Incremental Reasoning with Lazy Behavioral Subtyping for Multiple Inheritance☆

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

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

Abstract

Object-orientation supports code reuse and incremental programming. Multiple inheritance increases the possibilities for code reuse, but complicates the binding of method calls and thereby program analysis. Behavioral subtyping allows program analysis under an *open world assumption*; i.e., under the assumption that class hierarchies are extensible. However, method redefinition is severely restricted by behavioral subtyping, and multiple inheritance may lead to conflicting restrictions from independently designed superclasses. This paper presents a more liberal approach to incremental reasoning for multiple inheritance under an open world assumption. The approach, based on *lazy behavioral subtyping*, is well-suited for multiple inheritance, as it incrementally imposes context-dependent behavioral constraints on new subclasses. We first present the approach for a simple language and show how incremental reasoning can be combined with flexible code reuse. Then this language is extended with a hierarchy of interface types which is independent of the class hierarchy. In this setting, flexible code reuse can be combined with modular reasoning about external calls in the sense that each class is analyzed only once. We formalize the approach as a calculus and show soundness for both languages.

☆This work was partly supported by 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)

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

1. Introduction

Object-orientation supports code reuse and incremental programming through inheritance. Class hierarchies are extended over time as subclasses are developed and added. A class may reuse code from its superclasses but it may also specialize and adapt this code by providing additional method definitions, possibly overriding definitions in superclasses. This way, the class hierarchy allows programs to be represented in a compact and succinct way, significantly reducing the need for code duplication. *Late binding* is the underlying mechanism for this incremental programming style; the binding of a method call at run-time depends on the actual class of the called object. Consequently, the code to be executed depends on information which is not statically available. Although late binding is an important feature of object-oriented programming, this loss of control severely complicates reasoning about object-oriented programs.

Behavioral subtyping is the most prominent solution to regain static control of late bound method calls (see, e.g., [28, 2, 26]) with an *open world assumption*; i.e., where class hierarchies are extensible. This approach achieves incremental reasoning in the sense that a subclass may be analyzed in the context of previously defined classes, such that previously proved properties are ensured by additional verification conditions. However, the approach restricts how methods may be redefined in subclasses. To avoid reverification, any method redefinition must *preserve* certain properties of the method which is redefined. In particular, this applies to the method's contract; i.e., the pre- and postcondition for its body. This contract can be seen as a description of the promised behavior of all implementations of the method. Unfortunately, this restriction limits code reuse and is often violated in practice [38]; for example, it is not respected by the standard Java library definitions.

Multiple inheritance offers greater flexibility than single inheritance, as several class hierarchies can be combined in a subclass. However, it also complicates language design and is often explained in terms of complex run-time data structures such as virtual pointer tables [39], which are hard to understand. Formal treatments are scarce (e.g., [37, 10, 6, 19, 41]), but help clarify

intricacies, thus facilitating design and reasoning for programs using multiple inheritance. Multiple inheritance also complicates behavioral reasoning, as name conflicts may occur between methods which were independently defined in different branches of the class hierarchy.

Work on behavioral reasoning about object-oriented programs has mostly focused on languages with single inheritance (see, e.g., [35, 36, 9]). It is an open problem how to design an incremental proof system for multiple inheritance under an open world assumption, without severely restricting code reuse. In this paper we propose a solution to this problem. The approach extends *lazy behavioral subtyping*, which was developed for single inheritance systems [16, 18] to allow more flexible code reuse than reasoning systems based on behavioral subtyping. Our approach applies to a wide class of object-oriented systems, relying on the assumption of a *healthy* binding strategy, which is needed for incremental reasoning. Healthiness may easily be imposed on non-healthy binding strategies. The approach is formalized as a syntax-driven inference system, for which we show soundness. The inference system combines deductive style program logic with incremental program development, and is well-suited for program development environments [17].

Although this system ensures that old proofs are never violated, an external call $x.m(\overline{e})$, where x is an object variable, may result in additional proof obligations for the declared class of x, and that class may already have been verified. As a consequence, it may be necessary to revisit previously verified classes at a later stage in the program analysis. To improve this situation, we extend [17] by considering a refined version of the calculus which introduces behavioral interfaces to encapsulate objects. The behavioral constraints of the interface implemented by a class become proof obligations for that class. As a result, the refined calculus is both *incremental* and *modular*: it is no longer necessary to revisit a class due to requirements on calls which occur later during the analysis of unrelated classes. In the refined system, subtyping applies to the inheritance relationship on interfaces, whereas code may be reused more freely in the class hierarchy.

