# Lazy Behavioral Subtyping[☆]

Johan Dovland[a,*], Einar Broch Johnsen[a], Olaf Owe[a], Martin Steffen[a]

[a] *Department of Informatics, University of Oslo, Norway*

**Abstract**

Inheritance combined with 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 paper develops a novel method to reason about late bound method calls. In contrast to traditional behavioral subtyping, reverification of method specifications 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 paper use a Hoare style proof system. We show soundness of the analysis method. The paper finally demonstrates how lazy behavioral subtyping can be combined with interface specifications to produce an incremental and modular reasoning system for object-oriented class hierarchies.

*Key words:* object orientation, inheritance, code reuse, late binding, proof systems, method redefinition, incremental reasoning, behavioral subtyping

[☆] This work was done the context of the EU projects IST-33826 *CREDO: Modeling and analysis of evolutionary structures for distributed services* (`http://credo.cwi.nl`) and FP7-231620 *HATS: Highly Adaptable and Trustworthy Software using Formal Models* (`http://www.hats-project.eu`).

[*] Corresponding author.

*Email addresses:* `johand@ifi.uio.no` (Johan Dovland), `einarj@ifi.uio.no` (Einar Broch Johnsen), `olaf@ifi.uio.no` (Olaf Owe), `msteffen@ifi.uio.no` (Martin Steffen)

*Preprint submitted to Elsevier* *August 25, 2010*

## 1. Introduction

Inheritance and late binding of method calls are central features of object-oriented languages and contribute to flexible *code reuse*. A class may extend its superclasses with new methods, possibly overriding existing ones. This flexibility comes at a price: It significantly complicates reasoning about method calls as the behavior of a method call depends on the selected code and the binding of a 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. Class extensions will lead to new potential bindings for overridden methods. Thus, inherited methods may change behavior due to internal calls.

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., [47]). 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, 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 in order to avoid reverification, 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 *specification*. Best known as *behavioral subtyping* (e.g., [38, 39, 4, 5, 36, 48]), this strategy achieves incremental reasoning by limiting the possibilities for method overriding, and thereby code reuse. Once a specification is given for a method, this specification must be respected by later redefinitions. However, behavioral subtyping has been criticized for being overly restrictive and often

violated in practice [49].

A main difficulty with behavioral subtyping is that a strong class specification limits method overriding in subclasses, while a weak class specification limits reasoning. Thus, when writing a class specification one should think of all future code reuse in subclasses. This conflicts with the open world assumption. Another problem is that when reusing a class which only has a weak specification, one must look at the actual code to find out what the class does. The strategy of *lazy behavioral subtyping*, introduced in this paper, relaxes the restriction to property preservation which applies in behavioral subtyping, while embracing the open world assumption of incremental program development. A class may well be given a strong specification, while the properties to be preserved by subclasses are in general weaker, ensuring that internal calls are correct. The strong specification reduces the need for code inspection. The central idea is as follows: given a method $m$ specified by a precondition $p$ and a postcondition $q$, 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 $q$ that is actually *used to verify* the program at the current stage. Specifically, if $m$ is used in the program in the form of an internal method call $\{r\}\ m(\ldots)\ \{s\}$, the pre- and postconditions $r$ and $s$ at that call-site constitute $m$'s *required* behavior. Observe that the requirements are weaker than the specifications, and it is in fact these weaker requirements that need to be preserved by subclass overridings in order to avoid reverification. We therefore call the corresponding analysis strategy lazy behavioral subtyping.

**Example 1.** Consider the following two classes:

```
class Account {
  int bal;
  void deposit(nat x) {update(x)}
  void withdraw(nat x) {update(-x)}
  void update(int x) {bal := bal + x}
}
class FeeAccount extends Account {
  int fee;
  void withdraw(nat x) {update(-(x+fee))}
}
```

In this example, class *Account* implements ideal bank accounts for which the *withdraw* method satisfies the pre- and postcondition pair $(bal = bal_0, bal =$

$bal_0 - x$), where $bal_0$ is a logical variable used to capture the initial value of $bal$. The subclass *FeeAccount* redefines the *withdraw* method, charging an additional *fee* for each withdrawal. Thus, class *FeeAccount* is not a *behavioral subtype* of class *Account*. However, the example illustrates that it might be fruitful to implement *FeeAccount* as an extension of *Account* since much of the existing code can be *reused* by the subclass. In this paper we focus on incremental reasoning in this setting: Subclasses may reuse and override superclass code in a flexible manner such that superclass specifications need not be respected.

The paper formalizes the lazy behavioral subtyping analysis strategy using an object-oriented kernel language, based on Featherweight Java [29], 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 internal calls in the known class hierarchy. The strategy is incremental; for the analysis of a class $C$, only knowledge of $C$ and its superclasses is needed. We first present a simple form of the calculus, previously published in [21], in order to focus on the mechanics of the lazy behavioral subtyping inference system. In the present paper, the soundness proofs are given for this calculus. Although this system ensures that old proofs are never violated, external calls may result in additional proof obligations in a class which has already been analyzed. As a consequence, it may be necessary to revisit classes at a later stage in the program analysis. To improve this situation, we extend [21] by considering a refined version of the calculus which introduces behavioral interfaces to encapsulate objects. This refined calculus is, in our opinion, more practical for real program analysis and a better candidate for implementation. The behavioral constraints of the interface implemented by a class become proof obligations for that class, and external calls are verified against the behavioral constraints of the interface. As a result, the refined calculus is both *incremental* and *modular*: Each class is analyzed once, after its superclasses, ensuring that verified properties of superclasses are not violated, and external calls are analyzed based on interface constraints. Inherited code is analyzed in the context of the subclass only when new properties are needed. A subclass need not implement the interface of a superclass, thereby allowing code to be reused freely by the subclass without satisfying the behavioral constraints of the superclass. The lazy behavioral subtyping strategy may

$$
\begin{array}{lll}
P & ::= & \overline{L}\ \{t\} \\
L & ::= & \textbf{class}\ C\ \textbf{extends}\ C\ \{\overline{F}\ \overline{M}\ \overline{MS}\} \\
M & ::= & [T \mid \textbf{void}]\ m(\overline{T\ x})\{t\} \\
MS & ::= & [T \mid \textbf{void}]\ m(\overline{T\ x}) : (p, q) \\
F & ::= & T\ f\ [=\ [e \mid \textbf{new}\ C]]^? \\
T & ::= & \textbf{nat} \mid \textbf{int} \mid \textbf{bool} \mid C \\
t & ::= & v := \textbf{new}\ C \mid v := e.m(\overline{e}) \mid v := m(\overline{e}) \mid v := m@C(\overline{e}) \mid v := e \\
 & & \mid \textbf{skip} \mid \textbf{if}\ b\ \textbf{then}\ t\ \textbf{else}\ t\ \textbf{fi} \mid t; t \\
v & ::= & f \mid \textbf{return} \\
e & ::= & v \mid x \mid \textbf{this} \mid \textbf{true} \mid \textbf{false} \mid \mathbb{N} \mid op(\overline{e})
\end{array}
$$

Figure 1: *Syntax for the language OOL.* Here $C$ and $m$ are class and method names (of types *Cid* and *Mid*, respectively). Assignable program variables $v$ include field names $f$ and the reserved variable **return** for return values. The expression $op(\overline{e})$ denotes operations over integer and Boolean expressions ($b$). The notation $[A \mid B]$ denotes a choice between $A$ and $B$, and $[A]^?$ denotes that $A$ is optional.

serve as a blueprint for integrating a flexible system for program verification of late bound method calls into environments for object-oriented program development and analysis tools (e.g., [8, 9, 11]).

*Paper overview.* Section 2 introduces the problem of reasoning about late binding, Section 3 presents the lazy behavioral subtyping approach developed in this paper, and Section 4 formalizes the inference system. Section 5 extends the inference system with interface encapsulation. The extended system is illustrated by an example in Section 6. Related work is discussed in Section 7 and Section 8 concludes the paper.

## 2. Late Bound Method Calls

### 2.1. Syntax for an Object-Oriented Kernel Language OOL

To succinctly explain late binding and our analysis strategy, we use an object-oriented kernel language with a standard type system and operational semantics (e.g., similar to that of Featherweight Java [29] and Creol [33]). The language is named *OOL*, and the syntax is given in Figure 1. We assume a functional language of side-effect free expressions $e$, including primitive value types for (unbounded) integers and Booleans. Overbar notation denotes possibly empty lists; e.g., $\overline{e}$ is a list of expressions. A program $P$ consists of a list $\overline{L}$ of class definitions, followed by a method body $t$. A class extends a superclass, possibly the top class **Object**, with definitions of fields

$\overline{F}$, methods $\overline{M}$, and method specifications $\overline{MS}$. For simplicity, we assume that fields have distinct names, that methods with the same name have the same signature (i.e., method overriding is allowed but not overloading), and that programs are type-sound so method binding succeeds. The reserved variable **this** for self reference is read-only. For classes $B$ and $C$, $C \leq B$ denotes the reflexive and transitive *subclass relation* derived from class inheritance. If $C \leq B$, we say that $C$ is *below* $B$ and $B$ is *above* $C$. Two classes are *independent* if one is *not* below the other.

A method $M$ takes formal parameters $\overline{T\ x}$ and contains a statement $t$ as its method body where $\overline{x}$ are read-only. 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 $v$. (In *OOL*, a possible constructor method in the class must be called explicitly.) There are standard statements for **skip**, conditionals **if** $b$ **then** $t$ **else** $t$ **fi**, and assignments $v := e$. To simplify the presentation we do not consider explicit loop constructs, but allow recursion. Furthermore, we disallow external field access in order to focus the discussion on method calls and to simplify the extension of *OOL* with behavioral interfaces in Section 5.

*OOL* distinguishes syntactically between *internal late bound calls*, *internal static calls*, and *external calls*. For an internal late bound call $m(\overline{e})$, the method $m$ is executed on **this** with actual parameters $\overline{e}$. The call is bound at run-time depending on the actual class of the object. An internal static call $m@C(\overline{e})$ may occur in a class below $C$, and it is bound at compile time to the first matching definition of $m$ above $C$. This statement generalizes super calls, as found in e.g., Java. In an external method call $e.m(\overline{e})$, the object $e$ (which may be **this**) receives a call to the method $m$ with actual parameters $\overline{e}$. The statements $v := m(\overline{e})$, $v := m@C(\overline{e})$, and $v := e.m(\overline{e})$ assign the value of the method activation's **return** variable to $v$. If $m$ does not return a value, or if the returned value is of no concern, we sometimes use $e.m(\overline{e})$ or $m(\overline{e})$ directly as statements for simplicity. Note that the list $\overline{e}$ of actual parameter values may be empty.

In *OOL*, object references are typed by class names. We assume a static type system, and let $e : E$ denote that $E$ is the static type of expression $e$. For an external call $e.m(\overline{e})$, where $e : E$, the call can be bound to an instance of class $E$ or a subclass of $E$. A method specification $T'\ m(\overline{T\ x}) : (p, q)$ defines a pre/post specification $(p, q)$ of the method $m$. For convenience, we let $T'\ m(\overline{T\ x}) : (p, q)\{t\}$ abbreviate the combination of the definition

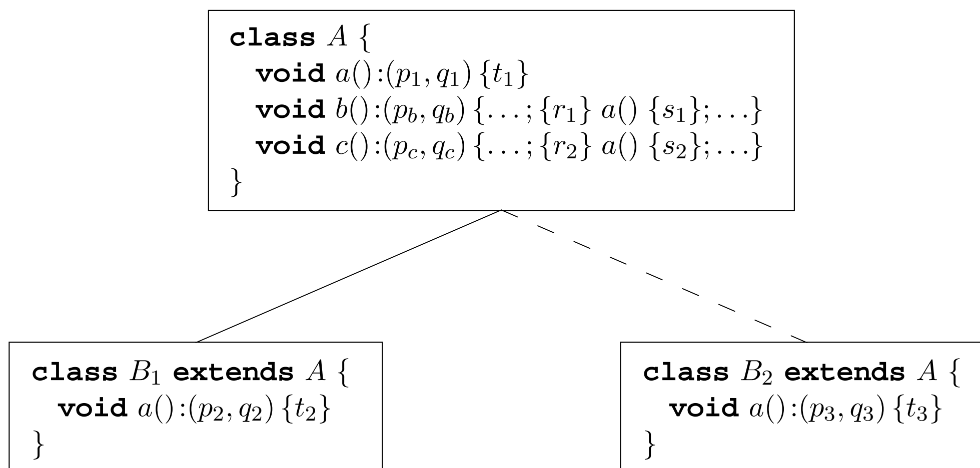


Figure 2: Example of a class hierarchy where the method definitions are decorated with assertions in the style of proof outlines.

$T'\ m(\overline{T\ x})\{t\}$ and the specification $T'\ m(\overline{T\ x}) : (p, q)$. Specifications of a method $m$ may be given in a class where $m$ is defined or in a subclass. Notice that even if $m$ is not redefined, independent subclasses may well have conflicting specifications of $m$ (see Example 10).

### *2.2. Late Binding*

Late binding (or dynamic dispatch), already present in Simula [15], is a central concept of object-orientation. A method call is late bound if the method body is selected at run-time, depending on the callee's actual class. Late bound calls are bound to the first implementation found above the actual class. For a class **class** $C$ **extends** $B$ $\{\overline{F}\ \overline{M}\ \overline{MS}\}$ we recursively define the partial function *bind* for binding late bound calls, by

$$bind(C, m) \triangleq \textbf{if } m \in \overline{M} \textbf{ then } C \textbf{ else } bind(B, m) \textbf{ fi},$$

where $m \in \overline{M}$ denotes that an implementation of $m$ is found in $\overline{M}$. Thus, $bind(C, m)$ returns the first class above $C$ where a definition of $m$ is found. Assuming type safety, this function is always well-defined. We say that an implementation of $m$ in $A$ is *visible* from $C$ if $bind(C, m) = A$. In this case, a late bound call to $m$ on an instance of $C$ will bind to the definition in $A$. Late binding is illustrated in Figure 2 in which a class $A$ and two independent

subclasses $B_1$ and $B_2$ are defined: an object of class $B_1$ executes an inherited method $b$ defined in its superclass $A$ and this method issues a call to method $a$ defined in both classes $A$ and $B_1$. With late binding, the code selected for execution is associated with the first matching $a$ above $B_1$; i.e., as the calling object is an instance of class $B_1$, the method $a$ of $B_1$ is selected and not the one of $A$. If, however, method $b$ were executed in an instance of $A$, the late bound invocation of $a$ would be bound to the definition in $A$. Late binding is central to object-oriented programming and especially underlies many of the well-known object-oriented design patterns [24].

For a late bound internal call to $m$, made by a method defined in class $C$, we say that a definition of $m$ in class $D$ is *reachable* if the definition in $D$ is visible from $C$, or if $D$ is a subclass of $C$. As $m$ may be overridden by any subclass of $C$, there may be several reachable definitions for a late bound call statement. For the calls to $a$ in class $A$ in Figure 2, the definitions of $a$ in $A$, $B_1$, and $B_2$ are all reachable. At run-time, one of the reachable definitions is selected based on the actual class of the called object. Correspondingly, for an external call $e.m()$ where $e : E$, a definition of $m$ in class $D$ is reachable if the definition is visible from $E$ or if $D$ is a subclass of $E$.

*2.3. Proof Outlines*

Apart from the treatment of late bound method calls, our initial reasoning system follows standard proof rules [6, 7] 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 [16]. We present the proof system using Hoare triples $\{p\}\ t\ \{q\}$ [25], where $p$ is the precondition and $q$ is the postcondition to the statement $t$. Triples $\{p\}\ t\ \{q\}$ have a standard partial correctness semantics: if $t$ is executed in a state where $p$ holds and the execution terminates, then $q$ holds after $t$ has terminated. The derivation of triples can be done in any suitable program logic. Let $PL$ be such a program logic and let $\vdash_{PL} \{p\}\ t\ \{q\}$ denote that $\{p\}\ t\ \{q\}$ is derivable in $PL$. A *proof outline* [44] for a method definition $T'\ m(\overline{T\ x})\{t\}$ is the method body decorated with assertions. For the purpose of this paper, we are mainly interested in method calls decorated with pre- and postconditions.

Let the notation $O \vdash_{PL} t : (p, q)$ denote that $O$ is an outline proving that the *specification* $(p, q)$ holds for a body $t$; i.e., $\vdash_{PL} \{p\}\ O\ \{q\}$ holds when assuming that the pre- and postconditions provided in $O$ for the method calls contained in $t$ are correct. The pairs of pre- and postconditions for these method calls are called *requirements*. Thus, for a decorated call $\{r\}\ n()\ \{s\}$

$$(\textsc{Assign})\ \{q[e/v]\}\ v := e\ \{q\}$$

$$(\textsc{New})\ \{q[\mathbf{fresh}_C/v]\}\ v := \mathbf{new}\ C()\ \{q\}$$

$$(\textsc{Skip})\ \{q\}\ \mathbf{skip}\ \{q\}$$

$$(\textsc{Seq})\ \frac{\{p\}\ t_1\ \{r\} \quad \{r\}\ t_2\ \{q\}}{\{p\}\ t_1; t_2\ \{q\}}$$

$$(\textsc{Cond})\ \frac{\{p \wedge b\}\ t_1\ \{q\} \qquad \{p \wedge \neg b\}\ t_2\ \{q\}}{\{p\}\ \mathbf{if}\ b\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{fi}\ \{q\}}$$

$$(\textsc{Adapt})\ \frac{p \Rightarrow p_1 \quad \{p_1\}\ t\ \{q_1\} \quad q_1 \Rightarrow q}{\{p\}\ t\ \{q\}}$$

$$(\textsc{LateCall})\ \frac{e : E \qquad \forall i \in \mathbf{reachable}(E, m) \cdot \{p_i\}\ body^i_{m(\overline{x})}\ \{q_i\}}{\{\bigwedge_i (p_i[\overline{e}/\overline{x}]) \wedge \overline{e} = \overline{e_0}\}\ v := e.m(\overline{e})\ \{\bigvee_i (q_i[\overline{e_0}, v/\overline{x}, \mathbf{return}])\}}$$

$$(\textsc{StatCall})\ \frac{\{p\}\ body^{bind(C,m)}_{m(\overline{x})}\ \{q\}}{\{p[\overline{e}/\overline{x}] \wedge \overline{e} = \overline{e_0}\}\ v := m@C(\overline{e})\ \{q[\overline{e_0}, v/\overline{x}, \mathbf{return}]\}}$$

Figure 3: *Closed world proof rules.* Let $p[e/v]$ denote the substitution of all occurrences of $v$ in $p$ by $e$ [25], extended for object creation by introducing unused symbols $\mathbf{fresh}_C$ following [47]. The function $\mathbf{reachable}(E, m)$ returns the set of classes statically reachable for the call $e.m$ (as explained above). For closed world systems this covers all possible definitions of $m$ chosen by late binding. The body of $m$ for class $i$ is given by $body^i_{m(\overline{x})}$, where $\overline{x}$ is the formal parameter list. The logical variable list $\overline{e_0}$ (assumed disjoint with other variables) is used to formalize that parameters are read-only.

in $O$, $(r, s)$ is a requirement for $n$. In order to ensure that this requirement is satisfied, every reachable definition of $n$ must be analyzed (including definitions which may appear in future subclasses).

### *2.4. Reasoning about Late Bound Calls in Closed Systems*

If the proof system assumes a closed world, all classes must be defined before the analysis can begin because the requirement to a method call is derived from the specifications of all reachable implementations of that method. To simplify the presentation in this paper, we omit further details of the assertion language and the proof system (e.g., ignoring the representation of the program semantics — for details see [47]). The corresponding proof system is given in Figure 3; the proof rule (LateCall) captures late binding under a closed world assumption. In this system external and internal calls can be analyzed in the same manner, taking all possible reachable definitions into

account: An internal call $v := m(\overline{e})$ is analyzed like $v :=$ **this**$.m(\overline{e})$. The proof rule for a static call $m@C(\overline{e})$ is simpler: Since it is bound by $bind(C, m)$, only this definition needs to be taken into account. The following example illustrates the proof system.

