a functional language of side-effect free expressions e. A program P consists of a list $\overline{L}$ of class definitions, and a method body. A class extends a superclass, which may be `Object`, with fields $\overline{f}$ and methods $\overline{M}$. To simplify, we let fields have distinct names, methods with the same name have the same signature (i.e., no method overloading), programs be well-typed, and ignore the types of fields and methods. For classes B and C, $B \leq C$ denotes the reflexive and transitive subclass relation derived from class inheritance. If $B \leq C$, we say that B is *below* C and C is *above* B.

A method M takes parameters $\overline{x}$ and contains a statement t, which may be composed. The sequential composition of statements t_1 and t_2 is written $t_1;t_2$. The statement $v :=$ `new` $C()$ creates a new object of class C with fields instantiated to default values, and assigns the new reference to v. A possible constructor method in the class must be called explicitly. In a method invocation $e.m(\overline{e})$, the object e receives a call to the method m with actual parameters $\overline{e}$. The statement $v := e.m(\overline{e})$ assigns the value of the method activation's `return` variable to v. (For convenience, we often write $e.m(\overline{e})$ or simply $e.m$ instead of $v := e.m(\overline{e})$.) There are standard statements for `skip`, conditionals `if` b `then` t `else` t `fi`, and assignments $v := e$. As usual, `this` is read only.

Late binding or dynamic dispatch is a central concept of object-orientation, already present in Simula [8]. A method call is late bound, or *virtual*, if the method body to be executed is selected at run-time, depending on the callee's actual class. Virtual calls are bound to the first implementation found above the actual class. The mechanism can be illustrated by an object of class C_2 which executes a method n_1 defined in its superclass C_1 and this method issues a call to a method m defined in both classes (see Fig. 2). With late binding, the code selected for execution is associated to the first matching signature for m above C_2; i.e., m of C_2 is selected and not the one in C_1. If n_1, however, were executed in an instance of C_1, the virtual invocation of m would be bound to the definition in C_1. This use of late binding underlies the template method pattern [11]: a base class provides architecture and subclasses provide the specialized (auxiliary)

$$\text{(ASSIGN)}\ \{q[e/v]\}\ v := e\ \{q\}$$

$$\text{(NEW)}\ \{q[\mathtt{new}_C/v]\}\ v := \mathtt{new}\ C()\ \{q\}$$

$$\text{(SKIP)}\ \{q\}\ \mathtt{skip}\ \{q\}$$

$$\text{(COND)}\ \frac{\{p \wedge b\}\ t_1\ \{q\} \qquad \{p \wedge \neg b\}\ t_2\ \{q\}}{\{p\}\ \mathtt{if}\ b\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2\ \mathtt{fi}\ \{q\}}$$

$$\text{(SEQ)}\ \frac{\{p\}\ t_1\ \{r\} \quad \{r\}\ t_2\ \{q\}}{\{p\}\ t_1 ; t_2\ \{q\}} \qquad \text{(ADAPT)}\ \frac{p \Rightarrow p_1 \quad \{p_1\}\ t\ \{q_1\} \quad q_1 \Rightarrow q}{\{p\}\ t\ \{q\}}$$

$$\text{(CALL)}\ \frac{\forall i \in S\,.\,\{p_i[\overline{e}/\overline{x}]\}\ body^{i}_{m(\overline{x})}\ \{q_i\}}{\{\bigwedge_{i \in S}(p_i[\overline{e}/\overline{x}])\}\ v := e.m(\overline{e})\ \{\bigvee_{i \in S}(q_i[v/\mathtt{return}])\}}\ S = \mathtt{implements}(\mathtt{classOf}(e),\ m)$$

Fig. 3. Closed world proof rules. Let $\mathtt{classOf}(e)$ denote the class of expression e and $p[e/v]$ the substitution of all occurrences of v in p by e [12], extended for object creation following [25]. The function $\mathtt{implements}(C,m)$ returns all classes where a call to m from class C may be bound.

methods, while code reuse is supported for the architecture. We say that a definition of m is *reachable* from C if there is a class $D \leq C$ such that a call to m will bind to that definition for instances of D. For instance, if m is overridden by D, that definition is reached from C for instances of D. Thus, for a virtual call there might be several reachable definitions.

2.2 Reasoning about Virtual Calls

Apart from the treatment of late bound method calls, our reasoning system for the other statements follows standard proof rules [3, 4] for partial correctness, adapted to the object-oriented setting; in particular, de Boer's technique using sequences in the assertion language addresses the issue of object creation [9]. We present the proof system using Hoare triples $\{p\}\ t\ \{q\}$, where p is the precondition and q is the postcondition to the statement t [12]. The meaning of a triple $\{p\}\ t\ \{q\}$ is standard: if t is executed in a state where p holds and the execution terminates, then q holds after t. The derivation of triples can be done in any suitable program logic. Let PL be such a program logic and let $\vdash_{\mathrm{PL}} \{p\}\ t\ \{q\}$ denote that $\{p\}\ t\ \{q\}$ is derivable in PL. A *proof outline* [24] for a method definition $m(\overline{x})\{t\}$ is an annotated method $m(\overline{x}) : (p,q)\, \{\langle t \rangle\}$ where $\langle t \rangle$ is the method body t annotated with pre- and postconditions to method calls. The derivability $\vdash_{\mathrm{PL}} m(\overline{x}) : (p,q)\, \{\langle t \rangle\}$ of a proof outline is given by $\vdash_{\mathrm{PL}} \{p\}\ \langle t \rangle\ \{q\}$. For $m(\overline{x}) : (p,q)\, \{\langle t \rangle\}$, the pair (p,q) is called the *commitment* of method m. For simplicity, we assume that `return` does not occur in p and that $\overline{x}$ do not occur in q. To prove an assertion, the annotated method body $\langle t \rangle$ may impose *requirements* on methods called within t, expressed by pre- and postconditions to those calls. For a call $\{r\}\ n()\ \{s\}$ in $\langle t \rangle$, (r,s) is the required assertion for n. To ensure that the requirement is valid, every reachable definition of n must be analyzed.

If the proof system assumes a closed world, all classes must be defined before the analysis can begin, as the requirement to a method call is derived from the commitments of all reachable implementations of that method. To simplify the presentation in this report, we omit further details of the assertion language and the proof system (e.g., ignoring the representation of the program semantics — for details see [25]). The corresponding proof system is given in Fig. 3; the proof rule (CALL) captures late binding under a closed world assumption. The following example illustrates the proof system.

Example 1. Consider the class hierarchy of Fig. 2, where the methods are decorated with proof outlines. The specifications of methods n_1 and n_2 play no role in the discussion and are given a wildcard notation $(\_,\_)$. Assume $\vdash_{\text{PL}} m() : (p_1, q_1)\,\{\langle t_1 \rangle\}$, $\vdash_{\text{PL}} m() : (p_2, q_2)\,\{\langle t_2 \rangle\}$, and $\vdash_{\text{PL}} m() : (p_3, q_3)\,\{\langle t_3 \rangle\}$ for the definitions of m in classes C_1, C_2, and C_3, respectively. Let us initially consider the class hierarchy consisting of C_1 and C_2 and ignore C_3 for the moment. The proof system of Fig. 3 gives the Hoare triple $\{p_1 \wedge p_2\}\mathtt{this}.m()\{q_1 \vee q_2\}$ for each call to m, i.e., for the calls in the bodies of methods n_1 and n_2 in class C_1. To apply (ADAPT), we get the proof obligations: $r_1 \Rightarrow p_1 \wedge p_2$ and $q_1 \vee q_2 \Rightarrow s_1$ for n_1, and $r_2 \Rightarrow p_1 \wedge p_2$, and $q_1 \vee q_2 \Rightarrow s_2$ for n_2. *Extending* now the class hierarchy with C_3 breaks the closed world assumption and requires to *reverify* the methods n_1 and n_2. With the new Hoare triple $\{p_1 \wedge p_2 \wedge p_3\}\mathtt{this}.m()\{q_1 \vee q_2 \vee q_3\}$ at every call site, the proof obligations given above for applying (ADAPT) no longer apply.