Paper overview. Section 2 discusses late binding and multiple inheritance. Section 3 introduces proof environments for behavioral reasoning, and Section 4 presents the inference system for incremental reasoning. Section 5 extends the inference system with interface encapsulation. Section 6 discusses related work and Section 7 concludes the paper.

$$
\begin{array}{lll}
P & ::= & \overline{L}\ \{t\} \\
L & ::= & \texttt{class}\ C\ \texttt{extends}\ \overline{C}\ \{\overline{f}\ \overline{M}\ \overline{MS}\} \\
M & ::= & m(\overline{x})\{tr\} \\
tr & ::= & \texttt{var}\ \overline{z};\ t;\ \texttt{return}\ e \\
MS & ::= & m@C(\overline{x}) : (p, q) \\
t & ::= & v := rhs \mid \texttt{skip} \mid \texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ t\ \texttt{fi} \mid t; t \\
v & ::= & f \mid f@C \\
rhs & ::= & e \mid \texttt{new}\ C \mid e.m(\overline{e}) \mid m(\overline{e}) \mid m@C(\overline{e}) \\
e & ::= & x \mid v \mid \texttt{this} \mid \texttt{true} \mid \texttt{false} \mid \mathbb{N} \mid op(\overline{e})
\end{array}
$$

Figure 1: Syntax for the language *MI*, with class names C and method names m. Expressions e include formal parameters x, fields f, **return**, **this**, Boolean expressions b, object creation, method calls, and operations *ops* on abstract data types. Let *nil* denote the empty list and whitespace list concatenation (e.g., $\overline{C}$ is a list and $C\ \overline{C}$ a non-empty list of class names), and let p and q denote program assertions.

2. Late Binding and Multiple Inheritance

2.1. Syntax for an Object-Oriented Kernel Language MI

To succinctly explain late binding and our analysis strategy, we consider an object-oriented kernel language called *MI* with a standard operational semantics, e.g., similar to that of Featherweight Java [22]. The syntax of *MI* is given in Figure 1. Vector notation denotes lists; e.g., a list of expressions is written $\overline{e}$. A program P consists of a list $\overline{L}$ of class definitions, followed by a method body. For simplicity, we let expressions e (other than method calls and object creation) be without side-effects and assume that methods with the same name have the same signature (i.e., no method overloading), class names are unique, programs are well-typed, and we ignore the types of fields and methods. Two notable differences to Featherweight Java are multiple inheritance and a corresponding form of static method calls. These are explained below.

For classes C_1 and C_2, we let $C_1 \leq C_2$ denote the reflexive and transitive subclass relation derived from class inheritance. We say that C_1 is *below* C_2 if $C_1 \leq C_2$. Thus, $C_1 \leq C_2$ if C_1 and C_2 are the same class or if C_1 extends a class that is below C_2. Furthermore, C_2 is *above* C_1 if C_1 is below C_2. A

subclass is below a *superclass*. The two classes are *related*, written $C_2 \bowtie C_1$, if one is below the other.

A class C extends a list $\overline{C}$ of superclass names with fields $\overline{f}$, methods $\overline{M}$, and method specifications $\overline{MS}$. The list $\overline{C}$ is assumed to consist of unique class names, and the names of methods $\overline{M}$ and fields $\overline{f}$ are assumed to be unique within the class. We say that C *defines* a method m if $\overline{M}$ contains an implementation of m. Let a partial function $body(C, m)$ return this implementation (so $body(C, m)$ is undefined if m is not in $\overline{M}$). For a method m defined in some superclass B of C, but not defined in any subclass of B above C, we say that C *inherits* m from B if m is not defined in C, otherwise m is *overridden* in C. Since C may extend more than one class, it is possible for C to inherit m from more than one superclass.

A method M takes formal parameters $\overline{x}$ and contains a statement t as its method body where $\overline{x}$ and **this** are read-only. The method may also declare a list $\overline{z}$ of local variables. 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 *MI*, a possible constructor method in the class must be called explicitly.) There are standard statements for **skip**, conditionals **if** b **then** t_1 **else** t_2 **fi**, and assignments $v := e$. We use **if** b **then** t **fi** as an abbreviation for **if** b **then** t **else skip fi**.

We syntactically distinguish *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 the actual parameters $\overline{e}$. The call is bound at run time depending on the actual class of the object. The symbol @ is used for static binding. An internal static call $m@C(\overline{e})$ may occur in a class below C, and the call is bound above C at compile time. This statement generalizes the call to the superclass found in languages with single inheritance; C may here be any class above the current class as long as an implementation of m can be found in a class above C. In an external method call $e.m(\overline{e})$, the object e (which may be self) receives a call to the method m with the actual parameters $\overline{e}$. All external calls are late bound. 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 $e.m(\overline{e})$, $m@C(\overline{e})$, or $m(\overline{e})$ directly as statements for simplicity. Note that the list $\overline{e}$ of actual parameter values may be empty. Similarly to static calls, $f@C$ binds a field f above C.