**Example 2.** Consider the class hierarchy of Figure 2, where the methods are decorated with proof outlines. Let $(r_1, s_1)$ and $(r_2, s_2)$ be the requirements for method $a$ imposed by the proof outlines for the given specifications $(p_b, q_b)$ and $(p_c, q_c)$ of methods $b$ and $c$, respectively. Assume $O_1 \vdash_{PL} t_1 : (p_1, q_1)$, $O_2 \vdash_{PL} t_2 : (p_2, q_2)$, and $O_3 \vdash_{PL} t_3 : (p_3, q_3)$ for the definitions of $a$ in classes $A$, $B_1$, and $B_2$, respectively. Consider initially the class hierarchy consisting of $A$ and $B_1$ and ignore $B_2$ for the moment. The proof system of Figure 3 gives the Hoare triple $\{p_1 \wedge p_2\}\ a()\ \{q_1 \vee q_2\}$ for each call to $a$, i.e., for the calls in the bodies of methods $b$ and $c$ in class $A$. In order 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 $b$, and $r_2 \Rightarrow p_1 \wedge p_2$, and $q_1 \vee q_2 \Rightarrow s_2$ for $c$. If the class hierarchy is now *extended* with $B_2$, the closed world assumption breaks and the methods $b$ and $c$ need to be *reverified*. With the new Hoare triple $\{p_1 \wedge p_2 \wedge p_3\}\ a()\ \{q_1 \vee q_2 \vee q_3\}$ at every call-site, the proof obligations given above for applying (Adapt) no longer apply.

## 3. A Lazy Approach to Incremental Reasoning

This section informally presents the approach of lazy behavioral subtyping. Based on an open world assumption, lazy behavioral subtyping supports incremental reasoning about extensible class hierarchies. The approach is oriented towards reasoning about late bound calls and is well-suited for program development, being less restrictive than behavioral subtyping. A formal presentation of lazy behavioral subtyping is given in Section 4.

To illustrate the approach, first reconsider class $A$ in Figure 2. The analysis for methods $b$ and $c$ requires that $\{r_1\}\ a()\ \{s_1\}$ and $\{r_2\}\ a()\ \{s_2\}$ hold for the internal calls to $a$ in the bodies of $b$ and $c$, respectively. The assertion pairs $(r_1, s_1)$ and $(r_2, s_2)$ may be seen as *requirements* to all reachable definitions of $a$. Consequently, for $a$'s definition in $A$, both $\{r_1\}\ t_1\ \{s_1\}$ and $\{r_2\}\ t_1\ \{s_2\}$ must hold. Compared to Example 2, the proof obligations for method calls have shifted from the call to the definition 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 $B_1$ or $B_2$, the *same requirements* apply to their definition of $a$. Thus, no reverification of the bodies of $b$ and $c$ is needed when new subclasses are analyzed.

Although $A$ is analyzed independently of $B_1$ and $B_2$, its requirements must be considered during the analysis of the subclasses. For this purpose, a *proof environment* is constructed and maintained during the analysis. While analyzing $A$, it is recorded in the proof environment that $A$ requires both $(r_1, s_1)$ and $(r_2, s_2)$ from $a$. 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 since internal calls in one subclass cannot bind to method definitions in 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 specifications established for each method definition. The analysis of a requirement to a method definition succeeds directly if the requirement follows from the previously established specifications of that method. Otherwise, the requirement may make a new proof outline for the method necessary.

### *3.1. Assertions and Assertion Entailment*

Consider an assertion language with expressions $e$ defined by

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

In the assertion language, $f$ is a program field, $x$ a formal parameter, $z$ a logical variable, and $op$ an operation on data types. An *assertion pair* (of type *APair*) is a pair $(p, q)$ of Boolean expressions. Let $p'$ denote the expression $p$ with all occurrences of program variables $f$ substituted by the corresponding primed variables $f'$, avoiding name capture. Since we deal with sets of assertion pairs, the standard adaptation rule of Hoare Logic given in Figure 3 is insufficient. We need an entailment relation which allows us to combine information from several assertion pairs.

**Definition 1. (Entailment.)** Let $(p, q)$ and $(r, s)$ be assertion pairs and let $\mathcal{U}$ and $\mathcal{V}$ denote the 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 over assertion pairs and sets of assertion pairs by

1. $(p, q) \rightarrow (r, s) \triangleq (\forall \overline{z}_1 \,.\, p \Rightarrow q') \Rightarrow (\forall \overline{z}_2 \,.\, r \Rightarrow s')$,
   where $\overline{z}_1$ and $\overline{z}_2$ are the logical variables in $(p, q)$ and $(r, s)$, respectively.

2. $\mathcal{U} \twoheadrightarrow (r, s) \triangleq (\bigwedge_{1\leq i\leq n}(\forall \overline{z}_i \ . \ p_i \Rightarrow q_i')) \Rightarrow (\forall \overline{z} \ . \ r \Rightarrow s')$.
3. $\mathcal{U} \twoheadrightarrow \mathcal{V} \triangleq \bigwedge_{1\leq i\leq m} \mathcal{U} \twoheadrightarrow (r_i, s_i)$.

The relation $\mathcal{U} \twoheadrightarrow (r, s)$ corresponds to classic Hoare style reasoning, proving $\{r\}\ t\ \{s\}$ from $\{p_i\}\ t\ \{q_i\}$ for all $1 \leq i \leq n$, by means of the adaptation and conjunction rules [6]. Note that when proving entailment, program variables (primed and unprimed) are implicitly universally quantified. Furthermore, entailment is reflexive and transitive, and $\mathcal{V} \subseteq \mathcal{U}$ implies $\mathcal{U} \twoheadrightarrow \mathcal{V}$.

**Example 3.** Let $x$ and $y$ be fields, and $z_1$ and $z_2$ be logical variables. The assertion pair $(x = y = z_1,\ x = y = z_1 + 1)$ entails $(x = y,\ x = y)$, but it 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 see that, we take the assertion pairs as pre- and postconditions for the program $t \triangleq y := y + 1; x := y$. The Hoare triple $\{x = y = z_1\}\ t\ \{x = y = z_1 + 1\}$ is valid, whereas $\{x = z_2\}\ t\ \{x = z_2 + 1\}$ is not valid.

**Example 4.** This example demonstrates entailment for sets of assertion pairs: The two assertion pairs $(x \neq \mathit{null},\ x \neq \mathit{null})$ and $(y = z_1,\ z_1 = \mathit{null} \vee z_1 = y)$ entail $(x \neq \mathit{null} \vee y \neq \mathit{null},\ x \neq \mathit{null} \vee y \neq \mathit{null})$. This kind of reasoning is relevant for reasoning about class invariants without behavioral subtyping: when defining a subclass with a different class invariant than the superclass, the established knowledge of inherited methods may be used to prove the class invariant of the subclass.

### *3.2. Class Analysis with a Proof Environment*

The role of the proof environment during the class analysis will now be illustrated through a series of examples. Classes are analyzed after their respective superclasses, and each class is analyzed without knowledge of its possible subclasses. The proof environment collects the method specifications and requirements in two mappings $S$ and $R$. Given the names of a class and a method, these mappings return a set of assertion pairs. The analysis of a class both uses and extends the proof environment. In particular, $S(C, m)$ is the set of specifications established for the (possibly inherited) definition

of $m$ in class $C$, and $R(C, m)$ is the set of assertion pairs that must be respected by any redefinition of $m$ below $C$, as required by the analysis so far. By the analysis of class $C$, the user given specifications are included in the $S$ mapping. The analysis of proof outlines for these specifications may in turn impose requirements on internally called methods. These requirements are included in the $R$ mapping as explained below. The $S$ and $R$ mappings accumulate the results of the analysis so far, and form the basis of a mechanizable reasoning system for open class hierarchies (excluding generation of proof outlines). Intuitively, the mapping $S$ reflects the *definition of methods*; each lookup $S(C, m)$ returns a set of specifications for a particular implementation of $m$. In contrast, the mapping $R$ reflects the *use of methods* and may impose requirements on several implementations.

*Propagation of requirements.* If the proof outline $O \vdash_{PL} t : (p, q)$ for a method $T'\ m(\overline{T\ x})\{t\}$ is derived while analyzing a class $C$, we extend $S(C, m)$ with $(p, q)$. The requirements on called methods which are encountered during the analysis of $O$ are verified for the known definitions of these methods that are visible from $C$, and imposed on future subclasses. Thus, for each $\{r\}\ n()\ \{s\}$ in $O$, the following two steps are taken:

1. The requirement $(r, s)$ is analyzed with regard to the definition of $n$ that is visible from $C$.
2. $R(C, n)$ is extended with $(r, s)$.

The analysis in Step 1 ensures that the requirement can be relied on when the call is executed on an instance of class $C$. The inclusion of $(r, s)$ in $R(C, n)$ in Step 2 acts as a *restriction* on future subclasses of $C$. Whenever $n$ is overridden by a subclass of $C$, requirements $R(C, n)$ are verified for the new definition of $n$. Thereby, the requirement $(r, s)$ can also be relied on when the call in the body of $m$ is executed on an instance of a subclass of $C$. Consequently, the specification $(p, q)$ of $m$ can be relied on when the method is executed on a subclass instance. For a static call $\{r\}\ n@A(\overline{e})\ \{s\}$ in $O$, the assertion pair $(r, s)$ must follow from $S(A, n)$, the specification of $n$ in $A$. There is no need to impose this assertion pairs on subclass overridings since the call is bound at compile time, i.e., the assertion pair is not included in the set $R(C, n)$.

**Example 5.** Consider the analysis of class $A$ in Figure 2. The specification $(p_1, q_1)$ is analyzed for the definition of $a$, and included in the mapping

$S(A, a)$. For method $b$, the specification $(p_b, q_b)$ is analyzed and included in $S(A, b)$. In the body of $b$, there is a call to $a$ with requirement $(r_1, s_1)$. This requirement is analyzed for $a$ in $A$ (by Step 1), and included in $R(A, a)$ (by Step 2). The analysis of method $c$ follows the same strategy, and leads to the inclusion of $(r_2, s_2)$ in $R(A, a)$. By Step 1, both requirements must be verified for the definition of $a$ in $A$, since this is the definition of $a$ that is visible from $A$. Consequently, for each $(r_i, s_i)$, $S(A, a) \rightharpoonup (r_i, s_i)$ must hold. This relation holds directly, assuming $(p_1, q_1) \rightharpoonup (r_i, s_i)$. To summarize, the following assertion pairs are thereby included in the different specification and requirement sets:

$$\begin{array}{ll} S(A, a) = \{(p_1, q_1)\} & R(A, a) = \{(r_1, s_1), (r_2, s_2)\} \\ S(A, b) = \{(p_b, q_b)\} & \\ S(A, c) = \{(p_c, q_c)\} & \end{array}$$

In Example 5, it was assumed that the requirements made by $b$ and $c$ followed from the established specification of $a$. Generally, the requirements need not follow from the previously shown specifications. In such a case, it is necessary to provide a new proof outline for the method.

**Example 6.** If $(r_i, s_i)$ does not follow from $(p_1, q_1)$ in Example 5 (i.e., the relation $(p_1, q_1) \rightharpoonup (r_i, s_i)$ does not hold), a new proof outline $O \vdash_{PL} t_1 : (r_i, s_i)$ must be analyzed similarly to the proof outlines in $A$. The mapping $S(A, a)$ is extended by $(r_i, s_i)$, ensuring the desired relation $S(A, a) \rightharpoonup (r_i, s_i)$.

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

**Example 7.** Consider the class $B_1$ in Figure 2, which redefines $a$. By analysis of the proof outline $O_2 \vdash_{PL} t_2 : (p_2, q_2)$, the specification $(p_2, q_2)$ is included in $S(B_1, a)$. In addition, the superclass requirements $R(A, a)$ must hold for the new definition of $a$ in order to ensure that the specifications $S(A, b)$ and $S(A, c)$ of methods $b$ and $c$, respectively, apply for instances of $B_1$. Hence, $S(B_1, a) \rightharpoonup (r_i, s_i)$ must be ensured for each $(r_i, s_i) \in R(A, a)$, similar to $S(A, a) \rightharpoonup (r_i, s_i)$ in Example 5.

When a method $m$ is (re)defined in a class $C$, all invocations of $m$ from methods in superclasses will bind to the new definition for instances of $C$. 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., the formula $S(C, m) \rightharpoonup R(C, m)$ must hold.

*Context-dependent properties of inherited methods.* Consider now methods that are inherited but not redefined. Assume that a method $m$ is inherited from a superclass of a class $C$. In this case, late bound calls to $m$ from instances of $C$ are bound to the first definition of $m$ above $C$. However, late bound calls *made by* $m$ are bound *in the context of* $C$, as $C$ may redefine methods invoked by $m$. Furthermore, $C$ may impose new requirements on $m$ which were 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 late bound calls are bound from $C$. It would be unsound to extend the specification mapping of the superclass, since the new specifications are only part of the subclass context. Instead, we use $S(C, m)$ and $R(C, m)$ for *local specification and requirement extensions.* These new specifications and requirements only apply in the context of $C$ and not in the context of its superclasses.

**Example 8.** Assume that the class hierarchy in Figure 2 is extended by a class $B_3$ as indicated in Figure 4. Class $B_3$ inherits the superclass implementation of $a$. The specification $(p_d, q_d)$ is included in $S(B_3, d)$ and the analysis of a proof outline for this specification yields $\{r_3\}\ a()\ \{s_3\}$ as requirement, which is included in $R(B_3, a)$ and verified for the inherited implementation of $a$. The verification succeeds if $S(A, a) \rightharpoonup (r_3, s_3)$. Otherwise, a new proof outline $O \vdash_{PL} t_1 : (r_3, s_3)$ is analyzed under the assumption that late bound calls are bound in the context of $B_3$. When analyzed, $(r_3, s_3)$ becomes a specification of $a$ and it is included in $S(B_3, a)$. This mapping acts as a local extension of $S(A, a)$ and contains specifications of $a$ that hold in the subclass context.

When analyzing a requirement $\{r\}\ m()\ \{s\}$ in $C$, type safety guarantees that there exists a class $A$ above $C$ such that $m$ is defined in $A$ and that

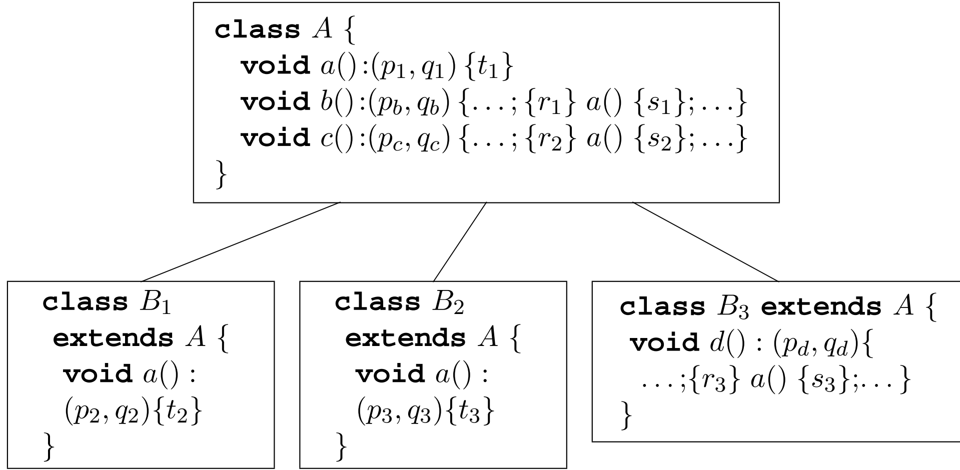


Figure 4: An extension of the class hierarchy in Figure 2 with a new class $B_3$.

this method definition is visible from $C$. For the requirement in Step 1, we can then rely on $S(A, m)$ and the local extensions of this set for all classes between $A$ and $C$. Let the function $S\!\uparrow$ be recursively defined 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 $C$. Equation 1 can now be revised to account for *inherited methods*:

$$S\!\uparrow\!(C, m) \rightharpoonup R\!\uparrow\!(C, m) \qquad (2)$$

Thus, each requirement in $R(B, m)$, for some class $B$ above $C$, must follow from the established specifications of $m$ in context $C$. Especially, for each $(r, s) \in R(C, m)$, $(r, s)$ must either follow from the superclass specifications or from the local extension $S(C, m)$. If $(r, s)$ 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 Equation 2 reduces to Equation 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. Methods are specified in terms of assertion pairs $(p, q)$. For each method $T'\ m(\overline{T\ x})\{t\}$ defined in a class $C$, we analyze each $(p, q)$ occurring either as a specification of $m$, or as an inherited requirement in $R\!\uparrow\!(C, m)$. If $S(C, m) \rightharpoonup (p, q)$, no further analysis of $(p, q)$ is needed. Otherwise a proof outline $O$ needs to be

provided such that $O \vdash_{PL} t : (p, q)$, after which $S(C, m)$ is extended with $(p, q)$. During the analysis of a proof outline, decorated late bound internal calls $\{r\}\ v := n(\overline{e})\ \{s\}$ 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 visible from $C$; i.e., the first implementation of $n$ above $C$, which means that the proof obligation $S{\uparrow}(C, n) \twoheadrightarrow (r, s)$ must hold. This obligation will hold directly if the already verified specifications of $n$ entail $(r, s)$. Otherwise, a new proof outline $O' \vdash_{PL} body(C, n) : (r, s)$ is needed, where method calls in $O'$ are analyzed in the same manner as for $O$. The set $S(C, n)$ is then extended with $(r, s)$, ensuring the proof obligation $S{\uparrow}(C, n) \twoheadrightarrow (r, s)$. For *static* calls $\{r\}\ n@A(\overline{e})\ \{s\}$ in $O$, which are bound to the first implementation of $n$ above $A$, the assertion pair $(r, s)$ must follow by entailment from $S{\uparrow}(A, n)$. For *external* calls $\{r\}\ e.n(\overline{e})\ \{s\}$ in $O$, with $e : E$, consider first the case that $(r, s)$ follows by entailment from the requirements $R{\uparrow}(E, n)$ of $n$ in $E$. By Equation 2 we then know that the requirement holds for $body(E, n)$. Since $R{\uparrow}(E, n)$ must be respected by overridings of $n$ below $E$, the assertion pair $(r, s)$ holds even if $e$ refers to an instance of a subclass of $E$ at run-time. In the opposite case, one may extend $R(E, n)$ upon the need of each external call, but this would lead to reverification of all subclasses of $E$. The analysis of external calls is further discussed in Section 5, where behavioral interfaces are used to achieve a modular reasoning system.

*Lazy behavioral subtyping.* Behavioral subtyping in the traditional sense does *not* follow from the analysis method outlined above. Behavioral subtyping enforces the property that whenever a method $m$ is redefined in a class $C$, its new definition must implement all superclass *specifications* 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 $a$ in both $B_1$ and $B_2$ in Figure 2 must satisfy $(p_1, q_1)$. Instead, the $R$ mapping identifies the requirements imposed by late bound internal calls. Only these assertion pairs must be supported by overriding methods to ensure that the execution of code from its superclasses does not have unexpected results. Thus, only the behavior assumed by the late bound internal 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 9.** The following class $A$ defines two methods $m$ and $n$, equipped with method specifications:

```
class A {
  int n(int y) : (true, return = 5 * y) {return := 5*y}
  int m(int x) : (x ≥ 0, return ≥ 2 * x) {return := n(x)}
}
```

For method $n$, analysis of the given specification leads to an inclusion of $(true, \mathbf{return} = 5 * y)$ in $S(A, n)$. No requirements are imposed by the analysis of this specification since the method body contains no call statements.