3 A Lazy Approach to Virtual Calls

This section presents informally the approach to reason about virtual calls which is based on an open world assumption. It supports incremental reasoning about classes and is well-suited for program development, being less restrictive than behavioral subtyping. A formal presentation is given in Sect. 4.

Reconsider class C_1 of Example 1. The proof outlines for n_1 and n_2 require that $\{r_1\}\,\mathtt{this}.m()\,\{s_1\}$ and $\{r_2\}\,\mathtt{this}.m()\,\{s_2\}$ hold in the bodies of n_1 and n_2, respectively. The assertions (r_1, s_1) and (r_2, s_2) may be seen as *requirements* to reachable definitions of m; for m's definition in C_1, both $\{r_1\}\ t_1\ \{s_1\}$ and $\{r_2\}\ t_1\ \{s_2\}$ must hold. However, the proof obligations for method calls have shifted from the call site to the declaration site, which allows incremental reasoning. During the verification of a class only the class and its superclasses need to be considered, subclasses are ignored. If we later analyze subclass C_2 or C_3, the *same requirements* apply to their definition of m. Thus, no reverification of the bodies of n_1 and n_2 is needed when new subclasses are analyzed.

Although C_1 is analyzed independently of C_2 and C_3, its requirements must be considered during subclass analysis. For this purpose, a *proof environment* is constructed while analyzing C_1 recording that C_1 requires both (r_1, s_1) and (r_2, s_2) from m. Subclasses are analyzed in the context of this proof environment, and may in turn extend the proof environment with new requirements, tracking the scope of each requirement. For two independent subclasses, the requirements made by one subclass should not affect the other. Hence, the order of subclass analysis does not influence the assertions to be verified in each class. To avoid reverification, the proof environment also tracks the commitments established for each method definition. The analysis of a requirement to a method definition immediately succeeds if the requirement follows from the previously established commitments of that method.

3.1 Assertions and Assertion Entailment

We consider an assertion language with expressions e constructed as follows:

$$e ::= f \mid z \mid ops(\bar{e}) \mid \texttt{this} \mid \texttt{return}$$

Here, f is a program field, z a logical variable, and ops an operation on abstract data types, ignoring field access. An *assertion* (of type *Assert*) is a pair of Boolean expressions. Let p' denote an expression p with all occurrences of fields f substituted by f', avoiding name capture. We define entailment for assertions and for sets of assertions:

Definition 1 (Entailment). *Let (p,q) and (r,s) be assertions and let $\mathcal{U}$ and $\mathcal{V}$ denote the assertion sets $\{(p_i,q_i) \mid 1 \leq i \leq n\}$ and $\{(r_i,s_i) \mid 1 \leq i \leq m\}$.* Entailment *is defined by*

1. $(p,q) \rightarrow (r,s) \triangleq (\forall \bar{z}_1 \, . \, p \Rightarrow q') \Rightarrow (\forall \bar{z}_2 \, . \, r \Rightarrow s')$,
   where $\bar{z}_1$ and $\bar{z}_2$ are the logical variables in (p,q) and (r,s), respectively.
2. $\mathcal{U} \rightarrow (r,s) \triangleq (\bigwedge_{1\leq i \leq n}(\forall \bar{z}_i \, . \, p_i \Rightarrow q_i')) \Rightarrow (\forall z \, . \, r \Rightarrow s')$.
3. $\mathcal{U} \rightarrow \mathcal{V} \triangleq \bigwedge_{1 \leq i \leq m} \mathcal{U} \rightarrow (r_i,s_i)$.

Example 2. Let x,y be program variables, and z_1,z_2 logical variables. The assertion $(x = y = z_1, \; x = y = z_1 + 1)$ does not entail $(x = z_2, \; x = z_2 + 1)$, since the implication

$$(\forall z_1 \, . x = y = z_1 \Rightarrow x' = y' = z_1 + 1) \Rightarrow (\forall z_2 \, . x = z_2 \Rightarrow x' = z_2 + 1)$$

does not hold. To illustrate, consider the program $y := y + 1; x := y$.

Note that the relation $\mathcal{U} \rightarrow (r,s)$ corresponds to classic Hoare-style reasoning to prove $\{r\}\; t \; \{s\}$ from $\{p_i\}\; t\; \{q_i\}$ for all $1 \leq i \leq n$, by means of the adaptation and conjunction rules [3] . Note that entailment is reflexive and transitive, and $\mathcal{V} \subseteq \mathcal{U}$ implies $\mathcal{U} \rightarrow \mathcal{V}$.

3.2 Class Analysis with a Proof Environment

We now illustrate the role of the proof environments during class analyses through a series of examples. The environment collects method commitments and requirements in two mappings S and R which, given a class name and method identifier, return a set of assertions. The analysis of a class both uses and changes the proof environment.

Propagation of requirements. Method requirements encountered during the analysis of a proof outline in a class C are verified for the known reachable definitions and imposed on future subclasses. If $m(\bar{x}) : (p,q) \, \{\langle t \rangle\}$ is shown while analyzing C, we extend $S(C,m)$ with (p,q). For each requirement $\{r\}\; n\; \{s\}$ in the proof outline, (r,s) must hold for definitions of n reached by instances of C. Furthermore, $R(C,n)$ is extended with (r,s) as a restriction on future subclass redefinitions of n.

Example 3. Consider the analysis of class C_1 in Fig. 2. The commitment (p_1,q_1) is included in $S(C_1,m)$ and the requirements (r_1,s_1) and (r_2,s_2) are included in $R(C_1,m)$. Both requirements must be verified for the definition of m in C_1, i.e., the definition of m reachable from C_1. Consequently, for each (r_i,s_i), $S(C_1,m) \rightarrow (r_i,s_i)$ must hold, which follows from $(p_1,q_1) \rightarrow (r_i,s_i)$.

In the example, the requirements made by n_1 and n_2 follow from the established commitment of m. Generally, the requirements need not follow from the previously shown commitments. It is then necessary to provide a new proof outline for the method.

Example 4. If (r_i, s_i) does not follow from (p_1, q_1) in Example 3, a new proof outline $m:(r_i, s_i)\,\{\langle t_1\rangle\}$ must be analyzed similarly to the proof outlines in C_1. The mapping $S(C_1, m)$ is extended by (r_i, s_i), ensuring the desired relation $S(C_1, m) \rightharpoonup (r_i, s_i)$.

The analysis strategy must ensure that once a commitment (p, q) is included in $S(C, m)$, it will always hold when the method is executed in an instance of any (future) subclass of C, without reverifying m. In particular, when m is overridden, the *requirements* made by methods in C to m must hold for the new definition of m.

Example 5. Consider class C_2 in Fig. 2, which redefines m. After analysis of the proof outline $m:(p_2, q_2)\,\{\langle t_2\rangle\}$, $S(C_2, m)$ is extended with (p_2, q_2). In addition, the superclass requirements $R(C_1, m)$ must hold for the new definition of m to ensure that the commitments of n_1 and n_2 apply for instances of C_2. Hence, $S(C_2, m) \rightharpoonup (r_i, s_i)$ must be shown for each $(r_i, s_i) \in R(C_1, m)$, similar to $S(C_1, m) \rightharpoonup (r_i, s_i)$ in Example 3.