User given method specifications may occur in class definitions, and are

```
class B {
  nat x:=0, y := 0;
  n() {y := y + x}
  m():(true, y ≥ x) {x := x + 1; n()}
}
class C extends B {
  n(){y:=x}
  m@B():(true, x = y)
}
```

```
class D extends B {
  n(){y := x + 1}
  m@B():(true, y > x)
}
```

Figure 2: Small example of a class hierarchy with subclass specification of inherited methods.

of the form $m@B(\overline{x}) : (p, q)$. We say that a specification is *given in the context of* a class C if the specification occurs syntactically in the definition of C. Here, B may be any class above C, as long as an implementation of m can be found above B. Remark that B and C may be the same class. The *assertion pair* (p, q) defines a pre/post specification for the definition of $m@B(\overline{x})$. As the specification is given in the context of C, it must be satisfied when this method definition is executed on instances of classes below C, but it need not be satisfied when m is executed on instances of B (unless B and C are the same class). Assertions may range over the available fields and method parameters, **this**, **return**, and logical variables. For convenience, we let $m(\overline{x}) : (p, q)\{t\}$, defined in class C, abbreviate the combination of the definition $m(\overline{x})\{t\}$ and the specification $m@C(\overline{x}) : (p, q)$.

2.2. Context dependent specifications

Specifications of a method m may be given in the class where m is defined or in a subclass. If m is not overridden by a subclass, the subclass may still provide a specification of m. This is feasible in the presence of late binding, as some method n called by m may be overridden by the subclass. A subclass specification of m may then account for the behavior of m when taking the overriding version of n into account. This is illustrated by the following example.

Example 1. Consider the classes B and C in Figure 2. Class B defines two methods m and n, where there is a call to n in the body of m. The method n is overridden by the subclass C, and C gives a specification of the inherited method m. When executed on an instance of C, execution of m will lead to a

state where x equals y. Note that the specification is given in the context of subclass C; it is not guaranteed to hold when m is executed on an instance of B. The specification of m given in class B holds when m is executed on an instance of B or C.

Observe that different subclasses may override methods in different manners. As illustrated by the next example, this means that different subclasses may have conflicting specifications of some method m, since internal calls in the body of m may bind differently due to late binding.

Example 2. Consider the class D in Figure 2. As in the above example, the subclass overrides method n and gives a specification of method m. This specification must hold when m is executed on instances of subclass D. There is a conflict between the two specifications of m given in C and D, in the sense that the conjunction of the postconditions equals the unwanted postcondition *false*. However, this apparent conflict does not lead to any reasoning problems as the specifications are given in the context of unrelated classes. The specification of m given in class B also holds when m is executed on an instance of D.

2.3. Name Conflicts and Healthiness

Inheritance relates classes in a class hierarchy. For single inheritance this hierarchy forms a tree, whereas for multiple inheritance, the hierarchy forms a directed, acyclic graph. In the single inheritance tree, *vertical* name conflicts occur when a subclass overrides a method from a superclass. The *binding strategy* for method calls must resolve such conflicts. Late binding or dynamic dispatch selects the method body to be executed at run-time, depending on the callee's run-time class: the selected body is found by the *first matching definition* of the method above the actual class of the object. In class hierarchies with multiple inheritance, there are also *horizontal* name conflicts. These occur when different definitions of the same method are found above a given class, depending on the chosen path through the hierarchy. More elaborate binding strategies are needed to resolve horizontal conflicts. Some binding strategies are infeasible, as they contradict incremental program development. This is illustrated by the following example.

Example 3. We consider a class hierarchy for a bank account system, given in Figure 3. Potential problems with horizontal name conflicts are illustrated by the classes in Figure 4, sketching an implementation of the classes

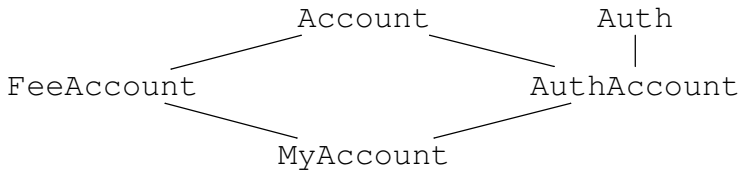

Figure 3: A multiple inheritance class hierarchy for an account system. The inheritance relation is indicated by lines; e.g., the class `FeeAccount` inherits from `Account`.