The analysis of method $m$ leads to an inclusion of the given specification, $(x \geq 0, \mathbf{return} \geq 2 * x)$, in $S(A, m)$. As the method body contains an internal late bound call, the analysis of a proof outline for this specification leads to a requirement towards the called method $n$. Let $(y \geq 0, \mathbf{return} \geq 2 * y)$ be the requirement imposed on the internal call to $n$ in the proof outline for $m$. During analysis of $A$, two steps are taken for this requirement. The requirement is verified with regard to the implementation of $n$ that is visible from $A$, and it is recorded in $R(A, n)$ in order to be imposed on future overridings below $A$. As the definition of $n$ in $A$ is the one that is visible from $A$, the first step is ensured by establishing $S(A, n) \rightharpoonup R(A, n)$, which follows directly by the definition of entailment, i.e.,

$$(true,\ \mathbf{return} = 5 * y) \rightharpoonup (y \geq 0,\ \mathbf{return} \geq 2 * y)$$

Next we consider the following extension of $A$:

```
class B1 extends A {
  int n(int y) : (true, return = 2 * y) {return := 2*y}
  int m(int x) : (true, return = 2 * x)
}
```

The method $n$ is overridden by $B_1$. Method $m$ is inherited without redefinition, but an additional specification of the inherited method is given. By analyzing $n$, the given specification $(true, \mathbf{return} = 2 * y)$ is included in $S(B_1, n)$ and is verified with regard to the method body. There are no method calls in this method body, and verification of the specification succeeds by standard Hoare reasoning. Additionally, the inherited requirement contained in $R(A, n)$ must be verified with regard to the subclass implementation of $n$, i.e., we need to ensure $S(B_1, n) \rightharpoonup R(A, n)$. This analysis succeeds by

entailment:

$$(true,\ \texttt{return} = 2 * y) \rightharpoondown (y \geq 0,\ \texttt{return} \geq 2 * y)$$

Note that behavioral subtyping does not apply to the overriding implementation of $n$, as the specification $S(A, n)$ cannot be proved for the new implementation. Even though the overriding does not support behavioral subtyping, the verified specification $S(A, m)$ of method $m$ still holds at the subclass level because the requirement imposed by the call to $n$ in the proof of this specification is satisfied by the overriding method.

Consider also the new specification $(true, \texttt{return} = 2 * x)$ of $m$. As the specification is given by the subclass, it is included in $S(B_1, m)$. Analysis of this specification leads to the requirement $(true, \texttt{return} = 2*y) \in R(B_1, n)$. This requirement is analyzed with regard to the visible implementation of $n$ in $B_1$ which follows directly by entailment: $S(B_1, n) \rightharpoondown R(B_1, n)$. Note that $S(B_1, m)$ gives a local extension of the inherited specification of $m$. Especially, the extension relies on the fact that the internal call to $n$ is bound in the context of class $B_1$; the requirement $R(B_1, n)$ imposed by the call could not be proven with regard to the implementation of $n$ in $A$. Combined, the verified specifications of $m$ in the context of $B_1$ are contained in $S\!\uparrow\!(B_1, m)$:

$$S\!\uparrow\!(B_1, m) = S(B_1, m) \cup S(A, m) = \\ \{(true, \texttt{return} = 2 * x), (x \geq 0, \texttt{return} \geq 2 * x)\}$$

which can be reduced to $\{(true, \texttt{return} = 2*x)\}$ by removing the redundant specification.

The following example extends Example 9 to illustrate how subclasses may have conflicting specifications of a superclass method.

**Example 10.** Consider the classes $A$ and $B_1$ given in Example 9 and let $B_2$ be the following extension of $A$:

```
class B2 extends A {
  int n(int y) : (true, return = 2 * y + 1) {return := 2*y+1}
  int m(int x) : (true, return = 2 * x + 1)
}
```

The analysis of $B_2$ is similar to the analysis of $B_1$, and $(true, \texttt{return} = 2 * y + 1)$ is added to $R(B_2, n)$. Notice that the $m$ specifications of the two

subclasses $B_1$ and $B_2$ are conflicting in the sense that their conjunction gives $(true,\ false)$. However, with lazy behavioral subtyping this does not cause a conflict since the two specifications are kept separate, extending $S(B_1, m)$ and $S(B_2, m)$, respectively. The verified specifications of $m$ in the context of $B_2$ are contained in $S{\uparrow}(B_2, m)$:

$$S{\uparrow}(B_2, m) = S(B_2, m) \cup S(A, m) = \\ \{(true, \mathbf{return} = 2 * x + 1), (x \geq 0, \mathbf{return} \geq 2 * x)\}.$$

## 4. An Assertion Calculus for Program Analysis

The incremental strategy outlined in Section 3 is now formalized as a calculus *LBS*(*PL*) which tracks specifications and requirements for method implementations in an extensible class hierarchy, given a sound program logic *PL*. Given a program, the calculus builds an environment which reflects the class hierarchy and captures method specifications and requirements. This environment forms the context for the analysis of new classes, possibly inheriting previously analyzed ones. To emphasize program analysis, we assume that programs are type-safe and hereafter ignore the types of fields and methods in the discussion. The proof environment is formally defined in Section 4.1, and *LBS*(*PL*) is given as a set of inference rules in Section 4.2. The soundness of *LBS*(*PL*) is established in Section 4.3.

### *4.1. The Proof Environment of LBS(PL)*

A class is represented by a unique name and a tuple $\langle B, \overline{f}, \overline{M} \rangle$ of type *Class* from which the superclass name $B$, the fields $\overline{f}$, and the methods $\overline{M}$ are accessible by observer functions *inh*, *atts*, and *mtds*, respectively. Method names are assumed to be unique within a class. Note that the method specifications $\overline{MS}$ in class definitions are not included in class tuples. Specifications are instead collected in the specification mapping $S$ of proof environments:

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

In a proof environment $\mathcal{E}$, the mapping $L_\mathcal{E}$ reflects the class hierarchy, the set $S_\mathcal{E}(C, m)$ contains the specifications for $m$ in $C$, and the set $R_\mathcal{E}(C, m)$ contains the requirements to $m$ from $C$. For the *empty environment* $\mathcal{E}_\emptyset$,

$$
\begin{array}{lcl}
bind_{\mathcal{E}}(nil,\, m) & \triangleq & \bot \\
bind_{\mathcal{E}}(C,\, m) & \triangleq & \textbf{if } m \in L_{\mathcal{E}}(C).mtds \textbf{ then } C \textbf{ else } bind_{\mathcal{E}}(L_{\mathcal{E}}(C).inh,\, m) \\[4pt]
S{\uparrow}_{\mathcal{E}}(nil, m) & \triangleq & \emptyset \\
S{\uparrow}_{\mathcal{E}}(C, m) & \triangleq & \textbf{if } m \in L_{\mathcal{E}}(C).mtds \textbf{ then } S_{\mathcal{E}}(C, m) \\
 & & \qquad\qquad \textbf{else } S_{\mathcal{E}}(C, m) \cup S{\uparrow}_{\mathcal{E}}(L_{\mathcal{E}}(C).inh, m) \\[4pt]
R{\uparrow}_{\mathcal{E}}(nil, m) & \triangleq & \emptyset \\
R{\uparrow}_{\mathcal{E}}(C, m) & \triangleq & R_{\mathcal{E}}(C, m) \cup R{\uparrow}_{\mathcal{E}}(L_{\mathcal{E}}(C).inh, m) \\[4pt]
body_{\mathcal{E}}(C, m) & \triangleq & L_{\mathcal{E}}(bind_{\mathcal{E}}(C,\, m)).mtds(m).body \\[4pt]
nil \ \leq_{\mathcal{E}} B & \triangleq & false \\
C \ \leq_{\mathcal{E}} B & \triangleq & C = B \vee L_{\mathcal{E}}(C).inh \ \leq_{\mathcal{E}} B
\end{array}
$$

Figure 5: Auxiliary function definitions, where $C, B : Cid$ and $m : Mid$.

$L_{\mathcal{E}_\emptyset}(C)$ is undefined and $S_{\mathcal{E}_\emptyset}(C, m) = R_{\mathcal{E}_\emptyset}(C, m) = \emptyset$ for all $C : Cid$ and $m : Mid$.

Some *auxiliary functions* on proof environments $\mathcal{E}$ are now defined. Assuming that *nil* is not a valid *Cid*, these functions range over the type $Cid_n$, where $Cid_n$ equals *Cid* extended with *nil*, assuming **`Object`**$.inh = nil$. Let $M.body = t$ for a method definition $M = m(\overline{x})\{t\}$. Denote by $\overline{M}(m)$ the definition of method with name $m$ in $\overline{M}$, by $m \in \overline{M}$ that $m$ is defined in $\overline{M}$, by $t' \in t$ that the statement $t'$ occurs in the statement $t$, and by $C \in \mathcal{E}$ that $L_{\mathcal{E}}(C)$ is defined. The partial function $bind_{\mathcal{E}}(C,\, m) : Cid_n \times Mid \rightarrow Cid$ returns the first class above $C$ in which the method $m$ is defined. This function is well-defined since programs are well-typed by assumption. Let the recursively defined functions $S{\uparrow}_{\mathcal{E}}(C, m)$ and $R{\uparrow}_{\mathcal{E}}(C, m) : Cid_n \times Mid \rightarrow Set[APair]$ return all specifications of $m$ above $C$ and below $bind_{\mathcal{E}}(C,\, m)$, and all requirements to $m$ that are made by all classes above $C$ in the proof environment $\mathcal{E}$, respectively. Finally, $body_{\mathcal{E}}(C, m) : Cid \times Mid \rightarrow Stm$ returns the implementation of $m$ in $bind_{\mathcal{E}}(C,\, m)$. Let $\leq_{\mathcal{E}}\colon Cid_n \times Cid \rightarrow Bool$ be the reflexive and transitive subclass relation on $\mathcal{E}$. The definitions of these functions are given in Figure 5.

A *sound environment* reflects that the analyzed classes are correct. If an assertion pair $(p, q)$ appears in $S_{\mathcal{E}}(C, m)$, there must be a verified proof outline $O$ in $PL$ for the corresponding method body, i.e., $O \vdash_{PL} body_{\mathcal{E}}(C, m) : (p, q)$.

Let $n$ be a method called by $m$, and let $\overline{x}$ be the formal parameters of $n$. For all late bound internal calls $\{r\}\ v := n(\overline{e})\ \{s\}$ in the proof outline $O$, the requirement $(r', s')$ is included in $R_{\mathcal{E}}(C, n)$, where $r' = (r \wedge \overline{x} = \overline{e})$, and $s' = s[\mathbf{return}/v]$, assuming that the variables $\overline{x}, \mathbf{return}$ do not occur in $r, s, \overline{e}$ (otherwise renaming is needed; in the special case where $v$ is $\mathbf{return}$, $s'$ is simply $s$.) Here, $r'$ accounts for the assignment of actual parameter values to the formal parameters, and $s'$ accounts for the assignment of the returned value to $v$. Thus, all requirements made by internal late bound calls in the proof outline are in the $R$ mapping. For static internal calls $\{r\}\ v := n@A(\overline{e})\ \{s\}$ in $O$, the assertion pair $(r', s')$ must follow from the specifications $S\!\uparrow\!(A, n)$. For external calls $\{r\}\ v := e.n(\overline{e})\ \{s\}$ in $O$, with $e : E$, the requirement $(r', s')$ must follow from $R\!\uparrow_{\mathcal{E}}\!(E, n)$. Note that $E$ may be independent of the analyzed class $C$; i.e., the classes need not be related by inheritance. Finally, method specifications must entail the requirements (see Equation 2 of Section 3.2). Sound environments are defined as follows:

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

1. $\forall (p, q) \in S_{\mathcal{E}}(C, m)\ .\ \exists O\ .\ O \vdash_{PL} body_{\mathcal{E}}(C, m) : (p, q)$
   $\wedge\ \forall \{r\}\ v := n(\overline{e})\ \{s\} \in O\ .\ R\!\uparrow_{\mathcal{E}} (C, n) \rightarrow (r', s')$
   $\wedge\ \forall \{r\}\ v := n@A(\overline{e})\ \{s\} \in O\ .\ S\!\uparrow_{\mathcal{E}}(A, n) \rightarrow (r', s')$
   $\wedge\ \forall \{r\}\ v := e.n(\overline{e})\ \{s\} \in O\ .\ e : E \Rightarrow R\!\uparrow_{\mathcal{E}}(E, n) \rightarrow (r', s')$
2. $S\!\uparrow_{\mathcal{E}}(C, m) \rightarrow R\!\uparrow_{\mathcal{E}}(C, m)$
3. $L_{\mathcal{E}}(C).inh \neq nil \Rightarrow L_{\mathcal{E}}(C).inh \in \mathcal{E}$
   $\wedge\ \forall B\ .\ B \notin \mathcal{E}\ .\ S_{\mathcal{E}}(B, m) = R_{\mathcal{E}}(B, m) = \emptyset.$

where $r' = (r \wedge \overline{x} = \overline{e})$, $s' = s[\mathbf{return}/v]$, and $\overline{x}$ are the formal parameters of $n$ (assuming that $\overline{x}, \mathbf{return}$ do not occur in $r, s$, or $\overline{e}$).

Note that in Condition 1 of Definition 3, the method implementation $body_{\mathcal{E}}(C, m)$ need not be in $C$ itself; the proof outline $O$ may be given for an inherited method definition.

Let $\models_C \{p\}\ t\ \{q\}$ denote $\models \{p\}\ t\ \{q\}$ under the assumption that internal calls in $t$ are bound in the context of $C$, and that each external call in $t$ is bound in the context of the actual class of the called object. 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 $\vdash_{PL} \{p\}\ t\ \{q\}$, then $\models \{p\}\ t\ \{q\}$ follows by the soundness of $PL$. The following property holds for sound environments:

**Lemma 1.** *Given a sound environment $\mathcal{E}$ and a sound program logic PL. For all classes $C$ : Cid, methods $m$ : Mid, and assertion pairs $(p, q)$ : APair 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.* By induction on the number of calls in $m$. Since $(p, q) \in S{\uparrow}_{\mathcal{E}}(C, m)$, it follows from the definition of $S{\uparrow}$ in Figure 5 that there exist some class $B$ such that $C \leq_{\mathcal{E}} B$, $bind_{\mathcal{E}}(C, m) = bind_{\mathcal{E}}(B, m)$, and $(p, q) \in S_{\mathcal{E}}(B, m)$. For such a class $B$, $body_{\mathcal{E}}(C, m) = body_{\mathcal{E}}(B, m)$. Since $(p, q) \in S_{\mathcal{E}}(B, m)$, there must, by Definition 3, Condition 1, exist some proof outline $O$ such that $O \vdash_{PL} body_{\mathcal{E}}(B, m) : (p, q)$.

In this proof outline, each method call is decorated with pre- and post-conditions; i.e., the outline is of the form

$$\{p\}t_0\{r_1\}call_1\{s_1\}t_1\{r_2\}call_2\{s_2\} \dots \{r_n\}call_n\{s_n\}t_n\{q\}$$

assuming no method calls in the statements $t_0, \dots, t_n$. For the different $t_i$, soundness of $PL$ then gives $\models_D \{p\}\ t_0\ \{r_1\}, \models_D \{s_i\}\ t_i\ \{r_{i+1}\}$, and $\models_D \{s_n\}\ t_n\ \{q\}$, for $1 \leq i < n$. Each call statement is of the form $v := n(\overline{e})$, $v := n@A(\overline{e})$, or $v := e.n(\overline{e})$.

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

*Induction step:* Take as the induction hypothesis that for each call to some method $n$ in the body of $m$, bound to an implementation $body_{\mathcal{E}}(F, n)$ in context $F'$ ($F' \leq_{\mathcal{E}} F$), we have $\models_{F'} n(\overline{x}) : (g, h) \{body_{\mathcal{E}}(F, n)\}$ for each $(g, h) \in S{\uparrow}_{\mathcal{E}}(F, n)$. The different call statements are considered separately.

Consider a method call $\{r\}\ v := n(\overline{e})\ \{s\}$ in $O$, and let $r'$ and $s'$ be as in Definition 3. By the assumptions of the Lemma, the call is bound to $body_{\mathcal{E}}(D, n)$ in the context of class $D \leq_{\mathcal{E}} C$. For all $(g, h) \in S{\uparrow}_{\mathcal{E}}(D, n)$, we have by the induction hypothesis that $\models_D n(\overline{x}) : (g, h) \{body_{\mathcal{E}}(D, n)\}$. By Definition 3, Condition 1, we have $R_{\mathcal{E}}(B, n) \rightharpoonup (r', s')$. Then the desired conclusion $\models_D \{r'\}\ body_{\mathcal{E}}(D, n)\ \{s'\}$ follows since $S{\uparrow}_{\mathcal{E}}(D, n) \rightharpoonup R{\uparrow}_{\mathcal{E}}(D, n)$ by Definition 3, Condition 2, which especially means $S{\uparrow}_{\mathcal{E}}(D, n) \rightharpoonup R_{\mathcal{E}}(B, n)$ since $D \leq_{\mathcal{E}} C \leq_{\mathcal{E}} B$, i.e., $R_{\mathcal{E}}(B, n) \subseteq R{\uparrow}_{\mathcal{E}}(D, n)$.

Consider a method call $\{r\}\ v := n@A(\overline{e})\ \{s\}$ in $O$ and let $r', s'$ be as in Definition 3. The call is bound to $body_{\mathcal{E}}(A, n)$ in the context $D$. By the induction hypothesis, we have $\models_D n(\overline{x}) : (g, h) \{body_{\mathcal{E}}(A, n)\}$ for all $(g, h) \in S{\uparrow}_{\mathcal{E}}(A, n)$. Then the conclusion $\models_D \{r'\}\ body_{\mathcal{E}}(A, n)\ \{s'\}$ follows since $S{\uparrow}_{\mathcal{E}}(A, n) \rightharpoonup (r', s')$ by Definition 3, Condition 2.

Consider a method call $\{r\}\ v := e.n(\overline{e})\ \{s\}$ in $O$, and let $e : E$, and $r', s'$ be as in Definition 3. From Definition 3, Condition 1, we have $R{\uparrow_{\mathcal{E}}}(E,n) \rightharpoonup (r', s')$. The call can be bound in the context of any class $E'$ below $E$. By the definition of $R\uparrow$ in Figure 5, we have $R{\uparrow_{\mathcal{E}}}(E,n) \subseteq R{\uparrow_{\mathcal{E}}}(E',n)$ for $E' \leq_{\mathcal{E}} E$. Since $R{\uparrow_{\mathcal{E}}}(E,n) \rightharpoonup (r', s')$, this gives $R{\uparrow_{\mathcal{E}}}(E',n) \rightharpoonup (r', s')$. By Definition 3, Condition 2, we have $S{\uparrow_{\mathcal{E}}}(E',n) \rightharpoonup R{\uparrow_{\mathcal{E}}}(E',n)$, which especially means $S{\uparrow_{\mathcal{E}}}(E',n) \rightharpoonup (r', s')$. The conclusion $\models_{E'} \{r'\}\ body_{\mathcal{E}}(E',n)\ \{s'\}$ then follows by the induction hypothesis. □

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 proof environment $\mathcal{E}$ is *minimal* iff for all $C : Cid$ and $n : Mid$ with formal parameters $\overline{x}$:

$$\begin{array}{l} \forall (r', s') \in R_{\mathcal{E}}(C,n)\ .\ \exists\, p, q, r, s, m, O\ . \\ \quad (p,q) \in S_{\mathcal{E}}(C,m) \wedge O \vdash_{PL} body_{\mathcal{E}}(C,m) : (p,q) \\ \quad \wedge\ \{r\}\ v := n(\overline{e})\ \{s\} \in O \wedge r' = (r \wedge \overline{x} = \overline{e}) \wedge s' = s[\mathbf{return}/v] \end{array}$$

*4.2. The Inference Rules of LBS(PL)*

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 $E$, and a method in $E$ 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 *modules* as the granularity of program analysis, where a module consists of a set of classes. Such a module is *self-contained* with respect 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 $\mathcal{E} \vdash \mathcal{M}$, where $\mathcal{E}$ is the proof environment and $\mathcal{M}$ is a list of *analysis operations* on the class hierarchy. The syntax for analysis operations is outlined in Figure 6, and the different operations are explained below. Let $LBS(PL)$ denote the reasoning system for lazy behavioral subtyping based on a (sound) program logic $PL$, which uses a proof environment $\mathcal{E} : Env$ and the *inference rules* given in Figure 7 and Figure 8.