When a method m is (re)defined in a class C, all superclass invocations of m from instances of C will bind to the new definition. The new definition must therefore support the requirements from all superclasses. Let $R{\uparrow}(C, m)$ denote the union of $R(B, m)$ for all $C \leq B$. For each method m defined in C, it is necessary to ensure the following property:

$$S(C, m) \rightharpoonup R{\uparrow}(C, m) \tag{1}$$

It follows that m must support the requirements from C itself; i.e., $S(C, m) \rightharpoonup R(C, m)$.

Context-dependent properties of inherited methods. Let us now consider methods that are inherited but not redefined, say, m is inherited from a superclass of C. In this case, virtual calls to m from instances of C are bound to the first definition of m above C, but virtual calls *by m* are bound *in the context of C*, as C may redefine methods invoked by m. Furthermore, C may impose new requirements on m not proved during the analysis of the superclass, resulting in new proof outlines for m. In the analysis of the new proof outlines, we know that virtual calls are bound from C. It would be unsound to extend the commitment mapping of the superclass, since the new commitments are only part of the subclass context. Instead, we use $S(C, m)$ and $R(C, m)$ for *local commitment and requirement extensions*. These new commitments and requirements only apply in the context of C and not in the context of its superclasses.

Example 6. Let the following class extend the hierarchy of Fig. 2:

```
class C4 extends C1 {
    n() : (_,_) {...; {r4} this.m() {s4};...}
}
```

Class C_4 inherits the superclass implementation of m. The analysis of n's proof outline yields $\{r_4\}\ m\ \{s_4\}$ as requirement, which is included in $R(C_4, m)$ and verified for the inherited implementation of m. The verification succeeds if $S(C_1,m) \rightarrow (r_4,s_4)$. Otherwise, a new proof outline $m:(r_4,s_4)\{\langle t_1\rangle\}$ is analyzed under the assumption that virtual calls are bound in the context of C_4. When analyzed, (r_4,s_4) becomes a commitment of m and it is included in $S(C_4,m)$. This mapping acts as a local extension of $S(C_1,m)$ and contains commitments of m that hold in the subclass context.

Assume that a definition of m in a class A is reachable from C. When analyzing a requirement $\{r\}\ m\ \{s\}$ in C, we can then rely on $S(A,m)$ and the local extensions of this mapping for all classes between A and C. We assume that programs are type-safe and define a function $S\!\uparrow$ recursively as follows: $S\!\uparrow\!(C,m) \triangleq S(C,m)$ if m is defined in C and $S\!\uparrow\!(C,m) \triangleq S(C,m) \cup S\!\uparrow\!(B,m)$ otherwise, where B is the immediate superclass of C. We can now revise Property 1 to account for *inherited methods*:

$$S\!\uparrow\!(C,m) \rightarrow R\!\uparrow\!(C,m) \tag{2}$$

Thus, each requirement in $R(B,m)$, for some B above C, must follow from the established commitments of m in context C. Especially, for each $(p,q) \in R(C,m)$, (p,q) must either follow from the superclass commitments or from the local extension $S(C,m)$. If (p,q) follows from the local extension $S(C,m)$, we are in the case when a new proof outline has been analyzed in the context of C. Note that Property 2 reduces to Property 1 if m is defined in C.

Analysis of class hierarchies. A class hierarchy is analyzed in a top-down manner, starting with `Object` and an empty proof environment. Classes are analyzed after their respective superclasses, and each class is analyzed without knowledge of possible subclasses. Methods are specified in terms of proof outlines. For each method $m(\overline{x})\{t\}$ defined in a class C, we analyze each (p,q) occurring either as a specification of m in some proof outline, or as an inherited requirement in $R\!\uparrow\!(C,m)$. If $S(C,m) \rightarrow (p,q)$, no further analysis of (p,q) is needed. Otherwise a proof outline $m(\overline{x}):(p,q)\{\langle t\rangle\}$ needs to be analyzed, after which $S(C,m)$ is extended with (p,q). During the analysis of a proof outline, annotated (internal) calls $\{r\}\ n\ \{s\}$ yield requirements (r,s) on reachable implementations of n. The $R(C,n)$ mapping is therefore extended with (r,s) to ensure that future redefinitions of n will support the requirement. In addition, (r,s) is analyzed with respect to the implementation of n that is reached for instances of C; i.e., the first implementation of n above C. This verification succeeds immediately if $S\!\uparrow\!(C,n) \rightarrow (r,s)$. Otherwise, a proof outline for n is analyzed in the context of C, which again extends $S(C,n)$ by (r,s). Each call statement in this proof outline is analyzed in this manner. For *external* calls $\{r\}\ x.m()\ \{s\}$, where x refers to an object of class C', we require that (r,s) follows from the requirements $R\!\uparrow\!(C',m)$ of m in C'.

The mapping S reflects the *definition of methods*; each lookup $S(C,m)$ returns a set of commitments for a particular implementation of m. In contrast, the mapping R reflects the *use of methods* and may impose requirements on several implementations.

Lazy behavioral subtyping. Behavioral subtyping in the traditional sense does *not* follow from the analysis. Behavioral subtyping would mean that whenever a method m is redefined in a class C, its new definition must implement all superclass *commitments* for m; i.e., the method would have to satisfy $S(B,m)$ for all B above C. For example, behavioral subtyping would imply that m in both C_2 and C_3 in Fig. 2 must satisfy (p_1,q_1). Instead, the R mapping identifies the requirements imposed by virtual calls. Only these assertions must be supported by overriding methods to ensure that the execution of superclass' code does not have unexpected results. Thus, only the behavior assumed by the virtual call statements is ensured at the subclass level. In this way, requirements are *inherited by need*, resulting in a lazy form of behavioral subtyping.

Example 7. Consider a class defining two methods which increment counters.

```
class A {
  int x = 0; y = 0
  inc() { x := x + 1; y := y + 1 }
  incX2() { this.inc(); this.inc() }
}
```

Let $(x = z_0, x = z_0 + 2)$ be a commitment of *incX2*, based on a requirement $(x = z_0, x = z_0 + 1)$ to *inc*, included in $R(A, inc)$. If A is later inherited by a class B, B may override *inc*, provided $R(A, inc)$ is supported by the new implementation. The behavior of *incX2* does not depend on other possible commitments in $S(A, inc)$; e.g., $(x = y, x = y)$ and $(y = z_0, y = z_0 + 1)$. In fact, the subclass implementation of *inc* may assign any value to y without breaking the reasoning system.

4 An Assertion Calculus for Program Analysis

The incremental strategy outlined in Sect. 3 is now formalized as a calculus which tracks commitments and requirements for method implementations in an extensible class hierarchy. Given a program, the calculus builds an environment which reflects the class hierarchy and captures method commitments and requirements. This environment forms the context for the analysis of new classes, possibly inheriting already analyzed ones. The proof environment is formally defined in Sect. 4.1, the operations used by the calculus are defined in Sect. 4.2, and the calculus is given as a set of inference rules in Sect. 4.3.

4.1 The Proof Environment of the Assertion Calculus

A class is represented by a tuple $\langle D, \overline{f}, \overline{M} \rangle$ from which the superclass identifier D, fields $\overline{f}$, and methods $\overline{M}$ are accessible by observer functions *inh*, *att*, and *mtds*, respectively. Let $M.body = t$ for a method $M = m(\overline{x})\{t\}$ (or its proof outline). Class names are assumed to be unique, and method names to be unique within a class. The superclass identifier may be *nil*, representing no superclass (for class `Object`).