`Account`, `Auth`, and `AuthAccount`. (Implementations of the remaining classes is considered in Example 5.) Class `Account` implements basic facilities for depositing and withdrawing money. The actual manipulation of the balance is implemented by a method `update`. Assuming that account owners are identified by integers, the owner of the account is the only client allowed to make withdrawals, as checked by the method `validate`. Class `Auth`, developed independently of `Account`, provides functionality for storing two client identities. These two classes are inherited by the subclass `AuthAccount`. By overriding the method `validate` in the subclass, the fields declared by `Auth` are now allowed to hold clients that are allowed to perform withdrawals. For `AuthAccount`, inheritance of `Account` and `Auth` gives a horizontal name conflict for method `update`. The behavior of the two versions of `update` is completely different, which means that the behavior assumed by the `add` method in `Auth` will not hold in the subclass if the internal call to `update` is bound to the implementation in `Account`. Thus, in order to support incremental design, the internal call in `Auth` should bind to the definition in `Auth`, and correspondingly, the internal calls in `Account` should bind to the definition in `Account`.

One solution to resolve horizontal name conflicts is by *explicit resolution* of the names of the superclass' methods, specified as part of the inheritance list; e.g., by qualification or renaming of methods as in C++ [39], Eiffel [30], and POOL [3]. However, it might be undesirable to force the programmer to modify method names, making programs more difficult to understand and maintain. We generalize this approach and decorate each internal call with a *binding clause* restricting the binding space. Such a clause may represent a specific name resolution strategy, or be explicitly provided by the programmer. This way, the approach of this paper is applicable to several resolution strategies. Binding clauses allow us to consider horizontal name conflicts as

```
class Account {
  int bal = 0; int owner;
  setOwner(nat o) {owner := o}
  deposit(nat x): {update(x)}
  withdraw(nat id, nat x):(bal = b0 ∧ id = owner, bal = b0 − x) {
    bool v := validate(id); if v then update(-x) fi}
  update(int y):(bal = b0, bal = b0 + y) {bal:=bal+y}
  int getbal() {return := bal}
  bool validate(nat id):(true, return = (owner = id)) {
    return (owner=id)}
}
class Auth {
  int a1 := -1, a2 := -1;
  add(nat a) {if (a1 != -1) then update(a) else a1 := a fi}
  update(int a) {a2 := a1; a1 := a}
  bool validate(nat a):(true, return = (a = a1 ∨ a = a2)) {
    return (a=a1 || a=a2)}
}
class AuthAccount extends Account Auth {
  bool validate(nat id) :(true, return = (id = owner ∨ id = a1 ∨ id = a2) {
    bool r := validate@Account(id);
    if (!r) then r := validate@Auth(id) fi; return r}
  withdraw@Account(nat id, nat x) :
            (bal = b0 ∧ (id = a1 ∨ id = a2), bal = b0 − x)
}
```

Figure 4: Implementation sketches for classes in Figure 3. Here, variable declarations are typed, and b_0, n_0 denote logical variables. When redefining a method, static binding allows calls to superclass versions (avoiding recursion).

a natural feature of multiple inheritance. In particular when using libraries, the programmer cannot be expected to know (or resolve) potential name conflicts of, e.g., auxiliary methods in the libraries. To support incremental program development and reasoning, we impose the following *healthiness condition* on the binding of a method call:

- an internal call made by a method defined in C must bind to a class related to C, and
- an external call $x.m$, where x has C as declared class, must bind to a class related to C.

It is assumed that vertical name conflicts are resolved as explained above.

We say that a *binding strategy is healthy* if all calls are guaranteed to be healthy. These healthiness restrictions resolve the binding problems illustrated in Example 3. Explicit resolution of horizontal name conflicts may ensure healthiness. It is easy to see that healthiness removes accidental overriding of methods, due to unfortunate binding. In particular, if an empty subclass C extends two unrelated classes A and B, then C does not cause unexpected behavior due to possible horizontal name conflicts in A and B.

Let $C\#m$ denote a call to m where the binding is restricted to classes related to C. In Example 3, if the call to `update` in `Auth` is replaced by `Auth#update`, the call becomes healthy. When executed in an instance of `AuthAccount`, the call will bind in a class related to `Auth`. For the rest of this paper, we use the convention that *an internal call to m made by a method defined in C is understood as $C\#m$.* Similarly, *an external call $x.m$ with C as the declared class of x, is understood as $x.C\#m$.* Note that these translations can be done mechanically and integrated with static type checking. As static calls are inherently healthy, this ensures healthy binding.