There are three different *environment updates*; the loading of a new class $L$ into the environment and the extension of the specification and requirement

$$
\begin{array}{lll}
\mathcal{O} & ::= & \epsilon \mid anReq(\overline{M}) \mid anSpec(\overline{MS}) \mid verify(m, \overline{R}) \mid anCalls(t) \mid \mathcal{O} \cdot \mathcal{O} \\
\mathcal{L} & ::= & \emptyset \mid L \mid require(C, m, (r, s)) \mid \mathcal{L} \cup \mathcal{L} \\
\mathcal{M} & ::= & module(\overline{L}) \mid [\langle C : \mathcal{O} \rangle ; \mathcal{L}] \mid [\epsilon ; \mathcal{L}] \mid \mathcal{M} \cdot module(\overline{L})
\end{array}
$$

Figure 6: Syntax for the analysis operations. Here, $M$, $MS$, and $L$ are as in Figure 1, $\overline{R}$ is a set of assertion pairs, and $t$ is a statement decorated with pre- and post conditions to method calls.

mappings with an assertion pair $(p, q)$ for a given method $m$ and class $C$. These are denoted $extL(C, B, \overline{f}, \overline{M})$, $extS(C, m, (p, q))$ and $extR(C, m, (p, q))$, respectively. The inference system below ensures that the same class is never loaded twice into the environment. Environment updates are represented by the operator $\oplus$ : $Env \times Update \rightarrow Env$, where the first argument is the current proof environment and the second argument is the environment update, defined as follows:

$$
\begin{array}{rcl}
\mathcal{E} \oplus extL(C, B, \overline{f}, \overline{M}) & \triangleq & \langle L_{\mathcal{E}}[C \mapsto \langle B, \overline{f}, \overline{M} \rangle], S_{\mathcal{E}}, R_{\mathcal{E}} \rangle \\
\mathcal{E} \oplus extS(C, m, (p, q)) & \triangleq & \langle L_{\mathcal{E}}, S_{\mathcal{E}}[(C, m) \mapsto (S_{\mathcal{E}}(C, m) \cup \{(p, q)\})], R_{\mathcal{E}} \rangle \\
\mathcal{E} \oplus extR(C, m, (p, q)) & \triangleq & \langle L_{\mathcal{E}}, S_{\mathcal{E}}, R_{\mathcal{E}}[(C, m) \mapsto (R_{\mathcal{E}}(C, m) \cup \{(p, q)\})] \rangle
\end{array}
$$

The main inference rules of the assertion calculus are given in Figure 7. In addition, there are lifting rules concerned with the analysis of set and list structures, and trivial cases, which are given in Figure 8. Note that $\mathcal{M}$ represents a list of *module* operations which will be analyzed later and which may be empty. Rule (NewModule) initiates the analysis of a new module $module(\overline{L})$. The analysis continues by manipulation of the $[\epsilon ; \overline{L}]$ operation that is generated by this rule. For notational convenience, we let $\overline{L}$ denote both a set and list of classes. During the analysis of a module, the proof environment is extended in order to keep track of the currently analyzed class hierarchy and the associated method specifications and requirements.

Rule (NewClass) selects a new class **class** $C$ **extends** $B$ $\{\overline{f}\ \overline{M}\ \overline{MS}\}$ from the current module, and initiates the analysis of the class in the current proof environment. Note that at this point in the analysis, class $C$ has no subclasses in the proof environment. Classes are assumed to be syntactically well-formed and well-typed. The premises of (NewClass) ensure that a class cannot be introduced twice and that the superclass has *already been analyzed.* The rule generates an operation of the form $[\langle C : \mathcal{O} \rangle ; \mathcal{L}]$, and the inference

$$\frac{\mathcal{E} \vdash [\epsilon\,;\, \overline{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash \mathit{module}(\overline{L}) \cdot \mathcal{M}} \quad (\textsc{NewModule})$$

$$\frac{\begin{array}{c} C \notin \mathcal{E} \qquad B \neq \text{nil} \Rightarrow B \in \mathcal{E} \\ \mathcal{E} \oplus \mathit{extL}(C, B, \overline{f}, \overline{M}) \vdash [\langle C :\ \mathit{anSpec}(\overline{MS}) \cdot \mathit{anReq}(\overline{M})\rangle\,;\, \mathcal{L}] \cdot \mathcal{M} \end{array}}{\mathcal{E} \vdash [\epsilon\,;\, \{\mathtt{\textbf{class}}\ C\ \mathtt{\textbf{extends}}\ B\ \{\overline{f}\ \overline{M}\ \overline{MS}\}\} \cup \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{NewClass})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathit{verify}(m, (p, q)) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anSpec}(m(\overline{x}) : (p, q)) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{NewSpec})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathit{verify}(m, R{\Uparrow_{\mathcal{E}}}(L_{\mathcal{E}}(C).inh, m)) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anReq}(m(\overline{x})\{t\}) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{NewMtd})$$

$$\frac{S{\Uparrow_{\mathcal{E}}}(C, m) \rightarrow (p, q) \qquad \mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{verify}(m, (p, q)) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{ReqDer})$$

$$\frac{\begin{array}{c} O \vdash_{PL} \mathit{body}_{\mathcal{E}}(C, m) : (p, q) \\ \mathcal{E} \oplus \mathit{extS}(C, m, (p, q)) \vdash [\langle C :\ \mathit{anCalls}(O) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M} \end{array}}{\mathcal{E} \vdash [\langle C :\ \mathit{verify}(m, (p, q)) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{ReqNotDer})$$

$$\frac{\mathcal{E} \oplus \mathit{extR}(C, n, (r', s')) \vdash [\langle C :\ \mathit{verify}(n, (r', s')) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anCalls}(\{r\}\ v := n(\overline{e})\ \{s\}) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{IntCall})$$

$$\frac{S{\Uparrow_{\mathcal{E}}}(A, n) \rightarrow (r', s') \qquad \mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anCalls}(\{r\}\ v := n@A(\overline{e})\ \{s\}) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{StaticCall})$$

$$\frac{e : E \qquad \mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle\,;\, \mathcal{L} \cup \{\mathit{require}(E, n, (r', s'))\}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anCalls}(\{r\}\ v := e.n(\overline{e})\ \{s\}) \cdot \mathcal{O}\rangle\,;\, \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{ExtCall})$$

$$\frac{C \in \mathcal{E} \qquad R{\Uparrow_{\mathcal{E}}}(C, m) \rightarrow (p, q) \qquad \mathcal{E} \vdash [\epsilon\,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\epsilon\,;\, \{\mathit{require}(C, m, (p, q))\} \cup \mathcal{L}] \cdot \mathcal{M}} \quad (\textsc{ExtReq})$$

Figure 7: *The inference system*, where $\mathcal{M}$ is a (possibly empty) list of analysis operations. In the call rules, we have $r' = (r \wedge \overline{x} = \overline{e})$ and $s' = s[\mathtt{\textbf{return}}/v]$, where $\overline{x}$ are the formal parameters of $n$ (assuming $\overline{x}$ and **`return`** do not occur in $r, s$, or $\overline{e}$).

rules analyzes the operations $\mathcal{O}$ in the context of class $C$ *before* operations in $\mathcal{L}$ are considered. Thus, the analysis of $C$ is completed before a new class is

$$\frac{\mathcal{E} \vdash [\epsilon \,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \epsilon\rangle \,;\, \mathcal{L}] \cdot \mathcal{M}} \ (\textsc{EmpClass}) \qquad \frac{\mathcal{E} \vdash \mathcal{M}}{\mathcal{E} \vdash [\epsilon \,;\, \emptyset] \cdot \mathcal{M}} \ (\textsc{EmpModule})$$

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

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle \,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anReq}(\emptyset) \cdot \mathcal{O}\rangle \,;\, \mathcal{L}] \cdot \mathcal{M}} \ (\textsc{NoMtds})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle \,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anSpec}(\emptyset) \cdot \mathcal{O}\rangle \,;\, \mathcal{L}] \cdot \mathcal{M}} \ (\textsc{NoSpec})$$

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

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathit{anCalls}(t_1) \cdot \mathit{anCalls}(t_2) \cdot \mathcal{O}\rangle \,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anCalls}(t_1; t_2) \cdot \mathcal{O}\rangle \,;\, \mathcal{L}] \cdot \mathcal{M}} \ (\textsc{DecompSeq})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathit{anCalls}(t_1) \cdot \mathit{anCalls}(t_2) \cdot \mathcal{O}\rangle \,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anCalls}(\mathbf{if}\ b\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{fi}) \cdot \mathcal{O}\rangle \,;\, \mathcal{L}] \cdot \mathcal{M}} \ (\textsc{DecompIf})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathcal{O}\rangle \,;\, \mathcal{L}] \cdot \mathcal{M} \qquad t \text{ does not contain call statements}}{\mathcal{E} \vdash [\langle C :\ \mathit{anCalls}(t) \cdot \mathcal{O}\rangle \,;\, \mathcal{L}] \cdot \mathcal{M}} \ (\textsc{Skip})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathit{anReq}(\overline{M_1}) \cdot \mathit{anReq}(\overline{M_2}) \cdot \mathcal{O}\rangle \,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anReq}(\overline{M_1}\ \overline{M_2}) \cdot \mathcal{O}\rangle \,;\, \mathcal{L}] \cdot \mathcal{M}} \ (\textsc{DecompMtds})$$

$$\frac{\mathcal{E} \vdash [\langle C :\ \mathit{anSpec}(\overline{MS_1}) \cdot \mathit{anSpec}(\overline{MS_2}) \cdot \mathcal{O}\rangle \,;\, \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C :\ \mathit{anSpec}(\overline{MS_1}\ \overline{MS_2}) \cdot \mathcal{O}\rangle \,;\, \mathcal{L}] \cdot \mathcal{M}} \ (\textsc{DecompSpec})$$

Figure 8: *The inference system: Lifting rules decomposing list-like structures and handling trivial cases.* Here $\mathcal{M}$ is a (possibly empty) list of analysis operations.

loaded for analysis. The syntax for analysis operations $\mathcal{O}$ is given in Figure 6.

By application of (NewClass), the class hierarchy is extended with $C$, and $\mathcal{O}$ consists initially of two operations: $\mathit{anSpec}(\overline{MS})$ and $\mathit{anReq}(\overline{M})$. These operations may be flattened by the rules (DecompSpec) and (DecompMtds) of Figure 8, respectively. For each specification $m(\overline{x}) : (p, q)$ in $\overline{MS}$, Rule

(NewSpec) initiates analysis of $(p, q)$ with regard to the visible implementation of the method by means of a $\mathit{verify}(m, (p, q))$ operation. Furthermore, for each method definition $m(\overline{x})\{t\}$ in $\overline{M}$, Rule (NewMtd) generates an operation $\mathit{verify}(m, R{\uparrow}(B, m))$. Here, $R{\uparrow}(B, m)$ contains the requirements towards $m$ that are imposed by the superclasses of $C$. The requirement set of a *verify* operation may be decomposed by rule (DecompReq) of Figure 8. (Figure 8 also contains rules (NoReq), (NoMtds), and (NoSpec) to discard empty *verify*, *anReq*, and *anSpec* operations, respectively.) The two analysis operations $\mathit{anSpec}(\overline{MS})$ and $\mathit{anReq}(\overline{M})$ thereby ensure that:

- For each specification $m(\overline{x}) : (p, q)$ in $\overline{MS}$, an operation $\mathit{verify}(m, (p, q))$ is generated.
- If $m$ is a method defined in $C$, and some superclass of $C$ imposes the requirement $(r, s)$ on $m$, i.e., $m$ overrides a superclass definition, an operation $\mathit{verify}(m, (r, s))$ is generated.

The generated *verify* operations are analyzed either by Rule (ReqDer) or by Rule (ReqNotDer). For each method $m$, the set $S(C, m)$ is initially empty, and this set is only extended by Rule (ReqNotDer). If an assertion pair $(p, q)$ is included in $S(C, m)$, this rule requires that a proof outline $O$ for $m$ is provided such that $O \vdash_{PL} \mathit{body}(C, m) : (p, q)$. The analysis then continues by considering the decorated method body by means of an $\mathit{anCalls}(O)$ operation, as described below. The assertion pair $(p, q)$ thereby leads to a new specification for $m$ with respect to $C$, and $(p, q)$ itself is assumed when analyzing the method body. This captures the standard approach to reasoning about recursive method calls [26].

Now, consider the analysis of some operation $\mathit{verify}(m, (p_1, q_1))$. Since the set $S(C, m)$ is incrementally extended during analysis of $C$, it might be the case that $(p_1, q_1)$ follows by entailment from the already verified assertion pairs, i.e., $S(C, m) \rightarrow (p_1, q_1)$. In this case, no further analysis of $(p_1, q_1)$ is needed, and the $\mathit{verify}(m, (p_1, q_1))$ operation is discarded by Rule (ReqDer). Otherwise, a proof outline for $(p_1, q_1)$ is needed. The operation is then verified by Rule (ReqNotDer) as described above. In general, the definition of method $m$ can be inherited by $C$ without redefinition, which means that $(p_1, q_1)$ may follow from already verified assertion pairs in the superclass. In Rule (ReqDer), this is captured by the relation $S{\uparrow}(C, m) \rightarrow (p_1, q_1)$. Remember that $S{\uparrow}(C, m)$ reduces to $S(C, m)$ if $m$ is defined in $C$ (cf. Figure 5).

Next we consider the analysis of *anCalls*$(O)$ operations generated by (ReqNotDer), where $O$ is a proof outline for some method body where call statements are decorated with pre- and postconditions. The proof outline is decomposed by the rules (DecompSeq) and (DecompIf) in Figure 8. Rule (Skip) applies to statements which are irrelevant to the *anCalls* analysis. Rule (IntCall) (cf. Figure 7) analyzes the requirement of an internal call in the 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 $C$. The extension of the $R$ mapping ensures that future redefinitions must respect the new requirement; i.e., the requirement is imposed whenever redefinitions are considered by (NewMtd). Rule (StaticCall) handles external calls by ensuring that the required assertion pair follows from the specification of the called method. Note that this rule does not extend the $R$ mapping since the call is bound at compile time. Rule (ExtCall) handles external calls of the form $v := e.n(\overline{e})$. The requirement to the external method is removed from the context of the current class and propagated as a *require* operation in the module operations $\mathcal{L}$. The type of the callee is found by static type analysis of $e$, expressed by the premise $e : E$. Rule (ExtReq) can first be applied *after* the analysis of the callee class is completed, and the requirement must then follow from the requirements of this class. For simplicity we have here omitted formalization of reverification, since this will complicate the system further, and since the next section gives a solution without reverification.

By the successful analysis of class $C$, an operation on the form $[\langle C : \epsilon\rangle\,; \mathcal{L}]$ is reached, and by application of Rule (EmpClass), this yields the operation $[\epsilon\ ;\ \mathcal{L}]$. Another class in $\mathcal{L}$ can then be enabled for analysis. 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. Note that a proof of $\mathcal{E} \vdash module(\overline{L})$ has exactly one leaf node $\mathcal{E}' \vdash [\epsilon; \emptyset]$; we call $\mathcal{E}'$ the *environment resulting* from the analysis of $module(\overline{L})$.

Program analysis is initiated by the judgment $\mathcal{E}_\emptyset \vdash module(\overline{L})$, where $\overline{L}$ is a module that is self-contained in the empty environment. Subsequent modules are analyzed in sequential order, such that each module is self-contained with respect to the environment resulting from the analysis of previous modules. When the analysis of a module is completed, the resulting environment represents a verified class hierarchy. New modules may introduce subclasses of classes which have been analyzed in previous modules. The calculus is

based on an open world assumption in the sense that a module is analyzed in the context of previously analyzed modules, but it is independent of subsequent modules.

**Example 11.** As an illustration of a derivation in $LBS(PL)$, we consider the analysis of class $B_1$ from Example 9. Thus, we assume that class $A$ has already been analyzed, resulting in the environment $\mathcal{E}_0$ where

$$L_{\mathcal{E}_0}(A) = \langle nil, \emptyset, \mathtt{n(y)\{}\mathbf{return} := \mathtt{5*y\}}\ \mathtt{m(x)\{}\mathbf{return} := \mathtt{n(x)\}}\rangle$$

and $L_{\mathcal{E}_0}$ is undefined for all other classes. For simplicity we here ignore class **Object** and take $A.inh = nil$. As explained in Example 9, the following $S$ and $R$ sets are non-empty:

$$\begin{array}{ll} S_{\mathcal{E}_0}(A,n) & = \{(true, \mathbf{return} = 5 * y)\} \\ S_{\mathcal{E}_0}(A,m) & = \{(x \geq 0, \mathbf{return} \geq 2 * x)\} \\ R_{\mathcal{E}_0}(A,n) & = \{(y \geq 0, \mathbf{return} \geq 2 * y)\} \end{array}$$

The analysis of class $B_1$ is initiated by the judgment

$$\mathcal{E}_0 \vdash\ module(\mathbf{class}\ B_1\ \mathbf{extends}\ A\ \{M\ \overline{MS}\})$$

where:

$$\begin{array}{llll} M & = \mathtt{n(y)\{}\mathbf{return} := \mathtt{2*y\}} & Sn & = n(y) : (true, \mathbf{return} = 2 * y) \\ \overline{MS} & = \{Sn\ \ Sm\} & Sm & = m(x) : (true, \mathbf{return} = 2 * x) \end{array}$$

The successful derivation of this judgment is given in Figure 9, leading to the resulting environment $\mathcal{E}_4$ shown in the figure.

### *4.3. Properties of LBS(PL)*

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.

**Lemma 2.** *Let $\mathcal{E}$ be an environment such that for all class names $B$ and method names $m$, the following holds: If $B \in \mathcal{E}$ then $S{\uparrow_{\mathcal{E}}}(B,m) \rightharpoonup R{\uparrow_{\mathcal{E}}}(B,m)$ and $L_{\mathcal{E}}(B).inh \neq nil \Rightarrow L_{\mathcal{E}}(B).inh \in \mathcal{E}$. Otherwise, if $B \notin \mathcal{E}$, then $S(B,m) = R(B,m) = \emptyset$.*

$$
\begin{array}{cl}
\mathcal{E}_4 \vdash & \\
\hline
\mathcal{E}_4 \vdash [\epsilon \; ; \; \emptyset] & (\textsc{EmpModule}) \\
\hline
\mathcal{E}_4 \vdash [\langle B_1 : \; \epsilon \rangle \; ; \; \emptyset] & (\textsc{EmpClass}) \\
\hline
\mathcal{E}_4 \vdash [\langle B_1 : \; \mathit{verify}(n, (y \geq 0, \mathtt{\textbf{return}} \geq 2 * y)) \rangle \; ; \; \emptyset] & (\textsc{ReqDer})(7) \\
\hline
\mathcal{E}_4 \vdash [\langle B_1 : \; \mathit{anReq}(M) \rangle \; ; \; \emptyset] & (\textsc{NewMtd})(6) \\
\hline
\mathcal{E}_4 \vdash [\langle B_1 : \; \mathit{verify}(n, (y = x, \mathtt{\textbf{return}} = 2 * x)) \cdot \mathit{anReq}(M) \rangle \; ; \; \emptyset] & (\textsc{ReqDer})(5) \\
\hline
\mathcal{E}_3 \vdash [\langle B_1 : \; \mathit{anCalls}(\{\mathit{true}\} \; \mathtt{\textbf{return}} := n(x) \; \{\mathtt{\textbf{return}} = 2 * x\}) \cdot \mathit{anReq}(M) \rangle \; ; \; \emptyset] & (\textsc{IntCall})(4) \\
\hline
\mathcal{E}_2 \vdash [\langle B_1 : \; \mathit{verify}(m, (\mathit{true}, \mathtt{\textbf{return}} = 2 * x)) \cdot \mathit{anReq}(M) \rangle \; ; \; \emptyset] & (\textsc{ReqNotDer})(3) \\
\hline
\mathcal{E}_2 \vdash [\langle B_1 : \; \mathit{anSpec}(Sm) \cdot \mathit{anReq}(M) \rangle \; ; \; \emptyset] & (\textsc{NewSpec}) \\
\hline
\mathcal{E}_2 \vdash [\langle B_1 : \; \mathit{anCalls}(\mathtt{\textbf{return}} := 2 * y) \cdot \mathit{anSpec}(Sm) \cdot \mathit{anReq}(M) \rangle \; ; \; \emptyset] & (\textsc{Skip}) \\
\hline
\mathcal{E}_1 \vdash [\langle B_1 : \; \mathit{verify}(n, (\mathit{true}, \mathtt{\textbf{return}} = 2 * y)) \cdot \mathit{anSpec}(Sm) \cdot \mathit{anReq}(M) \rangle \; ; \; \emptyset] & (\textsc{ReqNotDer})(2) \\
\hline
\mathcal{E}_1 \vdash [\langle B_1 : \; \mathit{anSpec}(Sn) \cdot \mathit{anSpec}(Sm) \cdot \mathit{anReq}(M) \rangle \; ; \; \emptyset] & (\textsc{NewSpec}) \\
\hline
\mathcal{E}_1 \vdash [\langle B_1 : \; \mathit{anSpec}(\overline{MS}) \cdot \mathit{anReq}(M) \rangle \; ; \; \emptyset] & (\textsc{DecompSpec}) \\
\hline
\mathcal{E}_0 \vdash [\epsilon \, ; \, \mathtt{\textbf{class}} \; B_1 \; \mathtt{\textbf{extends}} \; A \; \{M \; \overline{MS}\}] & (\textsc{NewClass})(1) \\
\hline
\mathcal{E}_0 \vdash \mathit{module}(\mathtt{\textbf{class}} \; B_1 \; \mathtt{\textbf{extends}} \; A \; \{M \; \overline{MS}\}) & (\textsc{NewModule})
\end{array}
$$