Definition 2 (Proof environments). *A* proof environment $\mathcal{E}$ *of type Env is a tuple* $\langle P_{\mathcal{E}}, S_{\mathcal{E}}, R_{\mathcal{E}} \rangle$*, where* $P_{\mathcal{E}} : Cid \rightarrow Class$ *is a partial mapping and* $S_{\mathcal{E}}, R_{\mathcal{E}} : Cid \times Mid \rightarrow Set[Assert]$ *are total mappings.*

$$
\begin{array}{ll}
\mathit{bind}_{\mathcal{E}}(\mathit{nil},\, m) & \triangleq \mathit{nil} \\
\mathit{bind}_{\mathcal{E}}(C,\, m) & \triangleq \textbf{if } m \in P_{\mathcal{E}}(C).\mathit{mtds} \textbf{ then } C \textbf{ else } \mathit{bind}_{\mathcal{E}}(P_{\mathcal{E}}(C).\mathit{inh},\, m) \\[1ex]
S{\uparrow_{\mathcal{E}}}(\mathit{nil}, m) & \triangleq \emptyset \\
S{\uparrow_{\mathcal{E}}}(C, m) & \triangleq \textbf{if } m \in P_{\mathcal{E}}(C).\mathit{mtds} \textbf{ then } S_{\mathcal{E}}(C,m) \textbf{ else } S_{\mathcal{E}}(C,m) \cup S{\uparrow_{\mathcal{E}}}(P_{\mathcal{E}}(C).\mathit{inh}, m) \\[1ex]
R{\uparrow_{\mathcal{E}}}(\mathit{nil}, m) & \triangleq \emptyset \\
R{\uparrow_{\mathcal{E}}}(C, m) & \triangleq R_{\mathcal{E}}(C,m) \cup R{\uparrow_{\mathcal{E}}}(P_{\mathcal{E}}(C).\mathit{inh}, m) \\[1ex]
\mathit{body}_{\mathcal{E}}(C,m) & \triangleq P_{\mathcal{E}}(\mathit{bind}_{\mathcal{E}}(C,\, m)).\mathit{mtds}(m).\mathit{body} \\[1ex]
C \leq_{\mathcal{E}} D & \triangleq C = D \vee P_{\mathcal{E}}(C).\mathit{inh} \leq_{\mathcal{E}} D
\end{array}
$$

Fig. 4. Auxiliary function definitions

In an environment $\mathcal{E}$, $P_{\mathcal{E}}$ reflects the class structure, $S_{\mathcal{E}}(C,m)$ the set of commitments for m in C and $R_{\mathcal{E}}(C,m)$ a set of requirements to m from C. For the *empty environment* $\mathcal{E}_{\emptyset}$, $P_{\mathcal{E}_{\emptyset}}(C)$ is undefined and $S_{\mathcal{E}_{\emptyset}}(C,m) = R_{\mathcal{E}_{\emptyset}}(C,m) = \emptyset$ for all $C : \mathit{Cid}$ and $m : \mathit{Mid}$. Let $\leq_{\mathcal{E}}$: $\mathit{Cid} \times \mathit{Cid} \rightarrow \mathit{Bool}$ be the reflexive and transitive subclass relation on $\mathcal{E}$.

Next we define some *auxiliary functions* on proof environments $\mathcal{E}$. Let $\uparrow P_{\mathcal{E}}(C).\mathit{att}$ denote the fields of C and of its superclasses; i.e., the declared fields accessible from methods in C, including the implicit declaration `this` : C. Denote by $\overline{M}(m)$ some proof outline for the method with name m in $\overline{M}$, by $m \in \overline{M}$ that there is a proof outline for m in $\overline{M}$, by $t' \in t$ that the statement t' is contained in the statement t, and by $C \in \mathcal{E}$ that $P_{\mathcal{E}}(C)$ is defined. The function $\mathit{bind}_{\mathcal{E}}(C,m) : \mathit{Cid} \times \mathit{Mid} \rightarrow \mathit{Cid}$ returns the first class above C in which the method m is defined. Assuming type safety, this function will never return *nil*. Let the recursively defined functions $S{\uparrow_{\mathcal{E}}}(C,m)$ and $R{\uparrow_{\mathcal{E}}}(C,m) : \mathit{Cid} \times \mathit{Mid} \rightarrow \mathit{Set}[\mathit{Assert}]$ return all commitments of m both above C and below $\mathit{bind}_{\mathcal{E}}(C,m)$, and all requirements to m that are made by all classes above C in the proof environment $\mathcal{E}$, respectively. Finally, $\mathit{body}_{\mathcal{E}}(C,m) : \mathit{Cid} \times \mathit{Mid} \rightarrow \mathit{Stm}$ returns the body of m in $\mathit{bind}_{\mathcal{E}}(C,m)$. The definitions of these functions are given in Fig. 4.

A *sound environment* reflects that previously analyzed classes are correct. If an assertion appears in $S_{\mathcal{E}}(C,m)$, there must be a verified proof outline M in PL for the corresponding method body. For internal calls $\{r\}\ n\ \{s\}$ in M, (r,s) must be included in $R_{\mathcal{E}}(C,n)$; i.e., all requirements made by the proof outline are in the R-mapping. For external calls $\{r\}\ x.n\ \{s\}$ in M, where x is of class D, the requirement (r,s) must follow from the requirements of n in the context of D. Note that D may be independent of C; i.e., neither above nor below C. Finally, method commitments must entail the requirements (see Property 2 of Sect. 3.2). Sound environments are defined as follows:

Definition 3 (Sound environments). *A* sound environment $\mathcal{E}$ *satisfies the following conditions for all $C : \mathit{Cid}$ and $m : \mathit{Mid}$:*

1. $\forall (p,q) \in S_{\mathcal{E}}(C,m)\ .\ \exists \langle \mathit{body}_{\mathcal{E}}(C,m) \rangle\ .\ \vdash_{\mathrm{PL}} m(\overline{x}) : (p,q)\,\{\langle \mathit{body}_{\mathcal{E}}(C,m) \rangle\}$
$\wedge\ \forall \{r\}\ n\ \{s\} \in \langle \mathit{body}_{\mathcal{E}}(C,m) \rangle\ .\ R_{\mathcal{E}}(C,n) \mathrel{-\!\!\triangleright} (r,s)$
$\wedge\ \forall \{r\}\ x.n\ \{s\} \in \langle \mathit{body}_{\mathcal{E}}(C,m) \rangle\ .\ \exists D\ .\ ((x : D) \in \uparrow P_{\mathcal{E}}(C).\mathit{att}) \Rightarrow R{\uparrow_{\mathcal{E}}}(D,n) \mathrel{-\!\!\triangleright} (r,s)$

2. $S{\uparrow}_{\mathcal{E}}(C,m) \rightarrowtriangle R{\uparrow}_{\mathcal{E}}(C,m)$

Note that in this definition, the proof outline required by Condition 1 need not be in C itself, but may be found above C as described by $body_{\mathcal{E}}(C{,}m)$. Let $\models_C \{p\}\ t\ \{q\}$ denote $\models \{p\}\ t\ \{q\}$ under the assumption that virtual calls in t are bound in the context of C, and let $\models_C m(\overline{x}) : (p,q)\,\{t\}$ be given by $\models_C \{p\}\ t\ \{q\}$. If there are no method calls in t and $\vdash_{\mathrm{PL}} \{p\}\ t\ \{q\}$, then $\models \{p\}\ t\ \{q\}$ follows by the soundness of PL.

Although method redefinitions in a subclass need not respect the commitments of method definitions in superclasses, Lemma 1 and Lemma 2 below ensures that the commitments of method definitions in superclasses will hold when invoked from a subclass, even if auxiliary methods have been redefined.

Lemma 1. *Given a sound environment $\mathcal{E}$ and a sound program logic* PL*. For all C : Cid, m : Mid, and (p,q) : Assert such that $C \in \mathcal{E}$ and $(p,q) \in S{\uparrow}_{\mathcal{E}}(C,m)$, we have $\models_C m(\overline{x}) : (p,q)\,\{body_{\mathcal{E}}(C,m)\}$.*

Proof. By induction on the call structure of m. Since $(p,q) \in S{\uparrow}_{\mathcal{E}}(C,m)$, there must, by Def. 3, Cond. 1, exist some class B such that $C \leq_{\mathcal{E}} B$, $bind_{\mathcal{E}}(C,\, m) = bind_{\mathcal{E}}(B,\, m)$ and $(p,q) \in S_{\mathcal{E}}(B,m)$. Thus, $body_{\mathcal{E}}(C,m) = body_{\mathcal{E}}(B,m)$. In addition, there must be a proof outline $\langle body_{\mathcal{E}}(C,m)\rangle$ for this method such that $\vdash_{\mathrm{PL}} m(\overline{x}) : (p,q)\,\{\langle body_{\mathcal{E}}(C,m)\rangle\}$.

Base case: The execution $\{p\}\ body_{\mathcal{E}}(C,m)\ \{q\}$ does not lead to any method calls. Then $\models_C m(\overline{x}) : (p,q)\,\{body_{\mathcal{E}}(C,m)\}$ follows by the soundness of PL.

Induction step: Assume as induction hypothesis that for each method n called by m and for all $(p',q') \in S{\uparrow}_{\mathcal{E}}(C,n)$, we have $\models_C n(\overline{y}) : (p',q')\,\{body_{\mathcal{E}}(C,n)\}$. Consider an invocation $\{r\}\ n\ \{s\}$ in $\langle body_{\mathcal{E}}(C,m)\rangle$. By Def. 3 Cond. 1 we have $R_{\mathcal{E}}(B,n) \rightarrowtriangle (r,s)$. Then $\models_C \{r\}\ n\ \{s\}$ follows since $S{\uparrow}_{\mathcal{E}}(C,n) \rightarrowtriangle R{\uparrow}_{\mathcal{E}}(C,n)$ by Cond. 2, which especially means $S{\uparrow}_{\mathcal{E}}(C,n) \rightarrowtriangle R_{\mathcal{E}}(B,n)$.

Lemma 2. *Given a sound environment $\mathcal{E}$ and a sound program logic* PL*. For all C : Cid, m : Mid, and (p,q) : Assert such that $C \in \mathcal{E}$ and $(p,q) \in S{\uparrow}_{\mathcal{E}}(C,m)$, we have $\models_D m(\overline{x}) : (p,q)\,\{body_{\mathcal{E}}(C,m)\}$ for each $D \leq_{\mathcal{E}} C$.*

Proof. Again, there exists a class B with the same properties as in the proof of Lemma 1 and we have a proof outline $\vdash_{\mathrm{PL}} m(\overline{x}) : (p,q)\,\{\langle body_{\mathcal{E}}(C,m)\rangle\}$. By Def. 3, Cond. 1, we have $R_{\mathcal{E}}(B,n) \rightarrowtriangle (r,s)$ for each $\{r\}\ n\ \{s\}$ in this proof outline. For any class D below C, we have $S{\uparrow}_{\mathcal{E}}(D,n) \rightarrowtriangle (r,s)$ by Cond. 2. The conclusion then follows by Lemma 1.

In a *minimal* environment $\mathcal{E}$, the mapping $R_{\mathcal{E}}$ only contains requirements that are caused by some proof outline; i.e., there are no superfluous requirements. Minimal environments are defined as follows:

Definition 4 (Minimal Environments). *A sound environment $\mathcal{E}$ is* minimal *iff*

$$\begin{aligned}&\forall (r,s) \in R_{\mathcal{E}}(C,n)\ .\ \exists (p,q), m, \langle body_{\mathcal{E}}(C,m)\rangle\ .\\&\quad (p,q) \in S_{\mathcal{E}}(C,m) \wedge \vdash_{\mathrm{PL}} m(\overline{x}) : (p,q)\,\{\langle body_{\mathcal{E}}(C,m)\rangle\} \wedge \{r\}\ n\ \{s\} \in \langle body_{\mathcal{E}}(C,m)\rangle\end{aligned}$$

Reverification is avoided by incrementally extending $S_{\mathcal{E}}(C,m)$. If a virtual call requires a verified specification, it is found in $S_{\mathcal{E}}(C,m)$. Thus, the avoidance of reverification can be seen as a dual to the first condition to Def. 3: If $\{p\}\ body_{\mathcal{E}}(C,m)\ \{q\}$ is proved, the commitment (p,q) is added to $S_{\mathcal{E}}(C,m)$.

4.2 The Analysis Operations of the Assertion Calculus

An open program may be extended with new classes, and there may be mutual dependencies between the new classes. For example, a method in a new class C can call a method in another new class D, and a method in D can call a method in C. In such cases, a complete analysis of one class cannot be carried out without consideration of mutually dependent classes. We therefore choose class sets as the granularity of program analysis. A *module* is a set of classes, and a module is *self-contained* with regard to an environment $\mathcal{E}$ if all method calls inside the module can be successfully bound inside that module or to classes represented in $\mathcal{E}$.

In the calculus, judgments have the form $\mathcal{E} \vdash \mathcal{A}$, where $\mathcal{E}$ is the proof environment and $\mathcal{A}$ is a list of *analysis operations* on the class hierarchy. The analysis operations have the following syntax:

$$
\begin{array}{l}
O ::= \varepsilon \mid \mathit{analyzeMtds}(\overline{M}) \mid \mathit{verify}(m,\overline{R}) \mid \mathit{analyzeOutline}(t) \mid O \cdot O \\
\mathcal{S} ::= \emptyset \mid L \mid \mathit{require}(C,m,(p,q)) \mid \mathcal{S} \cup \mathcal{S} \\
\mathcal{A} ::= \mathit{module}(\overline{L}) \mid [\langle C : O \rangle ; \mathcal{S}] \mid [\varepsilon ; \mathcal{S}] \mid \mathcal{A} \cdot \mathit{module}(\overline{L})
\end{array}
$$

These analysis operations may be understood as follows. A set $\overline{L}$ of class declarations is analyzed by the module operation $\mathit{module}(\overline{L})$. Classes are assumed to be syntactically well-formed and well-typed. Inside a module, the classes are analyzed in some order, captured by the set $\mathcal{S}$. The operation `class` C `extends` D $\{\overline{f}\ \overline{M}\}$ initiates the analysis of class C. The operation $[\langle C : O \rangle ; \mathcal{S}]$ analyzes O in the context of class C *before* operations in $\mathcal{S}$ are considered. Upon completion, the analysis yields a term of the form $[\varepsilon ; \mathcal{S}]$. The analysis of a specific class consists of the following operations, all inside the context of that class. The operation $\mathit{analyzeMtds}(\overline{M})$ initiates analysis of the proof outlines $\overline{M}$. The operation $\mathit{verify}(m,\overline{R})$ verifies the set $\overline{R}$ of assertions with respect to the method m. The operation $\mathit{analyzeOutline}(t)$ analyzes the method calls in the statement t. Since the operation only occurs in the context of a class C, virtual calls are bound in this context. The operation $\mathit{require}(D,m,(p,q))$ applies to external calls to ensure that m in D satisfies the requirement (p,q). Requirements are lifted outside the context of the calling class C by this operation, and the verification of requirement (p,q) for m in D is shifted into the set of analysis operations $\mathcal{S}$.