(1) $A \in \mathcal{E}_0, B_1 \notin \mathcal{E}_0,$ and $\mathcal{E}_1 = \mathcal{E}_0 \oplus \mathit{extL}(B_1, A, \emptyset, M)$
(2) $\mathtt{\textbf{return}} := 2 * y \vdash_{PL} \mathit{body}_{\mathcal{E}_1}(B_1, n) : (\mathit{true}, \mathtt{\textbf{return}} = 2 * y)$ and
$\mathcal{E}_2 = \mathcal{E}_1 \oplus \mathit{extS}(B_1, n, (\mathit{true}, \mathtt{\textbf{return}} = 2 * y))$
(3) $\{\mathit{true}\} \; \mathtt{\textbf{return}} := n(x) \; \{\mathtt{\textbf{return}} = 2 * x\} \vdash_{PL} \mathit{body}_{\mathcal{E}_2}(B_1, m) : (\mathit{true}, \mathtt{\textbf{return}} = 2 * x)$ and
$\mathcal{E}_3 = \mathcal{E}_2 \oplus \mathit{extS}(B_1, m, (\mathit{true}, \mathtt{\textbf{return}} = 2 * x))$
(4) $\mathcal{E}_4 = \mathcal{E}_3 \oplus \mathit{extR}(B_1, n, (y = x, \mathtt{\textbf{return}} = 2 * x))$
(5) $S{\uparrow}_{\mathcal{E}_4}(B_1, n) = \{(\mathit{true}, \mathtt{\textbf{return}} = 2 * y)\},$ and $(\mathit{true}, \mathtt{\textbf{return}} = 2 * y) \rightharpoonup (y = x, \mathtt{\textbf{return}} = 2 * x)$
(6) $R{\uparrow}_{\mathcal{E}_4}(L_{\mathcal{E}_4}(B_1).\mathit{inh}, n) = R{\uparrow}_{\mathcal{E}_4}(A, n) = R_{\mathcal{E}_4}(A, n) = R_{\mathcal{E}_0}(A, n) = \{(y \geq 0, \mathtt{\textbf{return}} \geq 2 * y)\}$
(7) $S{\uparrow}_{\mathcal{E}_4}(B_1, n) = \{(\mathit{true}, \mathtt{\textbf{return}} = 2 * y)\},$ and $(\mathit{true}, \mathtt{\textbf{return}} = 2 * y) \rightharpoonup (y \geq 0, \mathtt{\textbf{return}} \geq 2 * y)$

Figure 9: Analysis details of class $B_1$. Due to space limitations, side conditions and auxiliary computations for the different rule applications are given as notes.

*Let $L \triangleq$ **`class`** $C$ **`extends`** $A$ $\{\overline{f} \; \overline{M} \; \overline{MS}\}$ be a class definition such that $C \notin \mathcal{E}$. Let $\mathcal{E} \vdash [\epsilon \; ; \; L \cup \mathcal{L}]$ be the judgment under evaluation by LBS(PL). Assume that $C$ is loaded for analysis and that the analysis of $C$ succeeds, leading to the judgment $\mathcal{E}' \vdash [\langle C : \epsilon \rangle \, ; \, \mathcal{L}']$. Then the following properties hold for $\mathcal{E}'$ and $\mathcal{L}'$:*

1. $L_{\mathcal{E}'} = L_{\mathcal{E}}[C \mapsto \langle A, \overline{f}, \overline{M} \rangle]$.
2. $A \neq \mathit{nil} \Rightarrow A \in \mathcal{E}$.
3. *For all $B \in \mathcal{E}$ and method name $m$, we have $S_{\mathcal{E}'}(B, m) = S_{\mathcal{E}}(B, m)$ and $R_{\mathcal{E}'}(B, m) = R_{\mathcal{E}}(B, m)$. For any class $B$ such that $B \notin \mathcal{E}'$, we have $S_{\mathcal{E}'}(B, m) = R_{\mathcal{E}'}(B, m) = \emptyset$ for all method names $m$.*
4. $\mathcal{L} \subseteq \mathcal{L}'$.
5. *For each $(p, q) \in S_{\mathcal{E}'}(C, m)$ for some method name $m$, there is a proof outline $O$ such that $O \vdash_{PL} \mathit{body}_{\mathcal{E}'}(C, m) : (p, q)$, $R_{\mathcal{E}'}(C, n) \rightharpoonup (r', s')$ for*

*each* $\{r\}\ v := n(\overline{e})\ \{s\}$ *in* $O$, $S{\uparrow}_{\mathcal{E}'}(G,n) \twoheadrightarrow (r',s')$ *for each* $\{r\}\ v :=$ $n@G(\overline{e})\ \{s\}$ *in* $O$, *and* $require(D,n,(r',s')) \in (\mathcal{L}' \setminus \mathcal{L})$ *for each* $\{r\}\ v :=$ $e.n(\overline{e})\ \{s\}$ *in* $O$ *where* $e : D$. *Here,* $r' = (r \wedge \overline{x} = \overline{e})$, $s' = s[\textbf{return}/v]$, *and* $\overline{x}$ *are the formal parameters of* $n$.

6. $S{\uparrow}_{\mathcal{E}'}(C,m) \twoheadrightarrow R{\uparrow}_{\mathcal{E}'}(C,m)$ *for all* $m$.

*Proof.* By rule (NewClass), the judgment under consideration leads to $\mathcal{E} \oplus extL(C,A,\overline{f},\overline{M}) \vdash [\langle C : anSpec(\overline{MS}) \cdot anReq(\overline{M})\rangle\ ;\ \mathcal{L}]$. The inference rules manipulates this judgment until $\mathcal{E}' \vdash [\langle C : \epsilon\rangle\ ;\ \mathcal{L}']$ is reached. Note that none of the rules (NewModule), (NewClass), (ExtReq), (EmpClass), and (EmpModule) can be applied during this manipulation.

*Condition 1.* The only rule that extends the $L$ mapping is (NewClass). This rule is applied when the class is loaded, and the condition follows from the premise of this rule.

*Condition 2.* Again, this follows from the premise of (NewClass).

*Condition 3.* This is proved by induction over the inference rules. None of the rules remove information from the environment, and the only sets that are extended are the $S$ and $R$ sets of class $C$. Thus, for all $B \neq C$ and methods $m$, we have $S_{\mathcal{E}}(B,m) = S_{\mathcal{E}'}(B,m)$ and $R_{\mathcal{E}}(B,m) = R_{\mathcal{E}'}(B,m)$. Especially, this holds for all classes defined in $\mathcal{E}$ as required by the first part of Condition 3.

Furthermore, if $B \notin \mathcal{E}'$, we know from Condition 1 that $B \neq C$ and $B \notin \mathcal{E}$. The conclusion $S_{\mathcal{E}'}(B,m) = R_{\mathcal{E}'}(B,m) = \emptyset$ then follows by the above paragraph and the assumption $S_{\mathcal{E}}(B,m) = R_{\mathcal{E}}(B,m) = \emptyset$ of the lemma.

*Condition 4.* By induction over the inference rules. During analysis of $C$, no elements are removed from $\mathcal{L}$.

*Condition 5.* By induction over the inference rules. Initially, the mapping $S_{\mathcal{E}}(C,m)$ is empty, thus for each $(p,q) \in S_{\mathcal{E}'}(C,m)$, rule (ReqNotDer) must have been applied. This rule ensures the existence of $O \vdash_{PL} body_{\mathcal{E}'}(C,m) : (p,q)$. For each $\{r\}\ v := n(\overline{e})\ \{s\}$ in $O$, Rule (IntCall) is applied, ensuring $R_{\mathcal{E}'}(C,n) \twoheadrightarrow (r',s')$. For each $\{r\}\ v := n@G(\overline{e})\ \{s\}$ in $O$, Rule (StaticCall) ensures $S{\uparrow}_{\mathcal{E}'}(G,n) \twoheadrightarrow (r',s')$. For each $\{r\}\ v := e.n(\overline{e})\ \{s\}$ in $O$, Rule (ExtCall) ensures $require(D,n,(r',s')) \in (\mathcal{L}' \setminus \mathcal{L})$ for $e : D$.

*Condition 6.* The mapping $R_{\mathcal{E}}(C,m)$ is initially empty. Thus, if $(r,s) \in R_{\mathcal{E}'}(C,m)$, rule (IntCall) must have been applied during the analysis of $C$. For each such $(r,s)$, this rule leads to an operation $verify(m,(r,s))$. This operation either succeeds by Rule (ReqDer) or (ReqNotDer). If (ReqDer) is applied $S{\uparrow}_{\mathcal{E}'}(C,m) \twoheadrightarrow (r,s)$ must hold. Otherwise, Rule (ReqNotDer) ensures

$S_{\mathcal{E}'}(C,m) \rightharpoonup (r,s)$. Combined, this gives $S{\uparrow}_{E'}(C,m) \rightharpoonup (r,s)$ for each $(r,s) \in R_{\mathcal{E}'}(C,m)$, i.e., $S{\uparrow}_{\mathcal{E}'}(C,m) \rightharpoonup R_{\mathcal{E}'}(C,m)$.

Consider next the requirements inherited by $C$. If $A = nil$ (where $A = L_{\mathcal{E}'}(C).inh$), the desired $S{\uparrow}_{\mathcal{E}'}(C,m) \rightharpoonup R{\uparrow}_{\mathcal{E}'}(C,m)$ follows directly by the definition of $R{\uparrow}$ in Figure 5. Otherwise, if $A \neq nil$, we must prove that $S{\uparrow}_{\mathcal{E}'}(C,m) \rightharpoonup R{\uparrow}_{\mathcal{E}'}(A,m)$. By $A \neq nil$, we know that $A \in \mathcal{E}$ and that $S{\uparrow}_{\mathcal{E}'}(A,m) \rightharpoonup R{\uparrow}_{\mathcal{E}'}(A,m)$ by Conditions 2 and 3 above. If $m \notin L_{\mathcal{E}'}(C).mtds$ then $S{\uparrow}_{\mathcal{E}'}(A,m) \subseteq S{\uparrow}_{\mathcal{E}'}(C,m)$, which gives $S{\uparrow}_{\mathcal{E}'}(C,m) \rightharpoonup R{\uparrow}_{\mathcal{E}'}(A,m)$. Otherwise, if $m \in L_{\mathcal{E}'}(C).mtds$, the method is analyzed by Rule (NewMtd), leading to a *verify* operation on each requirement in $R{\uparrow}_{\mathcal{E}'}(A,m)$. The analysis of these *verify* operations ensures $S_{\mathcal{E}'}(C,m) \rightharpoonup R{\uparrow}_{\mathcal{E}'}(A,m)$. Consequently, we have $S{\uparrow}_{\mathcal{E}'}(C,m) \rightharpoonup R{\uparrow}_{\mathcal{E}'}(A,m)$ also in this case since $S{\uparrow}_{\mathcal{E}'}(C,m) = S_{\mathcal{E}'}(C,m)$. □

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

*Proof.* By Rule (NewModule), the judgment $\mathcal{E} \vdash module(\overline{L})$ evaluates to $\mathcal{E} \vdash [\epsilon \,;\, \overline{L}]$. The analysis continues by manipulation of this judgment until the judgment $\mathcal{E}' \vdash [\epsilon \,;\, \emptyset]$ is reached.

Consider some class $C$ already defined in the initial environment $\mathcal{E}$. Since $\mathcal{E}$ is sound, Condition 3 of Lemma 2 ensures that the analysis performed for $C$ remains sound during analysis of the classes in $\overline{L}$.

Let $\mathcal{E}''$ be the environment in which some class $C$ in $\overline{L}$ is analyzed, i.e., the judgment $\mathcal{E}'' \vdash [\epsilon \,;\, \{\texttt{class}\ C\ \texttt{extends}\ A\ \{\overline{f}\ \overline{M}\ \overline{MS}\} \cup \mathcal{L}'']$ is manipulated in $LBS(PL)$, and let $\mathcal{E}'''$ be the environment immediately after analysis of this class, i.e., the judgment $\mathcal{E}''' \vdash [\langle C : \epsilon\rangle \,;\, \mathcal{L}''']$ is reached.

From Lemma 2, we know that if $S{\uparrow}_{\mathcal{E}''}(B,m) \rightharpoonup R{\uparrow}_{\mathcal{E}''}(B,m)$ for all classes $B$ where $B \neq C$, then $S{\uparrow}_{\mathcal{E}'''}(B,m) \rightharpoonup R{\uparrow}_{\mathcal{E}'''}(B,m)$, and that $S{\uparrow}_{\mathcal{E}'''}(C,m) \rightharpoonup R{\uparrow}_{\mathcal{E}'''}(C,m)$ is established. Thus, if Condition 2 of Definition 3 holds in $\mathcal{E}''$, then it also holds in $\mathcal{E}'''$. As none of the rules (ExtReq) and (EmpClass) extends the environment, we may conclude that Condition 2 of Definition 3 holds in the resulting environment $\mathcal{E}'$.

Consider next Condition 1 of Definition 3 for class $C$ defined in $\overline{L}$. By Lemma 2, Condition 5, we know that for each $(p,q) \in S_{\mathcal{E}'''}(C,m)$ there is a proof outline $O$ such that $O \vdash_{PL} body_{\mathcal{E}'''}(C,m)$ and that for each $\{r\}\ v := n(\overline{e})\ \{s\}$ in $O$, we have $R_{\mathcal{E}'''}(C,n) \rightharpoonup (r',s')$. For each $\{r\}\ v := n@G(\overline{e})\ \{s\}$

in $O$, we have $S{\uparrow_{\mathcal{E}'''}}(G,n) \rightharpoonup (r',s')$. For each $\{r\}\ v := e.n(\overline{e})\ \{s\}$ in $O$, we have *require*$(D,n,(r',s')) \in (\mathcal{L}''' \setminus \mathcal{L}'')$ for $e : D$. Thus, Condition 1 of Definition 3 is ensured for $\mathcal{E}'''$ except that external *require* operations have not been verified. By Lemma 2, Condition 3, we have $(p,q) \in S_{\mathcal{E}'}(C,m)$ and $R_{\mathcal{E}'}(C,n) \rightharpoonup (r',s')$ for each $\{r\}\ v := n(\overline{e})\ \{s\}$ in $O$ also for the resulting environment $\mathcal{E}'$. Furthermore, since the analysis of each *require*$(D,n,(r',s'))$ operation succeed, Rule (ExtReq) is applied in some environment $\mathcal{F}$, where $\mathcal{F}$ either occurs between analysis of two classes, or $\mathcal{F}$ is the resulting environment $\mathcal{E}'$. In either case, Rule (ExtReq) ensures $D \in \mathcal{F}$ and $R{\uparrow_{\mathcal{F}}}(D,m) \rightharpoonup (r',s')$ for each $\{r\}\ v := e.n(\overline{e})\ \{s\}$, $e : D$, in $O$. If subsequent classes are analyzed, we have $R{\uparrow_{\mathcal{E}'}}(D,m) \rightharpoonup (r',s')$ by Lemma 2, Condition 3. Thereby, Condition 1 of Definition 3 is established for $\mathcal{E}'$.

Since the initial environment $\mathcal{E}$ is sound, Condition 3 of Definition 3 holds in $\mathcal{E}$. By Condition 2 and 3 of Lemma 2, this proof condition for sound environments is maintained by analysis of each class in $\overline{L}$, which means that also the last condition for sound environments holds for the resulting environment $\mathcal{E}'$. □

**Theorem 2 (Soundness).** *If PL is a sound program logic, then LBS(PL) 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 1 guarantee that the environment remains sound during the analysis of class 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 (IntCall) is the only rule that extends $R_{\mathcal{E}}(C,m)$. In order for the rule to be applied, an operation *anCalls*$(\{r\}\ v := m(\overline{e})\ \{s\})$ must be analyzed in the context of $C$ for some requirement $(r,s)$ 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 specifications are remembered.

**Lemma 4.** *Let* $\mathcal{E}$ *be an environment and* $\overline{L}$ *a list of class declarations. Whenever a proof outline* $O$ *such that* $O \vdash_{PL} body_{\mathcal{E}}(C, m) : (p, q)$ *is verified during analysis of some class* $C$ *in* $\overline{L}$*, the specification* $(p, q)$ *is included in* $S_{\mathcal{E}}(C, m)$*.*

*Proof.* By induction over the inference rules. The only rule requiring the verification of 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 $O \vdash_{PL} body_{\mathcal{E}}(C, m) : (p, q)$ is verified in $PL$. □

## 5. External Specification by Interfaces

In the approach presented so far, each class $C$ provides some specifications of the available methods, inherited or defined, in the form of assertion pairs. These are kept in the $S$ part of the proof environments. Their verification generates $R$ requirements for the late bound internal calls occurring in the class, which are imposed on subclass redefinitions of the called methods. In a subclass, redefined methods are allowed to violate the $S$ specifications of a superclass, but not the $R$ requirements.

A weakness of $LBS(PL)$ concerns the treatment of external calls (as opposed to internal calls): When reasoning about $e.m(\overline{e})$ with $e : E$, the pre/-post assertion of the call must follow from the $R$ requirements to $m$ that have been established for class $E$, or otherwise by adding the corresponding requirement to $E$ and verifying that it is holds for that class and any subclasses. Thus, class $E$ and any subclasses may need to be analyzed again with respect to the new requirement. As $R$ requirements generated from internal calls may not in general provide suitable external properties, as illustrated by the next example, reverification will be needed.

**Example 12.** Reconsider the class $A$ from Example 9:

```
class A {
  int n(int y) : (true, return = 5 * y) {return := 5*y}
  int m(int x) : (x ≥ 0, return ≥ 2 * x) {return := n(x)}
}
```

As explained in Example 9, the specification and requirement sets are built as follows during the analysis of $A$:

$$\begin{aligned} S(A,n) &= \{(true, \mathbf{return} = 5 * y)\} \\ S(A,m) &= \{(x \geq 0, \mathbf{return} \geq 2 * x)\} \\ R(A,n) &= \{(y \geq 0, \mathbf{return} \geq 2 * y)\} \end{aligned}$$

Note that the internal call to method $n$ gave a requirement in $R(A,n)$, whereas no requirements are recorded for method $m$ since $m$ is not called internally. Consider next the following client code:

```
class Client {
  A a := new A;
  int d1() : (true,return ≥ 10) {return := a.n(5)}
  int d2() : (true,return ≥ 10) {return := a.m(5)}
}
```

The analysis of the specification of method *d1* leads to a requirement on the call to $a.n$: $(y = 5, \mathbf{return} \geq 10)$. By Rule (ExtCall), this requirement generates an operation $require(A, n, (y = 5, \mathbf{return} \geq 10))$, since $A$ is the type of $a$. Correspondingly, analysis of the external call in *d2* gives an operation $require(A, m, (x = 5, \mathbf{return} \geq 10))$. In this manner, the analysis of *Client* generates two *require* operations. Rule (ExtReq) is the only rule that applies to these operations.

For the operation $require(A, n, (y = 5, \mathbf{return} \geq 10))$, Rule (ExtReq) requires that the relation $R(A,n) \rightharpoonup (y = 5, \mathbf{return} \geq 10)$ must hold, which follows by the definition of entailment since the following implication holds:

$$(y = 5 \wedge (y \geq 0 \Rightarrow \mathbf{return} \geq 2 * y)) \Rightarrow \mathbf{return} \geq 10$$

Since the requirement follows from $R(A,n)$, it is guaranteed to hold also if the call binds to an instance of a subclass of $A$, such as class $B_1$ in Example 9.