4.3 The Inference Rules of the Assertion Calculus

Class modules are analyzed in sequential order such that each module is self-contained with regard to the already analyzed modules. Program analysis is initiated by $\mathcal{E}_{\emptyset} \vdash \mathit{module}(\overline{L})$, where $\overline{L}$ is a module that is self-contained with regard to the empty environment. The analysis of a module is carried out by manipulation of the $\mathit{module}(\overline{L})$ operation according to the inference rules below. During module analysis, the proof environment is extended, keeping track of the currently analyzed class hierarchy and the associated method commitments and requirements. When a *module* operation succeeds, the resulting environment represents a verified class hierarchy. New modules may introduce subclasses of previously analyzed classes, and the calculus is based on an open world assumption as a module may be analyzed in the context of previously analyzed modules and independent of later modules.

There are three different *environment updates*; the loading of a new class L and the extension of the commitment and requirement mappings with an assertion (p,q) for a given method m and class C. These are denoted $\mathit{extS}(C,m,(p,q))$ and $\mathit{extR}(C,m,(p,q))$, respectively. Environment updates are represented by the operator $\oplus : \mathit{Env} \times \mathit{Update} \rightarrow \mathit{Env}$, where the first argument is the current proof environment and the second argument is the environment update, defined as follows:

$$\begin{aligned}
\mathcal{E} \oplus \texttt{class}\, C\, \texttt{extends}\, D\, \{\overline{f}\ \overline{M}\} &= \langle P_{\mathcal{E}}[C \mapsto \langle D, \overline{f}, \overline{M}\rangle], S_{\mathcal{E}}, R_{\mathcal{E}}\rangle \\
\mathcal{E} \oplus \mathit{extS}(C,m,(p,q)) &= \langle P_{\mathcal{E}}, S_{\mathcal{E}}[(C,m) \mapsto S_{\mathcal{E}}(C,m) \cup \{(p,q)\}], R_{\mathcal{E}}\rangle \\
\mathcal{E} \oplus \mathit{extR}(C,m,(p,q)) &= \langle P_{\mathcal{E}}, S_{\mathcal{E}}, R_{\mathcal{E}}[(C,m) \mapsto R_{\mathcal{E}}(C,m) \cup \{(p,q)\}]\rangle
\end{aligned}$$

The corresponding *inference rules* are given in Fig. 5. Note that $\mathcal{A}$ represents a list of modules which will be analyzed later, and which may be empty. Rule (NewModule) initiates the analysis of a new module $\mathit{module}(\overline{L})$. The analysis continues by manipulation of the $[\varepsilon\,;\overline{L}]$ operation that is generated by this rule. For notational convenience, we let $\overline{L}$ denote both a set and list of classes.

Rule (NewClass) selects a new class from the current module, and initiates analysis of the class in the current proof environment. The premises ensure that a class cannot be introduced twice and that the superclass has *already been analyzed*. The class hierarchy is extended with the new class and the analysis continues by traversing the proof outlines by means of the *analyzeMtds* operation. Note that at this point in the analysis, the class has no subclasses in the proof environment. Rule (NewMtd) generates a set of requirement assertions for a method. The requirement set is constructed from the specified commitment of the method and the superclass requirements to the method.

The rules (ReqDer) and (ReqNotDer) address the verification of a particular requirement with respect to a method implementation. If the requirement follows from the commitments of the method, rule (ReqDer) proceeds with the remaining analysis operations. Otherwise, a proof of the requirement is needed. As $\langle \mathit{body}_{\mathcal{E}}(C,m)\rangle$ nondeterministically selects a proof outline, the rule applies to any proof outline for the method available in class C. Remark that (ReqNotDer) is the only rule which extends the S mapping. The considered requirement leads to a new commitment for m with respect to C, and the commitment itself is assumed when analyzing the method body. This captures the standard approach to reasoning about recursive procedure calls [13].

Rule (Call) analyzes the requirement of a local call occurring in some proof outline. The rule extends the R mapping and generates a *verify* operation which analyzes the requirement with respect to the implementation bound from the current class. The extension of the R mapping ensures that future redefinitions of m must respect the requirement; i.e., the requirement applies whenever future redefinitions are considered by (NewMtd). Rule (ExtCall) handles external calls on the form $x.m$ (ignoring field shadowing). The requirement to the external method is removed from the context of the current class and inserted as a *require* operation in $\mathcal{S}$. The class of the callee is found by the declaration of x. Rule (ExtReq) can first be applied *after* the analysis of the callee class, and the requirement must then follow from the requirements of this class.

Rule (EmpClass) concludes the analysis of a class when all analysis operations have succeeded in the context of the class. The analysis of a module is completed by the rule (EmpModule). Thus, the analysis of a module is completed after the analysis of all the module classes and external requirements made by these classes have succeeded.

$$\frac{\mathcal{E} \vdash [\varepsilon ; \overline{L}] \cdot \mathcal{A}}{\mathcal{E} \vdash \mathit{module}(\overline{L}) \cdot \mathcal{A}} \;(\textsc{NewModule})$$

$$\frac{\begin{array}{c} C \notin \mathcal{E} \qquad D \neq \mathit{nil} \Rightarrow D \in \mathcal{E} \\ \mathcal{E} \oplus (\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{f}\ \overline{M}\}) \vdash [\langle C : \mathit{analyzeMtds}(\overline{M})\rangle ; \mathcal{S}] \cdot \mathcal{A} \end{array}}{\mathcal{E} \vdash [\varepsilon ; \{\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{f}\ \overline{M}\}\} \cup \mathcal{S}] \cdot \mathcal{A}} \;(\textsc{NewClass})$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{verify}(m, \{(p,q)\} \cup R\!\uparrow_{\mathcal{E}} (P_{\mathcal{E}}(C).\mathit{inh}, m)) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{analyzeMtds}(m(\overline{x}) : (p,q)\, \{\langle t \rangle\}) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}} \;(\textsc{NewMtd})$$

$$\frac{S\!\uparrow_{\mathcal{E}} (C,m) \rightarrowtriangle (p,q) \qquad \mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{verify}(m,(p,q)) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}} \;(\textsc{ReqDer})$$

$$\frac{\begin{array}{c} \vdash_{\mathrm{PL}} m(\overline{x}) : (p,q)\, \{\langle \mathit{body}_{\mathcal{E}}(C,m)\rangle\} \\ \mathcal{E} \oplus \mathit{extS}(C,m,(p,q)) \vdash [\langle C : \mathit{analyzeOutline}(\langle \mathit{body}_{\mathcal{E}}(C,m)\rangle) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A} \end{array}}{\mathcal{E} \vdash [\langle C : \mathit{verify}(m,(p,q)) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}} \;(\textsc{ReqNotDer})$$

$$\frac{\mathcal{E} \oplus \mathit{extR}(C,m,(p,q)) \vdash [\langle C : \mathit{verify}(m,(p,q)) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{analyzeOutline}(\{p\}\ m\ \{q\}) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}} \;(\textsc{Call})$$

$$\frac{x : D \in\uparrow P_{\mathcal{E}}(C).\mathit{att} \qquad \mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle ; \mathcal{S} \cup \{\mathit{require}(D,m,(p,q))\}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{analyzeOutline}(\{p\}\ x.m\ \{q\}) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}} \;(\textsc{ExtCall})$$

$$\frac{C \in \mathcal{E} \qquad R\!\uparrow_{\mathcal{E}}(C,m) \rightarrowtriangle (p,q) \qquad \mathcal{E} \vdash [\varepsilon ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\varepsilon ; \{\mathit{require}(C,m,(p,q))\} \cup \mathcal{S}] \cdot \mathcal{A}} \;(\textsc{ExtReq})$$

$$\frac{\mathcal{E} \vdash [\varepsilon ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \varepsilon\rangle ; \mathcal{S}] \cdot \mathcal{A}} \;(\textsc{EmpClass}) \qquad\qquad \frac{\mathcal{E} \vdash \mathcal{A}}{\mathcal{E} \vdash [\varepsilon ; \emptyset] \cdot \mathcal{A}} \;(\textsc{EmpModule})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{verify}(m,\emptyset) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}} \;(\textsc{NoReq})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{analyzeMtds}(\emptyset) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}} \;(\textsc{NoMtds})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A} \qquad \textit{t does not contain call statements}}{\mathcal{E} \vdash [\langle C : \mathit{analyzeOutline}(t) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}} \;(\textsc{Skip})$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{verify}(m,\overline{R_1}) \cdot \mathit{verify}(m,\overline{R_2}) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{verify}(m,\overline{R_1}\ \overline{R_2}) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}} \;(\textsc{DecompReq})$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{analyzeOutline}(t_1) \cdot \mathit{analyzeOutline}(t_2) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{analyzeOutline}(t_1 ; t_2) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}} \;(\textsc{DecompCalls})$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{analyzeMtds}(\overline{M_1}) \cdot \mathit{analyzeMtds}(\overline{M_2}) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}}{\mathcal{E} \vdash [\langle C : \mathit{analyzeMtds}(\overline{M_1}\ \overline{M_2}) \cdot \mathcal{O}\rangle ; \mathcal{S}] \cdot \mathcal{A}} \;(\textsc{DecompMtds})$$

Fig. 5. The inference system, where $\mathcal{A}$ is a (possibly empty) list of analysis operations. To simplify the presentation, we let m denote a method call including actual parameters.

In addition, there are some structural rules. The rules (NoReq) and (NoMtds) apply to the empty requirement set and the empty method list, respectively. Rule (Skip) applies to statements which are irrelevant to this analysis. These rules simply continue the analysis with the remaining analysis operations. Finally, the rules (DecompMtds), (DecompReq), and (DecompCalls) flatten non-empty methods lists, requirements sets and statements into separate analysis operations. Note that a proof of $\mathcal{E} \vdash \mathit{module}(\overline{L})$ has exactly one leaf node $\mathcal{E}' \vdash [\varepsilon\,;\emptyset]$; we call $\mathcal{E}'$ the environment resulting from the analysis of $\mathit{module}(\overline{L})$.

Properties of the inference system. Although the individual rules of the inference system do not preserve soundness of the proof environment, the soundness of the proof environment is preserved by the successful analysis of a module. This allows us to prove that the proof system is sound for module analysis.

Theorem 1. *Let $\mathcal{E}$ be a sound environment and $\overline{L}$ a set of class declarations. If a proof of $\mathcal{E} \vdash \mathit{module}(\overline{L})$ has $\mathcal{E}'$ as its resulting proof environment, then $\mathcal{E}'$ is also sound.*

Proof. Assume given a sound environment $\mathcal{E}$. The proof is by induction over the inference rules. For the first condition of sound environments (Def. 3), it suffices to consider rule (ReqNotDer), which asserts $\vdash_{\mathrm{PL}} m(\overline{x}) : (p,q)\,\{\langle \mathit{body}_{\mathcal{E}}(C,m)\rangle\}$ for the extension (p,q) to $S(C,m)$. For all call statements $\{r\}\;n\;\{s\}$ in $\langle \mathit{body}_{\mathcal{E}}(C,m)\rangle$, the rule (Call) ensures $R_{\mathcal{E}}(C,n) \rightarrowtriangle (r,s)$. For all $\{r\}\;x.m\;\{s\}$ where $x : D$, we must have $R_{\mathcal{E}}(D,n) \rightarrowtriangle (r,s)$ by the rules (ExtCall) and (ExtReq), since the operation $\mathit{module}(\overline{L})$ succeeds.

The second condition of sound environments can be proved by induction on the height of the class hierarchy, starting with classes without superclasses.

Base case: Consider a class C such that $P_{\mathcal{E}}(C).\mathit{inh} = \mathit{nil}$. The mapping $R_{\mathcal{E}}(C,m)$ is initially empty so if (p,q) is in $R_{\mathcal{E}}(C,m)$, the rule (Call) must have been applied adding the analysis operation $\mathit{verify}(m,(p,q))$ within the context of C. Since this operation succeeds, either (ReqDer) or (ReqNotDer) is applied. The desired relation $S{\uparrow}_{\mathcal{E}}(C,m) \rightarrowtriangle (p,q)$ must hold if (ReqDer) is applied. If (ReqNotDer) is applied, the mapping $S_{\mathcal{E}}(C,m)$ is extended with (p,q), ensuring $S{\uparrow}_{\mathcal{E}}(C,m) \rightarrowtriangle (p,q)$.

Induction step: We consider a class C of height $n+1$. For all classes C' at height $\leq n$, we get $S{\uparrow}_{\mathcal{E}}(C',m) \rightarrowtriangle R{\uparrow}_{\mathcal{E}}(C',m)$ as induction hypothesis. Since the immediate superclass of C is at height n, we may assume $S{\uparrow}_{\mathcal{E}}(P_{\mathcal{E}}(C).\mathit{inh},m) \rightarrowtriangle R{\uparrow}_{\mathcal{E}}(P_{\mathcal{E}}(C).\mathit{inh},m)$. There are two cases, depending on whether m is defined in C or not. If $m \notin P_{\mathcal{E}}(C).\mathit{mtds}$ then $S{\uparrow}_{\mathcal{E}}(P_{\mathcal{E}}(C).\mathit{inh},m) \subseteq S{\uparrow}_{\mathcal{E}}(C,m)$, giving $S{\uparrow}_{\mathcal{E}}(C,m) \rightarrowtriangle R{\uparrow}_{\mathcal{E}}(P_{\mathcal{E}}(C).\mathit{inh},m)$ by the induction hypothesis. If $m \in P_{\mathcal{E}}(C).\mathit{mtds}$, the method is analyzed with the rule (NewMtd), leading to a *verify* operation on each requirement in $R{\uparrow}_{\mathcal{E}}(P_{\mathcal{E}}(C).\mathit{inh},m)$. The analysis of these *verify* operations ensures that $S_{\mathcal{E}}(C,m) \rightarrowtriangle R{\uparrow}_{\mathcal{E}}(P_{\mathcal{E}}(C).\mathit{inh},m)$. Consequently, we have $S{\uparrow}_{\mathcal{E}}(C,m) \rightarrowtriangle R{\uparrow}_{\mathcal{E}}(P_{\mathcal{E}}(C).\mathit{inh},m)$ also in this case since $S{\uparrow}_{\mathcal{E}}(C,m) = S_{\mathcal{E}}(C,m)$. In both cases we additionally need to ensure $S{\uparrow}_{\mathcal{E}}(C,m) \rightarrowtriangle R_{\mathcal{E}}(C,m)$. This proof is similar to the base case.

Theorem 2 (Soundness). *If* PL *is a sound program logic, then the derived proof outline logic combined with the calculus also constitutes a sound proof system.*

Proof. It follows directly from the definition of sound environments that the empty environment is sound. Theorem 1 and Lemma 2 guarantee that the environment remains sound during analysis of modules.