For the operation $require(A, m, (x = 5, \mathbf{return} \geq 10))$ however, the relation $R(A,m) \rightharpoonup (x = 5, \mathbf{return} \geq 10)$ *does not* hold, since $R(A,m)$ is empty. Therefore, the analysis of the specification for *d2* requires reverification of $A$ and of any subclasses of $A$.

The situation illustrated in Example 12 is not desirable since previously analyzed classes must be analyzed again.

In this section we use *behavioral interfaces* as a means to specify and reason about requirements on external method calls [14, 52]. A behavioral

$$
\begin{array}{lcl}
P & ::= & \overline{KL}\ \{t\} \\
KL & ::= & K \mid L \\
L & ::= & \texttt{\textbf{class}}\ C\ \texttt{\textbf{extends}}\ C\ \texttt{\textbf{implements}}\ I\ \{\overline{F}\ \overline{M}\ \overline{MS}\} \\
K & ::= & \texttt{\textbf{interface}}\ I\ \texttt{\textbf{extends}}\ \overline{I}\ \{\overline{MS}\} \\
T & ::= & \texttt{\textbf{nat}} \mid \texttt{\textbf{int}} \mid \texttt{\textbf{bool}} \mid I
\end{array}
$$

Figure 10: Syntax for the language *IOOL*, extending the syntactic category $L$ of *OOL* (see Figure 1) with interfaces. Types now range over interface names instead of class names. The other syntactic categories of Figure 1 remain unchanged. Here, $I$ denotes interface names of type *Iid*.

interface describes the visible methods of a class and their specifications, and inheritance may be used to form new interfaces from old ones. These behavioral interfaces are used to type object variables (references), and subtyping follows the interface inheritance hierarchy. A class definition explicitly declares which interface it implements. (For simplicity we consider at most one interface per class.) This allows the inheritance hierarchies of interfaces and classes to be kept distinct. Static type checking of an assignment $v := e$ must then ensure that the expression $e$ denotes an object supporting the declared interface of the object variable $v$. In this setting, the substitution principle for objects can be reformulated as follows: *For an object variable $v$ with declared interface $I$, the actual object referred to by $v$ at run-time will satisfy the behavioral specification $I$.* Reasoning about an external call $e.m(\overline{e})$ can then be done by relying on the behavioral interface of the object expression $e$, simplifying the (ExtCall) rule presented above to simply check interface requirements. In this way, *require* operations are no longer needed in the proof system.

In Section 5.1, we define the programming language *IOOL*, which extends *OOL* with interfaces. In Section 5.2 we define proof environments of type *IEnv* where interface information is accounted for, and in Section 5.3 we define the calculus *LBSI*(*PL*) for reasoning about *IOOL* programs.

### *5.1. Behavioral Interfaces*

Let the programming language *IOOL* extend *OOL* with behavioral interfaces. In the syntax for *IOOL*, given in Figure 10, classes are modified such that a class implements a single interface. Note that the types of variables and methods no longer range over class names, as object references are typed by interface names. A *behavioral interface* $I$ may extend a list $\overline{I}$ of super-

interfaces, and consists of a set $\overline{MS}$ of method names with signatures and semantic constraints on the use of these methods. The constraints are given as $(pre, post)$ specifications for the methods. An interface may declare signatures of new methods not found in its superinterfaces, and it may declare additional specifications of methods declared in the superinterfaces. The inheritance relationship between interfaces is restricted to a form of behavioral subtyping. Consequently, an interface may not declare method specifications that are in conflict with the specifications declared by the superinterfaces. In the sequel, it is assumed that the interface hierarchy conforms to these requirements. The interfaces thus form a type hierarchy: if $I'$ extends $I$, then $I'$ is a *subtype* of $I$ and $I$ is a *supertype* of $I'$. Let $\preceq$ denote the reflexive and transitive subtype relation, which is given by the nominal extends-relation over interfaces under the assumption above. Thus, $I' \preceq I$ if $I'$ equals $I$ or if $I'$ (directly or indirectly) extends $I$. An interface $I$ *exports* the methods declared in $I$ or in the superinterfaces of $I$, with the associated constraints (or requirements) on method use.

A class $C$ *implements* $I$ if it has an **`implements`** $I$ in the class definition and all methods exported by $I$ are defined, satisfying the constraints of $I$. The analysis of the class must ensure that this requirement holds. Observe that only the methods exported by $I$ are available for external invocations on references typed by $I$. The class may implement additional *auxiliary* methods for internal use. Inside a class the type of **`this`** is the interface implemented by the class. By type safety, external calls which bind to **`this`** can be assumed to be safe also when the calls are executed on an instance of a subclass of $C$. An instance of $C$ is said to *support* $I$ and all superinterfaces of $I$; thus ensuring that the object provides the methods exported by $I$ and adheres to the specifications imposed by $I$ on these methods. Objects of different classes may support the same interface, corresponding to different implementations of the interface behavior. If an object supports $I$ (or a subtype of $I$) then the object may be referenced by a variable typed by $I$. The separation of class and inheritance hierarchies means that a subclass $D$ of $C$ need not implement (a subtype of) the interface $I$ implemented by $C$ [4, 31]: If $D$ implements $J$, then $J$ need not be a subtype of $I$. In this case, the subclass may freely reuse and redefine superclass methods without adhering to the behavioral constraints imposed by $I$, and instances of $D$ will not behave as subtypes of $I$.

**Example 13.** Let $A$ be a class implementing an interface $I$ as depicted in

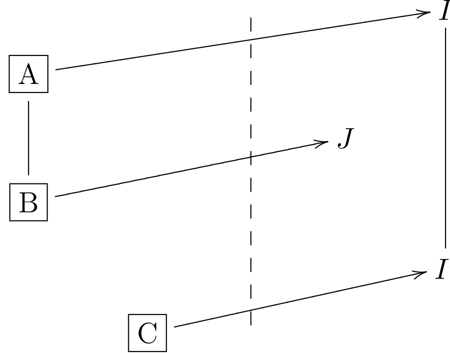


Figure 11: A graphical representation of the class and type hierarchy of Example 13: $A, B$, and $C$ are classes and $I, J$, and $I'$ are interfaces. The arrows indicate **implements** relationships; e.g., $A$ implements $I$, Solid vertical lines indicate **extends** relationships; i.e., $B$ extends $A$ and $I'$ extends $I$. The dashed line indicates the separation of the class hierarchy from the interface hierarchy. Note that instances of class $C$ support $I$, whereas instances of class $B$ do not support $I$.

Figure 11, thus instances of $A$ support $I$. A variable $x$ declared with type $I$ (i.e., $x : I$) may refer to an instance of $A$. A subclass $B$ of $A$ may reuse the code of $A$ without implementing the interface $I$; i.e., $B$ may extend $A$ but implement a different interface $J$ where $J$ is not a subtype if $I$. In that case, the substitution principle will ensure that $x$ never refers to an instance of $B$. Given a third class $C$ implementing $I'$, where $I'$ is a subinterface of $I$, the variable $x$ may refer to an instance of $C$ since $C$ implements the subinterface $I'$ of $I$. Assuming that these are the only classes and interfaces in the system, an assignment $x := e$ is type safe if $e$ denotes an expression of either type $I$ or type $I'$. When reasoning about an external method call $x.m()$, we can rely on the behavioral constraints of $m$ given by the type $I$ of $x$.

A variable $y : J$ may refer to an instance of $B$, but the substitution principle prohibits $y$ from referring to an instance of $A$ or $C$; e.g., the assignment $y := x$ is not type safe. Correspondingly, also the assignment $x := y$ is not type safe. Thus, the substitution principle applies to the supported interface of an object, ensuring that a subclass instance cannot be accessed through the type of a superclass unless the subclass explicitly implements the superclass type.

*5.2. A Proof Environment with Interfaces*

As before, classes are analyzed in the context of a proof environment. Let *Interface* denote interface tuples $\langle \overline{I}, \overline{MS} \rangle$, and *IClass* denote class tuples

$$
\begin{array}{ll}
\mathit{mids}(\emptyset) & \triangleq \emptyset \\
\mathit{mids}(m(\overline{x}) : (p,q)\ \overline{MS}) & \triangleq \{m\} \cup \mathit{mids}(\overline{MS}) \\
\mathit{public}_{\mathcal{E}}(\mathit{nil}) & \triangleq \emptyset \\
\mathit{public}_{\mathcal{E}}(I) & \triangleq \mathit{mids}(K_{\mathcal{E}}(I).\mathit{mtds}) \cup \mathit{public}_{\mathcal{E}}(K_{\mathcal{E}}(I).\mathit{inh}) \\
\mathit{public}_{\mathcal{E}}(I\ \overline{I}) & \triangleq \mathit{public}_{\mathcal{E}}(I) \cup \mathit{public}_{\mathcal{E}}(\overline{I}) \\
\mathit{specs}(\emptyset, m) & \triangleq \emptyset \\
\mathit{specs}(n(\overline{x}) : (p,q)\ \overline{MS}, m) & \triangleq \textbf{if } n = m \textbf{ then } \{(p,q)\} \cup \mathit{specs}(\overline{MS}, m) \\
 & \qquad\qquad\quad \textbf{else } \mathit{specs}(\overline{MS}, m) \textbf{ fi} \\
\mathit{spec}_{\mathcal{E}}(\mathit{nil}, m) & \triangleq \emptyset \\
\mathit{spec}_{\mathcal{E}}(I, m) & \triangleq \mathit{specs}(K_{\mathcal{E}}(I).\mathit{mtds}, m) \cup \mathit{spec}_{\mathcal{E}}(K_{\mathcal{E}}(I).\mathit{inh}, m) \\
\mathit{spec}_{\mathcal{E}}(I\ \overline{I}, m) & \triangleq \mathit{spec}_{\mathcal{E}}(I, m) \cup \mathit{spec}_{\mathcal{E}}(\overline{I}, m) \\
\mathit{nil} \preceq_{\mathcal{E}} J & \triangleq \mathit{false} \\
I \preceq_{\mathcal{E}} J & \triangleq I = J \vee K_{\mathcal{E}}(I).\mathit{inh} \preceq_{\mathcal{E}} J \\
(I\ \overline{I}) \preceq_{\mathcal{E}} J & \triangleq I \preceq_{\mathcal{E}} J \vee \overline{I} \preceq_{\mathcal{E}} J
\end{array}
$$

Figure 12: Auxiliary function definitions, using space as the list separator.

$\langle B, I, \overline{f}, \overline{M} \rangle$. Assuming type safety, we ignore the types of fields and methods. The list of superinterfaces $\overline{I}$ and method specifications $\overline{MS}$ of an interface tuple are accessible by the observer functions *inh* and *mtds*, respectively. The supported interface of a class is accessible by the observer function *impl*. Environments of type *IEnv* are defined as follows.

**Definition 5. (Proof environments with interfaces.)** A proof environment $\mathcal{E}$ of type *IEnv* is a tuple $\langle L_{\mathcal{E}}, K_{\mathcal{E}}, S_{\mathcal{E}}, R_{\mathcal{E}} \rangle$ where $L_{\mathcal{E}} : \mathit{Cid} \to \mathit{IClass}$, $K_{\mathcal{E}} : \mathit{Iid} \to \mathit{Interface}$ are partial mappings and $S_{\mathcal{E}}, R_{\mathcal{E}} : \mathit{Cid} \times \mathit{Mid} \to \mathit{Set}[APair]$ are total mappings.

For an interface $I$, let $I \in \mathcal{E}$ denote that $K_{\mathcal{E}}(I)$ is defined, and let $\mathit{public}(I)$ denote the set of method names exported by $I$; thus, $m \in \mathit{public}(I)$ if $m$ is declared by $I$ or by a supertype of $I$. A subtype cannot remove methods declared by a supertype, so $\mathit{public}(I) \subseteq \mathit{public}(I')$ if $I' \preceq I$. If $m \in \mathit{public}(I)$, the function $\mathit{spec}(I, m)$ returns a *set* of type $\mathit{Set}[APair]$ with the behavioral constraints imposed on $m$ by $I$, as declared in $I$ or in a supertype of $I$.

The function *spec* returns a set as a subinterface may provide additional specifications of methods inherited from superinterfaces; if $m \in \mathit{public}(I)$ and $I' \preceq I$, then $\mathit{spec}(I,m) \subseteq \mathit{spec}(I',m)$. These functions are defined in Figure 12. The superinterface name may be *nil*, representing no interface.

The definition of sound environments is revised to account for interfaces. In Condition 1, the requirement to an external call must now follow from the interface specification of the called object. Consider a requirement stemming from the analysis of an external call $e.m(\overline{e})$ in some proof outline, where $e : I$. As the interface hides the actual class of the object referenced by $e$, the call is analyzed based on the interface specification of $m$. A requirement $(r,s)$ must follow from the specification of $m$ given by type $I$, expressed by $\mathit{spec}(I,m) \rightharpoonup (r,s)$. Furthermore, a new condition of sound environments is introduced, expressing that a class satisfies the specifications of the implemented interface. If $C$ implements an interface $I$, the class defines (or inherits) an implementation of each $m \in \mathit{public}(I)$. For each such method, the behavioral specification declared by $I$ must follow from the method specification in the class; i.e., $S{\uparrow}(C,m) \rightharpoonup \mathit{spec}(I,m)$.

**Definition 6. (Sound environments.)** A proof environment $\mathcal{E}$ of type *IEnv* is sound if it satisfies the following conditions for each $C : \mathit{Cid}$ and $m : \mathit{Mid}$.

1. $\forall (p,q) \in S_{\mathcal{E}}(C,m)\ .\ \exists O\ .\ O \vdash_{PL} \mathit{body}_{\mathcal{E}}(C,m) : (p,q)$
   $\wedge\ \forall \{r\}\ v := n(\overline{e})\ \{s\} \in O\ .\ R_{\mathcal{E}}(C,n) \rightharpoonup (r',s')$
   $\wedge\ \forall \{r\}\ v := n@A(\overline{e})\ \{s\} \in O\ .\ S{\uparrow}_{\mathcal{E}}(A,n) \rightharpoonup (r',s')$
   $\wedge\ \forall \{r\}\ v := e.n(\overline{e})\ \{s\} \in O\ .\ e : I \Rightarrow \mathit{spec}_{\mathcal{E}}(I,n) \rightharpoonup (r',s')$
2. $S{\uparrow}_{\mathcal{E}}(C,m) \rightharpoonup R{\uparrow}_{\mathcal{E}}(C,m)$
3. $\forall n \in \mathit{public}_{\mathcal{E}}(I)\ .\ S{\uparrow}_{\mathcal{E}}(C,n) \rightharpoonup \mathit{spec}_{\mathcal{E}}(I,n)$, where $I = L_{\mathcal{E}}(C).\mathit{impl}$
4. $L_{\mathcal{E}}(C).\mathit{inh} \neq \mathit{nil} \Rightarrow L_{\mathcal{E}}(C).\mathit{inh} \in \mathcal{E}$
   $\wedge\ \forall B\ .\ B \notin \mathcal{E}\ .\ S_{\mathcal{E}}(B,m) = R_{\mathcal{E}}(B,m) = \emptyset.$

where $r' = (r \wedge \overline{x} = \overline{e})$, $s' = s[\mathbf{return}/v]$, $\overline{x}$ are the formal parameters of $n$, and $I : \mathit{Iid}$.

Lemma 1 is adapted to the setting of interfaces as follows:

**Lemma 5.** *Given a sound environment $\mathcal{E}$ : IEnv and a sound program logic PL. For all classes $C$ : Cid, methods $m$ : Mid, and assertion pairs $(p,q)$ : APair such that $C \in \mathcal{E}$ and $(p,q) \in S{\uparrow}_{\mathcal{E}}(C,m)$, we have $\models_D m(\overline{x}):(p,q)\,\{\mathit{body}_{\mathcal{E}}(C,m)\}$ for each $D \leq_{\mathcal{E}} C$.*

*Proof.* The proof is similar to the proof for Lemma 1, except for the treatment of external calls in the induction step. Let $O$ be a proof outline such that $O \vdash_{PL} body_{\mathcal{E}}(B,m) : (p,q)$, where $C \leq_{\mathcal{E}} B$, $bind_{\mathcal{E}}(C,\, m) = bind_{\mathcal{E}}(B,\, m)$, and $(p,q) \in S_{\mathcal{E}}(B,m)$. Assume as the induction hypothesis that for any external call to $n$ in $O$, possibly bound in context $E$ and for all $(g,h) \in S{\uparrow}_{\mathcal{E}}(E,n)$, that $\models_E n(\overline{x}) {:} (g,h)\, \{body_{\mathcal{E}}(E,n)\}$.

Consider a method call $\{r\}\ v := e.n(\overline{e})\ \{s\}$ in $O$. Let $r', s'$ be as in Definition 6, and $e : I$. From Definition 6, Condition 1, we have $spec_{\mathcal{E}}(I,n) \rightarrow (r',s')$. Consider some class $E$ where $L_{\mathcal{E}}(E).impl = J$. From Definition 6, Condition 3, we have $S{\uparrow}_{\mathcal{E}}(E,n) \rightarrow spec_{\mathcal{E}}(J,n)$. If the call to $n$ can bind in context $E$, then type safety ensures $J \preceq_{\mathcal{E}} I$, giving $spec_{\mathcal{E}}(I,n) \subseteq spec_{\mathcal{E}}(J,n)$. We then have $S{\uparrow}_{\mathcal{E}}(E,n) \rightarrow spec_{\mathcal{E}}(J,n) \rightarrow spec_{\mathcal{E}}(I,n) \rightarrow (r',s')$. By the induction hypothesis, we then arrive at $\models_E \{r'\}\ body_{\mathcal{E}}(E,n)\ \{s'\}$. $\square$

We define an operation to update a proof environment with a new interface, and redefine the operation for loading a new class:

$$\begin{aligned}
\mathcal{E} \oplus extL(C,B,I,\overline{f},\overline{M}) \quad &\triangleq \quad \langle L_{\mathcal{E}}[C \mapsto \langle B,I,\overline{f},\overline{M}\rangle], K_{\mathcal{E}}, S_{\mathcal{E}}, R_{\mathcal{E}}\rangle \\
\mathcal{E} \oplus extK(I,\overline{I},\overline{MS}) \quad &\triangleq \quad \langle L_{\mathcal{E}}, K_{\mathcal{E}}[I \mapsto \langle \overline{I},\overline{MS}\rangle], S_{\mathcal{E}}, R_{\mathcal{E}}\rangle
\end{aligned}$$

### *5.3. The Calculus LBSI(PL) for Lazy Behavioral Subtyping with Interfaces*

In the calculus for lazy behavioral subtyping with interfaces, judgments have the form $\mathcal{E} \vdash \mathcal{M}$, where $\mathcal{E}$ is the proof environment and $\mathcal{M}$ is a sequence of interfaces and classes. As before, we assume that superclasses appear before subclasses. This ordering ensures that requirements imposed by superclasses are verified in an incremental manner on subclass overridings. Furthermore, we assume that an interface appears before it is used. More precisely we assume that whenever a class is analyzed, the supported interface is already part of the environment, and for each external call statement $v := e.m(\overline{e})$ in the class where $e : I$, the interface $I$ is in the environment. These assumptions ensure that the analysis of a class will not be blocked due to a missing superclass or interface.