Furthermore, the inference system preserves minimality of proof environments; i.e., only requirements needed by some proof outline are recorded in the $R_{\mathcal{E}}$ mapping.

Lemma 3. *If $\mathcal{E}$ is a minimal environment and $\overline{L}$ is a set of class declarations such that a proof of $\mathcal{E} \vdash module(\overline{L})$ leads to the resulting environment $\mathcal{E}'$, then $\mathcal{E}'$ is also minimal.*

Proof. By induction over the inference rules. For a class C and method m, the rule (CALL) is the only rule that extends $R_{\mathcal{E}}(C,m)$. In order for the rule to be applied, an operation $analyzeOutline(\{p\}\ m\ \{q\})$ must be analyzed in the context of C for some requirement (p,q) to m. This operation can only have been generated by an application of (REQNOTDER), which guarantees that the requirement is needed by some analyzed proof outline.

Finally we show that the proof system supports verification reuse in the sense that commitments are remembered.

Lemma 4. *Let $\mathcal{E}$ be an environment $\mathcal{E}$ and $\overline{L}$ a list of class declarations. Whenever a proof outline $m(\overline{x}) : (p,q)\,\{\langle t\rangle\}$ is verified during analysis of some class C in $\overline{L}$, the commitment (p,q) is included in $S_{\mathcal{E}}(C,m)$.*

Proof. By induction over the inference rules. A commitment is verified whenever a proof outline $m(\overline{x}) : (p,q)\,\{\langle body_{\mathcal{E}}(C,m)\rangle\}$ is verified in PL. The only rule requiring the verification of such a proof outline is (REQNOTDER), so it suffices to consider this rule. From the premises of (REQNOTDER) it follows that $S_{\mathcal{E}}(C,m)$ is extended with (p,q) whenever $\vdash_{\mathrm{PL}} m(\overline{x}) : (p,q)\,\{\langle body_{\mathcal{E}}(C,m)\rangle\}$ is established.

5 Related Work

Object-orientation poses several challenges to program logics; e.g., inheritance, late binding, recursive method calls, aliasing, and object creation. In the last years several programming logics have been proposed, addressing various of these challenges. For example, object creation has been addressed by means of specialized allocation predicates [1] or by encoding heap information in sequences [9]. Numerous proof methods, verification condition generators, and validation environments for object-oriented languages have been developed, including [1] [23] [22] [14] [16] [6]. In particular, Java has attracted much interest, with advances being made for different (mostly sequential) aspects and sublanguages of that language. In particular, most such formalizations concentrate on closed systems. A recent state-of-the-art survey of challenges and results for proof systems and verification in the field is given in [18], and for an overview of verification tools based on the Java modeling language JML, see [7].

Proof systems especially studying late bound methods have been shown to be sound and complete assuming a closed world [25]. While this is proof-theoretically satisfactory, the closed world assumption is unrealistic in practice and necessitates costly reverification when the class hierarchy is extended (as discussed in Sect. 1). To support object-oriented design, proof systems should be constructed for incremental reasoning. Most prominent in that context are different variations of behavioral subtyping [19, 20, 27]. Virtual methods [26] similarly allow incremental reasoning by committing to certain abstract properties about a method, which must hold for all its implementations. Although sound, the approach does not generally provide complete program logics, as these abstract properties would, in non-trivial cases, be too weak to obtain completeness without over-restricting method redefinition from the point of view of the programmer. Virtual methods furthermore force the developer to commit to specific abstract specifications of method behavior early in the design process. This seems overly restrictive and lead to less reasoning modularity than the approach as such suggests. In particular, the verification platforms for *Spec#* [5] and JML [7] rely on versions of behavioral subtyping.

The fragile base class problem emerges when seemingly harmless superclass updates lead to unexpected behavior of subclass instances [21]. Many variations of the problem relate to imprecise specifications and assumptions made in super- or subclasses. By making method requirements and assumptions explicit, our calculus can detect many issues related to the fragile base class problem. Subclasses can only rely on requirements made explicit in the requirement property set of the class. Updates in the superclass must respect these assumptions.

6 Conclusion

This report presents lazy behavioral subtyping, a novel strategy for reasoning about late bound method calls. The strategy is designed to support incremental reasoning and avoid reverification in an open setting, where class hierarchies can be extended by inheritance. Lazy behavioral subtyping is more flexible than strategies based on traditional behavioral subtyping, while retaining the open world assumption. To focus the presentation, we have abstracted from many object-oriented language features and presented the approach for an object-oriented kernel language supporting single inheritance. This reflects the mainstream object-oriented languages today, such as Java and C#.

We currently integrate lazy behavioral subtyping in a program logic for Creol [10, 17], a language for dynamically reprogrammable active objects, developed in the context of the European project Credo. This integration requires a generalization of the analysis to *multiple inheritance* and concurrent objects, as well as to Creol's mechanism for *class upgrades*. Moreover an adaptation is needed to Creol's type system, which is purely based on interfaces. Interface types provide a clear distinction between internal and external calls. By separating interface level subtyping from class level inheritance, class inheritance can freely exploit code reuse based on lazy behavioral subtyping while still supporting incremental reasoning techniques. This program logic with lazy behavioral subtyping will be part of the programming environment for Creol, based on Eclipse.

References

1. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In N. Dershowitz, editor, *Verification: Theory and Practice, Essays Dedicated to Zohar Manna*, volume 2772 of *LNCS*, pages 11–41. Springer, 2003.

2. P. America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 60–90. Springer, 1991.
3. K. R. Apt. Ten years of Hoare's logic: A survey — Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, Oct. 1981.
4. K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Systems*. Texts and Monographs in Computer Science. Springer, 1991.
5. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Intl. Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
6. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *LNAI*. Springer, 2007.
7. L. Burdy, Y. Cheon, D. R. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, , and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Proceedings of FMICS '03*, volume 80 of *ENTCS*. Elsevier Science Publishers, 2003.
8. O.-J. Dahl, B. Myhrhaug, and K. Nygaard. (Simula 67) Common Base Language. Technical Report S-2, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, May 1968.
9. F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Proceedings of Foundations of Software Science and Computation Structure, (FOSSACS'99)*, volume 1578 of *LNCS*, pages 135–149. Springer, 1999.
10. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer-Verlag, Mar. 2007.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.
12. C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.
13. C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium On Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer, 1971.
14. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.
15. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
16. B. Jacobs and E. Poll. A logic for the Java Modelling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering*, volume 2029 of *LNCS*, pages 284–299. Springer, 2001.
17. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
18. G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.
19. G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20a, Department of Computer Science, Iowa State University, Ames, Iowa, 2006.
20. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.
21. L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In E. Jul, editor, *12th European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *LNCS*, pages 355–382. Springer, 1998.

22. D. v. Oheimb. *Analysing Java in Isabelle/HOL: Formalization, Type Safety, and Hoare-Logics*. PhD thesis, Technische Universität München, 2001.
23. D. v. Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects, and virtual methods revisited. In L.-H. Eriksson and P. A. Lindsay, editors, *Formal Methods Europe (FME 2002)*, volume 2391 of *LNCS*, pages 89–105. Springer, 2002.
24. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.
25. C. Pierik and F. S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Computer Science*, 343(3):413–442, 2005.
26. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *8th European Symposium on Programming Languages and Systems (ESOP'99)*, volume 1576 of *LNCS*, pages 162–176. Springer, 1999.
27. N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proc. Fifth International Conference on Software Reuse (ICSR5)*, pages 206–215. IEEE Computer Society Press, 1998.