As the requirements of external calls are now verified against the interface specifications of the called methods, a complete analysis of a class $C$ can be performed based on the knowledge of its superclasses only; other classes need not be considered in order to analyze $C$. For the revised calculus, it therefore suffices to consider individual classes and interfaces as the granularity of program analysis. The module layer of Section 4 is therefore omitted. The

$$\frac{I \notin \mathcal{E} \qquad \overline{I} \neq nil \Rightarrow \overline{I} \in \mathcal{E} \qquad \mathcal{E} \oplus extK(I, \overline{I}, \overline{MS}) \vdash \mathcal{P}}{\mathcal{E} \vdash (\texttt{\textbf{interface}}\ I\ \texttt{\textbf{extends}}\ \overline{I}\ \{\overline{MS}\}) \cdot \mathcal{P}} \ (\textsc{NewInt})$$

$$\frac{\begin{array}{c} I \in \mathcal{E} \qquad C \notin \mathcal{E} \qquad B \neq nil \Rightarrow B \in \mathcal{E} \\ \mathcal{E} \oplus extL(C, B, I, \overline{f}, \overline{M}) \vdash \\ \langle C :\ anSpec(\overline{MS}) \cdot anReq(\overline{M}) \cdot intSpec(public_{\mathcal{E}}(I))\rangle \cdot \mathcal{P} \end{array}}{\mathcal{E} \vdash (\texttt{\textbf{class}}\ C\ \texttt{\textbf{extends}}\ B\ \texttt{\textbf{implements}}\ I\ \{\overline{f}\ \overline{M}\ \overline{MS}\}) \cdot \mathcal{P}} \ (\textsc{NewClass}')$$

$$\frac{e : I \qquad I \in \mathcal{E} \qquad spec_{\mathcal{E}}(I, m) \rightharpoondown (r', s') \qquad \mathcal{E} \vdash \langle C :\ \mathcal{O}\rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C :\ anCalls(\{r\}\ v := e.m(\overline{e})\ \{s\}) \cdot \mathcal{O}\rangle \cdot \mathcal{P}} \ (\textsc{ExtCall}')$$

$$\frac{S{\uparrow_{\mathcal{E}}}(C, m) \rightharpoondown spec_{\mathcal{E}}(L_{\mathcal{E}}(C).impl, m) \qquad \mathcal{E} \vdash \langle C :\ \mathcal{O}\rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C :\ intSpec(m) \cdot \mathcal{O}\rangle \cdot \mathcal{P}} \ (\textsc{IntSpec})$$

$$\frac{\mathcal{E} \vdash \langle C :\ intSpec(\overline{m_1}) \cdot intSpec(\overline{m_2}) \cdot \mathcal{O}\rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C :\ intSpec(\overline{m_1} \cup \overline{m_2}) \cdot \mathcal{O}\rangle \cdot \mathcal{P}} \ (\textsc{DecompInt})$$

Figure 13: The extension of the inference system, where $\mathcal{P}$ is a (possibly empty) sequence of classes and interfaces. Rules (NewClass′) and (ExtCall′) replace (NewClass) and (ExtCall). The three other rules, concerning interfaces, are new. In Rule (ExtCall′), we have $r' = (r \wedge \overline{x} = \overline{e})$ and $s' = s[\texttt{\textbf{return}}/v]$, where $\overline{x}$ are the formal parameters of $m$.

syntax for analysis operations is given by:

$$\begin{array}{ll} \mathcal{M} ::= \mathcal{P} \mid \langle C :\ \mathcal{O}\rangle \cdot \mathcal{P} & \mathcal{O} ::= \epsilon \mid anReq(\overline{M}) \mid anSpec(\overline{MS}) \mid anCalls(t) \\ \mathcal{P} ::= K \mid L \mid \mathcal{P} \cdot \mathcal{P} & \qquad \mid verify(m, \overline{R}) \mid intSpec(\overline{m}) \mid \mathcal{O} \cdot \mathcal{O} \end{array}$$

The new operation $intSpec(\overline{m})$ is used to analyze the interface specifications of methods $\overline{m}$ with regard to implementations found in the considered class.

For *IOOL*, we define a calculus $LBSI(PL)$, consisting of a (sound) program logic $PL$, a proof environment $\mathcal{E} : IEnv$, and the inference rules listed in Figure 13. In addition to the rules in Figure 13, $LBSI(PL)$ contains the rules in Figure 7 and Figure 8, except the rules (NewClass), (ExtCall), (NewModule), (EmpModule) and (ExtReq). Rules (NewClass) and (ExtCall) are renewed as shown in Figure 13, and rule (ExtReq) is superfluous as the requirements from external calls are analyzed in terms of interface specifications. Rules (NewModule) and (EmpModule) are not needed as modules are removed. For the remaining rules in Figure 7 and Figure 8, we assume that module operations are removed as illustrated by (NewClass′) and (ExtCall′).

Focusing on the changes from Figure 7 and Figure 8, the calculus rules are outlined in Figure 13. Rule (NewInt) extends the environment with a new interface. No analysis of the interface is needed at this point, the specifications of the interface will later be analyzed with regard to each class that implements the interface. (Recall that interfaces are assumed to appear in the sequence $\mathcal{P}$ before they are used.) The rule (NewClass′) is similar to the rule from $LBS(PL)$, except that an operation *intSpec* is introduced which is used to analyze the specifications of the implemented interface. Rule (ExtCall′) handles the analysis of external calls; here, the requirement of the call is analyzed with regard to the interface specification of the callee. Rule (IntSpec) is used to verify interface specifications, and rule (DecompInt) is used to flatten the argument of *intSpec* operations.

In $LBSI(PL)$, the different method specifications play a more active role when analyzing classes. Method specifications are used to establish interface properties, which again are used during the analysis of external calls. Thus, requirements to external calls are no longer analyzed based on knowledge from the $R$ mapping of the callee. The $R$ mapping is only used during the analysis of internal calls.

*Soundness.* For soundness of $LBSI(PL)$, Theorem 1 is modified as follows.

**Theorem 3.** *Let PL be a sound program logic, $\mathcal{E}$:IEnv a sound environment, and $KL$ be an interface or a class definition. If a proof of $\mathcal{E} \vdash KL$ in LBSI(PL) has $\mathcal{E}'$ as its resulting proof environment, then $\mathcal{E}'$ is also sound.*

*Proof.* The analysis of a new interface maintains soundness as interfaces are assumed to be loaded in the environment before they are used. Consider analysis of the judgment

$$\mathcal{E} \vdash (\texttt{class}\ C\ \texttt{extends}\ A\ \texttt{implements}\ I\ \{\overline{f}\ \overline{M}\ \overline{MS}\}) \quad (3)$$

The analysis of $C$ succeeds, leading to the judgment $\mathcal{E}' \vdash \langle C : \epsilon\rangle$ where the operation $\langle C : \epsilon\rangle$ is discarded by Rule (EmpClass), yielding the resulting environment $\mathcal{E}'$.

Since $\mathcal{E}'$ is the environment resulting by analyzing $C$ in the initial environment $\mathcal{E}$, we have $L_{\mathcal{E}'} = L_{\mathcal{E}}[C \mapsto \langle A, I, \overline{f}, \overline{M}\rangle]$, and for all $B \in \mathcal{E}$ and methods $m$ that $S_{\mathcal{E}'}(B, m) = S_{\mathcal{E}}(B, m)$ and $R_{\mathcal{E}'}(B, m) = R_{\mathcal{E}}(B, m)$. Especially, for $B \in \mathcal{E}$ and for each $(p, q) \in S_{\mathcal{E}}(B, m)$, we know that Condition 1

of Definition 6 holds also for $\mathcal{E}'$. Furthermore, since the analysis of $C$ does not modify the $K$ mapping, we have $public_{\mathcal{E}'}(I) = public_{\mathcal{E}}(I)$.

For the analysis of a class $C$, we consider each condition of Definition 6 by itself.

Condition 1 of Definition 6 applies to each element $(p, q)$ of $S_{\mathcal{E}'}(C, m)$. The proof is by induction over the inference rules, and it suffices to consider rule (ReqNotDer) which is the only rule that extends the $S$ mapping. If $(p, q) \in S_{\mathcal{E}'}(C, m)$, this rule ensures the existence of a proof outline $O$ such that $O \vdash_{PL} body_{\mathcal{E}'}(C, m) : (p, q)$. The analysis then continues with an $anCalls(O)$ operation. For each decorated late bound internal call $\{r\}\ v := n(\overline{e})\ \{s\}$ in $O$, rule (IntCall) ensures $R_{\mathcal{E}'}(C, n) \rightharpoonup (r', s')$ as required by Definition 6. For each static call $\{r\}\ v := n@G(\overline{e})\ \{s\}$ in $O$, rule (StaticCall) ensures $S{\uparrow_{\mathcal{E}'}}(G, n) \rightharpoonup (r', s')$ as required by Definition 6. For each external call $\{r\}\ v := e.n(\overline{e})\ \{s\}$ in $O$ where $e : I$, rule (ExtCall′) ensures $spec_{\mathcal{E}'}(I, n) \rightharpoonup (r', s')$ as required by Definition 6.

Consider next Condition 2 of Definition 6. Since $\mathcal{E}$ is sound, we may assume $S{\uparrow_{\mathcal{E}}}(B, m) \rightharpoonup R{\uparrow_{\mathcal{E}}}(B, m)$ for any $B \in \mathcal{E}$ and method $m$. By the above discussion, we then have $S{\uparrow_{\mathcal{E}'}}(B, m) \rightharpoonup R{\uparrow_{\mathcal{E}'}}(B, m)$. Consider first the requirements in $R_{\mathcal{E}'}(C, m)$ for some method $m$. For each $(r, s) \in R_{\mathcal{E}'}(C, m)$, Rule (IntCall) must have been applied during the analysis of $C$, generating an operation $verify(m, (r, s))$ which is analyzed in the context of $C$. Analysis of these $verify$ operations succeed either by Rule (ReqDer) of (ReqNotDer), ensuring $S{\uparrow_{\mathcal{E}'}}(C, m) \rightharpoonup R_{\mathcal{E}'}(C, m)$. If $C$ has no superclass (i.e., $L_{\mathcal{E}'}(C).inh = nil$), the relation $S{\uparrow_{\mathcal{E}'}}(C, m) \rightharpoonup R{\uparrow_{\mathcal{E}'}}(C, m)$ then follows directly. Otherwise, we have $A = L_{\mathcal{E}'}(C).inh$ and $A \in \mathcal{E}$, i.e. $S{\uparrow_{\mathcal{E}'}}(A, m) \rightharpoonup R{\uparrow_{\mathcal{E}'}}(A, m)$ holds. For class $C$, we have $R{\uparrow_{\mathcal{E}'}}(C, m) = R{\uparrow_{\mathcal{E}'}}(A, m) \cup R_{\mathcal{E}'}(C, m)$. Since $S{\uparrow_{\mathcal{E}'}}(C, m) \rightharpoonup R_{\mathcal{E}'}(C, m)$, we need to establish $S{\uparrow_{\mathcal{E}'}}(C, m) \rightharpoonup R{\uparrow_{\mathcal{E}'}}(A, m)$. We consider two cases, $m \notin L_{\mathcal{E}'}(C).mtds$ and $m \in L_{\mathcal{E}'}(C).mtds$. If $m \notin L_{\mathcal{E}'}(C).mtds$, we have $S{\uparrow_{\mathcal{E}'}}(C, m) = S{\uparrow_{\mathcal{E}'}}(A, m) \cup S_{\mathcal{E}'}(C, m)$. The relation $S{\uparrow_{\mathcal{E}'}}(C, m) \rightharpoonup R{\uparrow_{\mathcal{E}'}}(A, m)$ thereby holds by the assumption $S{\uparrow_{\mathcal{E}'}}(A, m) \rightharpoonup R{\uparrow_{\mathcal{E}'}}(A, m)$. If $m$ is defined in $C$ (i.e., $m \in L_{\mathcal{E}'}(C).mtds$), Rule (NewMtd) will lead to an operation $verify(m, R{\uparrow_{\mathcal{E}}}(A, m))$ which is analyzed in the context of class $C$. For each $(r, s) \in R{\uparrow_{\mathcal{E}}}(A, m)$, either Rule (ReqDer) or Rule (ReqNotDer) applies, ensuring $S{\uparrow_{\mathcal{E}'}}(C, m) \rightharpoonup (r, s)$. The relation $S{\uparrow_{\mathcal{E}'}}(C, m) \rightharpoonup R{\uparrow_{\mathcal{E}'}}(A, m)$ is thereby ensured by the analysis of $C$.

Condition 3 of Definition 6 concerns the interface $I$ implemented by $C$, i.e., $L_{\mathcal{E}'}(C).impl = I$. Given the initial judgment (3), application of Rule

(NewClass′) gives the judgment

$$\mathcal{E} \oplus extL(C, A, I, \overline{f}, \overline{M}) \vdash \langle C : \ anSpec(\overline{MS}) \cdot anReq(\overline{M}) \cdot intSpec(\overline{m})\rangle$$

where $\overline{m} = public_{\mathcal{E}'}(I)$. Since the analysis of $C$ succeeds, the analysis of each of these operations must succeed. Let $\mathcal{E}''$ be the environment after analysis of the first two operations, i.e., the judgment $\mathcal{E}'' \vdash \langle C : \ intSpec(\overline{m})\rangle$ is reached. The *intSpec* operation is analyzed by rules (IntSpec) and (DecompInt). None of these rules extend the environment, and for the successful analysis of $intSpec(\overline{m})$ we therefore have $\mathcal{E}'' = \mathcal{E}'$. For each $m \in \overline{m}$, rule (IntSpec) ensures $S{\uparrow}_{\mathcal{E}'}(C, m) \Rightarrow spec_{\mathcal{E}'}(I, m)$ as required by Condition 3 in Definition 6.

Condition 4 in Definition 6 follows from the soundness of $\mathcal{E}$, the premises of Rule (NewClass′), and the property that the analysis of $C$ only extends $S(C, m)$ and $R(C, m)$ for different methods $m$ where $C \in \mathcal{E}'$. □

We conclude this section with an example, extending Example 12 with interfaces.

**Example 14.** Consider the following interface declaration:

```
interface IA {
  int n(int y) : (y ≥ 0, return ≥ 2 * y)
  int m(int x) : (x ≥ 0, return ≥ 2 * x)
}
```

For this interface, we have $spec(IA, n) = (y \geq 0, \mathbf{return} \geq 2 * y)$ for the specification of $n$ and $spec(IA, m) = (x \geq 0, \mathbf{return} \geq 2 * x)$ for that of $m$.

Let the class $A$ be as in Example 12, except that $A$ is now defined to implement the interface $IA$:

```
class A implements IA {
  int n(int y) : (true, return = 5 * y) {return := 5*y}
  int m(int x) : (x ≥ 0, return ≥ 2 * x) {return := n(x)}
}
```

The internal analysis of $A$ is as above, i.e., the $S$ and $R$ mappings are as in Example 12, but we now need to ensure that $A$ implements $IA$. By Rule (IntSpec), this means that the following two relations must hold:

$$S(A, n) \Rightarrow spec(IA, n) \ \text{ and } \ S(A, m) \Rightarrow spec(IA, m)$$

These hold given the specifications in Example 12. Consider next the client code, where field $a$ is now typed by interface $IA$ (we omit the declaration of

the interface $J$ implemented by *Client* as it plays no role in establishing the specifications of the class):

```
class Client implements J {
  IA a := new A;
  int d1() : (true, return ≥ 10) {return := a.n(5)}
  int d2() : (true, return ≥ 10) {return := a.m(5)}
}
```

The verification of *d1* leads to the requirement $(y = 5, \texttt{return} \geq 10)$ towards the call to $n$. This requirement is now verified against the interface specification $spec(IA, n)$, and follows by entailment:

$$(y \geq 0, \texttt{return} \geq 2 * y) \rightharpoonup (y = 5, \texttt{return} \geq 10)$$

In contrast to the situation in Example 12, the verification of method *d2* now also succeeds. The call to $m$ leads to the requirement $(x = 5, \texttt{return} \geq 10)$ which follows from $spec(IA, m)$ by entailment:

$$(x \geq 0, \texttt{return} \geq 2 * x) \rightharpoonup (x = 5, \texttt{return} \geq 10)$$

## 6. Example

In this section we illustrate our approach by a small bank account system implemented by a class *PosAccount* and its subclass *FeeAccount*. The example illustrates how interface encapsulation and the separation of class inheritance and subtyping facilitate code reuse. Class *FeeAccount* reuses the implementation of *PosAccount*, but the type of *PosAccount* is not supported by *FeeAccount*. *Thus, FeeAccount does not represent a behavioral subtype of PosAccount.*

A system of communicating components can be specified in terms of the observable interaction between the different components [50, 10, 27, 14]. In an object-oriented setting with interface encapsulation, the observable interaction of an object may be described by the *communication history*, which is a sequence of invocation and completion messages of the methods declared by the interface (ignoring outgoing calls). At any point in time, the communication history abstractly captures the system state. Previous work [20] illustrates how the observable interaction and the internal implementation of an object can be connected. Expressing pre- and postconditions to methods declared by an interface in terms of the communication history allows

abstract specifications of objects supporting the interface. For this purpose, we assume an auxiliary variable $h$ of type $\mathit{Seq}[\mathit{Msg}]$, where $\mathit{Msg}$ ranges over invocation and completion (return) messages to the methods declared by the interface. However, for the example below it suffices to consider only completion messages, so a history $h$ will be constructed as a sequence of completion messages by the empty ($\epsilon$) and right append ($\cdot$) constructor. In [20], the communication messages are sent between two named objects, the caller and the callee. However, for the purposes of this example, it suffices to record only the name of the completed method and its parameters, where **this** is implicitly taken as the callee. Furthermore, the considered specifications are independent of the actual callers. We may therefore represent completion messages by $\langle m(\overline{e}, r)\rangle$, where $m$ is a method name, $\overline{e}$ are the actual parameter values for this method call, and $r$ is the returned value. For reasoning purposes, such a completion message is implicitly appended to the history as a side effect of each method termination, and the postcondition of the method must hold after the history extension. As the history accumulates information about method executions, it allows abstract specification of objects in terms of previously executed method calls.

### 6.1. Class PosAccount

Let an interface *IPosAccount* support three methods *deposit*, *withdraw*, and *getBalance*. The *deposit* method deposits an amount on the bank account as specified by the parameter value and returns the current balance after the deposit. The *getBalance* method returns the current balance. The *withdraw* method returns *true* if the withdrawal succeeded, and *false* otherwise. A withdrawal succeeds only if it leads to a non-negative balance. The current balance of the account is abstractly captured by the function $\mathit{Val}(h)$ defined by induction over the local communication history as follows:

$$\begin{array}{ll}
\mathit{Val}(\epsilon) & \triangleq 0 \\
\mathit{Val}(h \cdot \langle \mathit{deposit}(x, r)\rangle) & \triangleq \mathit{Val}(h) + x \\
\mathit{Val}(h \cdot \langle \mathit{withdraw}(x, r)\rangle) & \triangleq \textbf{if } r \textbf{ then } \mathit{Val}(h) - x \textbf{ else } \mathit{Val}(h) \textbf{ fi} \\
\mathit{Val}(h \cdot \textbf{others}) & \triangleq \mathit{Val}(h)
\end{array}$$

In this definition, **others** matches all completion messages that are not captured by any of the above cases. In the interface, the three methods are required to maintain $\mathit{Val}(h) \geq 0$.

```
interface IPosAccount {
```

```
  int deposit(nat x): (Val(h) ≥ 0, return = Val(h) ∧ return ≥ 0)
  bool withdraw(nat x):
                  (Val(h) ≥ 0 ∧ h = h0, return = (Val(h0) ≥ x) ∧ Val(h) ≥ 0)
  int getBalance(): (Val(h) ≥ 0, return = Val(h) ∧ return ≥ 0)
}
```

As before, $h_0, b_0, \ldots$ denote logical variables. The interface *IPosAccount* is implemented by a class *PosAccount*, given below. To make method specifications more compact, we have used the notation **inv** $I$ as an abbreviation for the pre/post specification $(I, I)$ for each public method in the class. In this sense, $I$ becomes a *class invariant*. The analysis of **inv** $I$ is captured by our reasoning system since the systems allows a method to have more than one specification. In *PosAccount*, the balance is maintained by a variable *bal*, and the invariant expresses that the balance equals $Val(h)$ and remains non-negative. The expression $bal = Val(h)$ relates the internal state of *PosAccount* objects and the abstract value $Val(h)$, and is used in order to ensure the postconditions declared in the interface.

```
class PosAccount implements IPosAccount {
  int bal = 0;
  int deposit(nat x): (true, return = bal) {
    update(x); return := bal
  }
  bool withdraw(nat x): (bal = b0, return = (b0 ≥ x)) {
    if (bal >= x) then update(-x); return := true
                  else return := false fi
  }
  int getBalance(): (true, return = bal) {return := bal}
  void update(int v): (bal = b0 ∧ h = h0, bal = b0 + v ∧ h = h0) {
    bal := bal + v
  }
  inv bal = Val(h) ∧ bal ≥ 0
}
```

Notice that the *update* method is hidden by the interface. This means that the method is only available for internal invocation; i.e. by method calls on the form $z := update(e)$. The following simple definition of *withdraw* maintains the invariant of the class as it preserves $bal = Val(h)$ and does not modify the balance:

```
bool withdraw(int x) {return := false}
```

However, this implementation is not suitable as it fails to meet the pre/post specification of *withdraw*, which requires that the method must return *true* if the withdrawal can be performed without resulting in a non-negative balance. Next we consider the verification of the *PosAccount* class.

*Pre- and postconditions.* The pre- and postconditions given in the class lead to the inclusion of the following specifications in the $S$ mapping:

$$(true,\ \mathbf{return} = bal) \in S(PosAccount, deposit) \quad (4)$$
$$(bal = b_0,\ \mathbf{return} = (b_0 \geq x)) \in S(PosAccount, withdraw) \quad (5)$$
$$(true,\ \mathbf{return} = bal) \in S(PosAccount, getBalance) \quad (6)$$
$$(bal = b_0 \wedge h = h_0,\ bal = b_0 + v \wedge h = h_0) \in S(PosAccount, update) \quad (7)$$

These specifications are easily verified for the bodies of their respective methods. For *deposit* and *withdraw*, these specifications do not lead to any requirements on *update*. The method *update* is verified by the following proof outline:

$$\{bal = b_0 \wedge h = h_0\}\ bal := bal + v\ \{bal = b_0 + v \wedge h = h_0\}$$

Since the method is not public, a completion message is not appended to the history by method termination. Furthermore, since there are no calls to public methods in the body of *update*, the relation $h = h_0$ is preserved by the method.

*Invariant analysis.* The class invariant is analyzed as a pre/post specification for each public method, i.e., for the methods *deposit*, *withdraw*, and *getBalance*. As a result, the $S$ mapping is extended such that

$$(bal = Val(h) \wedge bal \geq 0,\ bal = Val(h) \wedge bal \geq 0) \in S(PosAccount, m), \quad (8)$$

for $m \in \{deposit, withdraw, getBalance\}$. The two methods *deposit* and *withdraw* make internal calls to *update*, which result in the following two requirements:

$$\begin{array}{l} R(PosAccount, update) = \\ \quad \{\ (bal = Val(h) \wedge bal \geq 0 \wedge x \geq 0 \wedge v = x, \\ \qquad bal = Val(h) + x \wedge bal \geq 0), \\ \quad (bal = Val(h) \wedge bal \geq 0 \wedge bal \geq x \wedge x \geq 0 \wedge v = -x, \\ \qquad bal = Val(h) - x \wedge bal \geq 0)\ \} \end{array} \quad (9)$$

These requirements follow by entailment from Specification (7).

*Interface specifications.* The implementation of each method exported by interface *IPosAccount* must satisfy the corresponding interface specification, according to rule (IntSpec). For *getBalance*, it can be proved that the method specification, as given by Specifications (6) and (8), entails the interface specification

$$(\mathit{Val}(h) \geq 0,\ \mathbf{return} = \mathit{Val}(h) \wedge \mathbf{return} \geq 0).$$

The verification of the other two methods follows the same outline, which concludes the verification of class *PosAccount*.

### *6.2. Class FeeAccount*

The interface *IFeeAccount* resembles *IPosAccount*, as the same methods are supported. However, *IFeeAccount* takes an additional *fee* for each successful withdrawal, and the balance is no longer guaranteed to be non-negative. For simplicity we take *fee* as a (read-only) parameter of the interface and of the class (which means that it can be used directly in the definition of *Fval* below). As before, the assertion pairs of the methods are expressed in terms of functions on the local history. Define the allowed overdrafts predicate $AO(h)$ by means of a function $\mathit{Fval}(h)$ over local histories $h$ as follows:

$$\begin{array}{ll}
AO(h) & \triangleq \mathit{Fval}(h) \geq -\mathit{fee} \\
\mathit{Fval}(\epsilon) & \triangleq 0 \\
\mathit{Fval}(h \cdot \langle deposit(x,r)\rangle) & \triangleq \mathit{Fval}(h) + x \\
\mathit{Fval}(h \cdot \langle withdraw(x,r)\rangle) & \triangleq \mathbf{if}\ r\ \mathbf{then}\ \mathit{Fval}(h) - x - \mathit{fee}\ \mathbf{else}\ \mathit{Fval}(h)\ \mathbf{fi} \\
\mathit{Fval}(h \cdot \mathbf{others}) & \triangleq \mathit{Fval}(h)
\end{array}$$

The interface *IFeeAccount* is declared by

```
interface IFeeAccount(nat fee) {
 int deposit(nat x): (AO(h), return = Fval(h) ∧ AO(h))
 bool withdraw(nat x):
          (AO(h) ∧ h = h0, return = (Fval(h0) ≥ x) ∧ AO(h))
 int getBalance(): (AO(h), return = Fval(h) ∧ AO(h))
}
```

Note that *IFeeAccount* is not a behavioral subtype of *IPosAccount*: a class that implements *IFeeAccount* will not implement *IPosAccount*. Informally, this can be seen from the postcondition of *withdraw*. For both interfaces, *withdraw* returns true if the parameter value is less or equal to the current

balance, but *IFeeAccount* charges an additional fee in this case, as reflected by the *withdraw* case of the *Fval* definition. As an example, consider the following sequence of method calls executed on a newly created object $o$:

$$\texttt{o.deposit}(5); \texttt{o.withdraw}(4); \texttt{o.withdraw}(1)$$

When executed on an instance of *IPosAccount*, the last withdrawal will return *true*: After `o.deposit(5)` we have $Val(h) = 5$, and after `o.withdraw(4)` we have $Val(h) = 1$. Since $Val(h) \geq 1$ when `o.withdraw(1)` is called, the invocation will return *true* and we have $Val(h) = 0$ after the three calls. However, if the calls are executed on an instance of *IFeeAccount*, the last withdrawal may return $false$. Assume that $fee = 2$. After `o.deposit(5)`, we have $Fval(h) = 5$, but after the first withdrawal we have $Fval(h) = -1$. Since $\neg(Fval(h) \geq 1)$ when `o.withdraw(1)` is called, the last invocation of *withdraw* will return $false$. Thus, this example illustrates that an instance of *IFeeAccount* is not a behavioral subtype of an *IPosAccount* instance. Especially, an instance of *IFeeAccount* cannot be used whenever an *IPosAccount* instance is expected.

Given that the implementation provided by the *PosAccount* class is available, it might be desirable to reuse the code from this class when implementing *IFeeAccount*. In fact, only the *withdraw* method needs reimplementation. The class *FeeAccount* below implements *IFeeAccount* and extends the implementation of *PosAccount*.

```
class FeeAccount(int fee)
        extends PosAccount implements IFeeAccount {
  bool withdraw(nat x): (bal = b0, return = (b0 ≥ x)) {
    if (bal >= x) then update(-(x+fee)); return := true
                  else return := false fi
  }
  inv bal = Fval(h) ∧ bal ≥ −fee
}
```

Note that the interface supported by the superclass is not supported by the subclass. Typing restrictions prohibit that methods on an instance of *FeeAccount* are called through the superclass interface *IPosAccount*.

*Pre- and postconditions.* As the methods *deposit* and *getBalance* are inherited without redefinition, the specifications of these methods still hold in the context of the subclass. Especially, Specifications (4), (6), and (7) above

remain valid. For *withdraw*, the declared specification can be proved:

$$(bal = b_0,\ \mathbf{return} = (b_0 \geq x)) \in S(FeeAccount, withdraw) \tag{10}$$

*Invariant analysis.* Again, we take **inv** $I$ as an abbreviation of a pre/post specification $(I, I)$ of each public method in the class. The subclass invariant can be proved for the inherited methods *deposit* and *getBalance* as well as for the new definition of the *withdraw* method. From the proof outline for *deposit*, the following requirement on *update* is included in the requirement mapping:

$$\begin{aligned}(bal = Fval(h) \wedge bal \geq -fee \wedge x \geq 0 \wedge v = x,&\\ bal = Fval(h) + v \wedge bal \geq -fee) \in R(FeeAccount, update)&\end{aligned}$$

This requirement follows from Specification (7) of *update*. The analysis of *withdraw* gives the following requirement on *update*, which also follows from Specification (7):

$$\begin{aligned}(bal = Fval(h) \wedge bal \geq -fee \wedge x \geq 0 \wedge bal \geq x \wedge v = -(x + fee),&\\ bal = Fval(h) - x - fee \wedge bal \geq -fee) \in R(FeeAccount, update)&\end{aligned}$$

The invariant analysis leads to the inclusion of the invariant as a pre/post specification in the sets $S(FeeAccount, deposit)$, $S(FeeAccount, withdraw)$, and $S(FeeAccount, getBalance)$, similar to Specification (8).

*Interface specification.* Now reconsider the method *getBalance*. After the above analysis, the specification set for this method is given by:

$$\begin{aligned}&S{\uparrow}(FeeAccount, getBalance) =\\ &\quad S(FeeAccount, getBalance) \cup S(PosAccount, getBalance) =\\ &\quad\ \{(bal = Fval(h) \wedge bal \geq -fee,\ bal = Fval(h) \wedge bal \geq -fee)\} \cup\\ &\quad\ \{(bal = Val(h) \wedge bal \geq 0,\ bal = Val(h) \wedge bal \geq 0),\\ &\quad\ \ (true,\ \mathbf{return} = bal)\}\end{aligned} \tag{11}$$

The interface specification of *getBalance* given by *IFeeAccount* is:

$$(AO(h),\ \mathbf{return} = Fval(h) \wedge AO(h)) \tag{12}$$

Interface specification (12) follows by entailment from Specification (11), using (IntSpec). Note that the superclass invariant is not established by the precondition of Specification (12), which means that the superclass invariant cannot be assumed when establishing the postcondition of Specification (12). However, the other superclass specification is needed, expressing that **return** equals *bal*. The verification of the interface specifications for *deposit* and *withdraw* follows the same outline.

## 7. Related and Future Work

Object-orientation poses several challenges to program logics; e.g., inheritance, late binding, recursive and re-entrant 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 into sequences [16]. Numerous proof methods, verification condition generators, and validation environments for object-oriented languages have been developed, including [1, 3, 23, 43, 42, 28, 30, 9]. Java in particular 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.

Class inheritance is a central feature of object orientation which allows subclasses to be designed by reusing and redefining the code of superclasses with a flexibility which goes beyond behavioral subtyping [49]. However, proof systems usually restrict code reuse to behavioral subtyping. For example, a recent survey of challenges and results for the verification of sequential object-oriented programs [35] relies on behavioral subtyping when reasoning about late binding and inheritance. In contrast, proof systems studying late bound methods without relying on behavioral subtyping have been shown to be sound and complete by Pierik and de Boer [47], assuming a closed world. See also [46] for a discussion of (relative) completeness in connection with behavioral subtyping. While proof-theoretically satisfactory, the closed world assumption is unrealistic in practice and necessitates costly reverification when the class hierarchy is extended (as discussed in Section 1). Our work on lazy behavioral subtyping is situated between these two approaches.

In order to better support object-oriented design, proof systems should be constructed for incremental (or modular [19]) reasoning. Most prominent in that context are different variations of *behavioral subtyping* [39, 49, 36]. The underlying idea is quite simple: subtyping in general is intended to capture "specialization" and in object-oriented languages, this may be interpreted such that instances of a subclass can be used where instances of a superclass are expected. To generalize this subsumption property from types (such as method signatures) to behavioral properties is the step from standard to behavioral subtyping. The notion of behavioral subtyping dates back to America [4] and Liskov and Wing [38, 39], and is also sometimes referred to as Liskov's substitutability principle. The general idea has been explored from

various angles. For instance, behavioral subtyping has been characterized model-theoretically [37, 18] and proof-theoretically [5, 39].

Specification inheritance is used to enforce behavioral subtyping in [19], where subtypes inherit specifications from their supertypes (see also [52] which describes specification inheritance for the language Fresco). Virtual methods [48] 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. Such specifications of 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 flexibility in subclass design than the approach as such suggests. In particular, the verification platforms for *Spec*$^\sharp$ [8] and JML [11] rely on versions of behavioral subtyping. Wehrheim [51] investigates behavioral subtyping not in a sequential setting but for active objects. Dynamic binding in a general sense, namely that the code executed is not statically known, does not only arise in object-oriented programs. Ideas from behavioral subtyping have been used to support modular reasoning in the context of aspect-oriented programs [13, 34].

The fragile base class problem emerges when seemingly harmless superclass updates lead to unexpected behavior of subclass instances [41]. 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 provides an approach to dealing with 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.

Recently incremental reasoning, both for single and multiple inheritance, has been considered in the context of *separation logic* [12, 45, 40]. These approaches distinguish "static" specifications, given for each method implementation, from "dynamic" specifications used to verify late-bound calls. The dynamic specifications are given at the declaration site, in contrast to our work on lazy behavioral subtyping in which late-bound calls are verified based on call-site requirements. As in lazy behavioral subtyping, the goal is "modularity"; i.e., the goal is to avoid reverification when incrementally developing a program. Complementing the results presented in this paper, we have shown

how lazy behavioral subtyping can be used in the setting of multiple inheritance in [22], in which strategies for method binding in multiple inheritance class hierarchies are related to lazy behavioral subtyping.

We currently integrate lazy behavioral subtyping in a program logic for Creol [32, 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*. Creol's type system is purely based on interfaces. Interface types provide a clear distinction between internal and external calls. As shown in this paper, the separation of interface level subtyping from class level inheritance allows class inheritance to exploit code reuse quite freely based on lazy behavioral subtyping, while still supporting incremental reasoning techniques. Classes in Creol may implement several interfaces, slightly extending the approach presented in this paper. It is also possible to let interfaces influence the reasoning for internal calls in a more fine-grained manner, with the aim of obtaining even weaker requirements to redefinitions. We are currently investigating the combination of lazy behavioral subtyping with class upgrades. This combination allows class hierarchies to not only evolve by subclass extensions, but also by restructuring the previously analyzed class hierarchy in ways which control the need for reverification.

## 8. Conclusion

This paper presents lazy behavioral subtyping, a novel strategy for reasoning about late bound internal method calls. The strategy is designed to support incremental reasoning and avoid reverification of method specifications in an open setting, where class hierarchies can be extended by inheritance. To focus the presentation, we have abstracted from many features of object-oriented languages and presented lazy behavioral subtyping for an object-oriented kernel language based on single inheritance. This reflects the mainstream object-oriented languages today, such as Java and C$^\sharp$.

Behavioral subtyping has the advantage of providing incremental and modular reasoning for open object-oriented systems, but severely restricts code reuse compared to programming practice due to behavioral constraints on method overriding. Lazy behavioral subtyping also provides incremental reasoning, but supports significantly more flexible reuse of code. In addition lazy behavioral subtyping offers modularity when combined with interfaces,

separating the interface and class hierarchies to support both subtyping and flexible code reuse. This paper presents both systems with soundness proofs. An example of code reuse in the banking domain demonstrates how incremental reasoning is achieved by lazy behavioral subtyping in a setting where behavioral subtyping does not apply. Lazy behavioral subtyping appears as a promising framework for controlling a range of desirable changes in the development of object-oriented class hierarchies.


## Acknowledgment

The authors gratefully recognize the very thorough efforts of the anonymous reviewers of this paper.


## 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 *Lecture Notes in Computer Science*, pages 11–41. Springer-Verlag, 2003.

[2] ACM. *37th Annual Symposium on Principles of Programming Languages (POPL)*, Jan. 2008.

[3] S. Alagic and S. Kouznetsova. Behavioral compatibility of self-typed theories. In *16th European Conference on Object-Oriented Programming (ECOOP'02)*, volume 2374 of *Lecture Notes in Computer Science*, pages 585–608. Springer-Verlag, 2002.

[4] P. America. A behavioural approach to subtyping in object-oriented programming languages. In *Inheritance hierarchies in knowledge representation and programming languages*, pages 173–190. John Wiley and Sons Ltd., 1991.

[5] 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 (REX Workshop)*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, 1991.

[6] 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.

[7] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Systems*. Texts and Monographs in Computer Science. Springer-Verlag, 1991.

[8] 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 *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.

[9] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2007.

[10] M. Broy and K. Stølen. *Specification and Development of Interactive Systems*. Monographs in Computer Science. Springer-Verlag, 2001.

[11] 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, *Proc. FMICS '03*, volume 80 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2003.

[12] W.-N. Chin, C. David, H.-H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In POPL'08 [2], pages 87–99.

[13] C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *SPLAT 2003: Software engineering Properties of Languages for Aspect Technologies at AOSD 2003*, Mar. 2003. Available as Computer Science Tech. Rep. TR03-01a from ftp://ftp.cs.iastate.edu/pub/techreports/TR03-01/TR.pdf.

[14] O.-J. Dahl. *Verifiable Programming*. International Series in Computer Science. Prentice Hall, New York, N.Y., 1992.

[15] 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.

[16] F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Proc. Foundations of Software Science and Computation Structure, (FOSSACS'99)*, volume 1578 of *Lecture Notes in Computer Science*, pages 135–149. Springer-Verlag, 1999.

[17] 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 *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, Mar. 2007.

[18] K. K. Dhara and G. T. Leavens. Weak behavioral subtyping for types with mutable objects. In S. D. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1995.

[19] K. K. Dhara and G. T. Leavens. Forcing behavioural subtyping through specification inheritance. In *Proc. 18th International Conference on Software Engineering*, pages 258–267. IEEE Computer Society Press, 1996.

[20] J. Dovland, E. B. Johnsen, and O. Owe. Observable Behavior of Dynamic Systems: Component Reasoning for Concurrent Objects. *Electronic Notes in Theoretical Computer Science*, 203(3):19–34, 2008.

[21] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. In J. Cuellar and T. Maibaum, editors, *Proc. 15th International Symposium on Formal Methods (FM'08)*, volume 5014 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, May 2008.

[22] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Incremental reasoning for multiple inheritance. In M. Leuschel and H. Wehrheim, editors, *Proc. 7th International Conference on Integrated Formal Methods (iFM'09)*, volume 5423 of *Lecture Notes in Computer Science*, pages 215–230. Springer-Verlag, Feb. 2009.

[23] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*. ACM, 2001. In *SIGPLAN Notices* 36(11).

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[25] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[26] 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-Verlag, 1971.

[27] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.

[28] M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.

[29] 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.

[30] 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 *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag, 2001.

[31] E. B. Johnsen and O. Owe. Inheritance in the presence of asynchronous method calls. In *Proc. 38th Hawaii International Conference on System Sciences (HICSS'05)*. IEEE Computer Society Press, Jan. 2005.

[32] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[33] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.

[34] R. Khatchadourian, J. Dovland, and N. Soundarajan. Enforcing Behavioral Constraints in Evolving Aspect-Oriented Programs. In *Proc. International Workshop on Foundations of Aspect-Oriented Languages (FOAL'08)*, pages 19–28. ACM Press, 2008.

[35] 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.

[36] 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.

[37] G. T. Leavens and W. E. Wheil. Reasoning about object-oriented programs that use subtypes. In *Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'90)*, pages 212–223. ACM, 1990. In *SIGPLAN Notices* 25(10).

[38] B. Liskov. Data abstraction & hierarchy. In *Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'87)*. ACM, 1987. In *SIGPLAN Notices* 22(12).

[39] 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.

[40] C. Luo and S. Qin. Separation logic for multiple inheritance. *Electronic Notes in Theoretical Computer Science*, 212:27–40, 2008.

[41] 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'98)*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382. Springer-Verlag, 1998.

[42] D. v. Oheimb. *Analysing Java in Isabelle/HOL: Formalization, Type Safety, and Hoare-Logics*. PhD thesis, Technische Universität München, 2001.

[43] 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 *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 2002.

[44] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

[45] M. J. Parkinson and G. M. Biermann. Separation logic, abstraction, and inheritance. In POPL'08 [2].

[46] C. Pierik and F. S. de Boer. On behavioral subtyping and completeness. In *Proc. ECOOP 2005 workshop on Formal Techniques for Java-like Programs*, 2005.

[47] C. Pierik and F. S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Computer Science*, 343(3):413–442, 2005.

[48] 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 *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

[49] 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.

[50] N. Soundararajan. Axiomatic semantics of communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 6(4):647–662, 1984.

[51] H. Wehrheim. Behavioral subtyping relations for active objects. *Formal Methods in System Design*, 23(2):143–170, 2003.

[52] A. Wills. Specification in Fresco. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, chapter 11, pages 127–135. Springer-Verlag, 1992.