We remark that the notation $C\#m$ might also be used by the programmer to distinguish between definitions of m that are inherited from different superclasses. In general, several superclasses may restrict the binding of methods, using the notation $C_1\#C_2\#...\#m$ to restrict m to definitions in classes related to every C_i. However, in order to keep the presentation simple, we will not consider multiple restrictions, and therefore limit the presentation to calls restricted by the conventions above.

2.4. The Binding of Method Calls and Fields

For the reasoning system, we need an explicit definition of a healthy resolution strategy. In this paper, we formalize such a strategy by a function *bind* defined below. Other definitions of *bind* are possible and would lead to variations of the calculus. We say that a call to a method m is bound with respect to a *search class* D; i.e., $bind(D, m)$ denotes the search for a definition of m which starts in D. In this case, the call must bind to a definition of m in a class above D, such that no other definition of m is found by the search below this class. Following [11, 14, 23], ambiguities are solved by fixing the order in which inherited classes are searched, e.g., from left to right. Let *Cid* and *Mid* denote class and method names. To make the representation of class hierarchies compact, a class name is bound to a tuple $\langle \overline{C}, \overline{f}, \overline{M} \rangle$ of type *Class*, where the declared superclasses $\overline{C}$, the fields $\overline{f}$, and the method definitions $\overline{M}$ are accessible by observer functions *inh*, *fields*, and

mtds, respectively. This binding strategy can be defined by a partial function $bind : List[Cid] \times Mid \to Cid$ as follows:

$$\begin{array}{lcll} bind(nil, m) & \triangleq & \bot & \\ bind(D\ \overline{D}, m) & \triangleq & D & \text{if } m \in D.mtds \\ bind(D\ \overline{D}, m) & \triangleq & bind(D.inh\ \overline{D}, m) & \text{otherwise,} \end{array}$$

where $D.inh\ \overline{D}$ reduces to $\overline{D}$ when $D.inh$ is empty. Observe that this strategy is not healthy, since an internal call would be bound independently of where the *call-site* occurs in the class hierarchy, i.e., the class in which the call textually occurs. For internal late bound calls, a healthy strategy can be obtained by restricting the binding to classes related to the call-site. We denote by $bind(D, C\#m)$ the binding of a call $C\#m$ for search class D. The search is restricted by C; the returned class must be either above or below C. This ensures the healthiness condition described above. By type-safety, there is a definition of m above C; thus $bind(D, C\#m)$ is well-defined for D below C. A healthy binding strategy may then be defined by the following function:

Definition 1. Define $bind(\_, \_\#\_) : List[Cid] \times Cid \times Mid \to Cid$ by:

$$\begin{array}{lcll} bind(nil, C\#m) & \triangleq & \bot & \\ bind(D\ \overline{D}, C\#m) & \triangleq & D & \text{if } C \bowtie D \wedge m \in D.mtds \\ bind(D\ \overline{D}, C\#m) & \triangleq & bind(D.inh\ \overline{D}, C\#m) & \text{if } C \bowtie D \wedge m \notin D.mtds \\ bind(D\ \overline{D}, C\#m) & \triangleq & bind(\overline{D}, C\#m) & \text{otherwise} \end{array}$$

For simplicity we here ignore typing of parameters. With type information, one should in addition to the $m \in D.mtds$ check in line two, check that the types of the actual parameters are subtypes of the corresponding formal parameters. See [24] for more details.

For *external calls*, healthiness is ensured by binding the call $x.m$ by $bind(D, C\#m)$ where C is the declared class of x and D the actual class of x. A *statically bound* method call $m@C$ is bound above C by $bind(C, C\#m)$. For a method specification $m@C(\overline{x}) : (p, q)$ given in the context of some class D, we take (p, q) as a specification of the first implementation of m above C as found by $bind(C, C\#m)$.

Similar binding functions may be used to define the binding of fields: An occurrence of $f@B$ is allowed inside a class definition C if B is above C, and is bound above B; and an unqualified occurrence of f inside C is understood as $f@C$.

3. Lazy Behavioral Subtyping

Lazy behavioral subtyping supports incremental reasoning for extensible class hierarchies; each class is analyzed based on the analysis of its superclasses, but independently of (future) subclasses. Lazy behavioral subtyping was presented for single inheritance in [16, 18]. We here present an extension for multiple inheritance and horizontal name conflicts based on the language *MI* and the healthy binding strategy defined in Section 2.4. With healthy binding, an internal late bound method call binds to a class related to the call-site. Behavioral constraints may therefore be propagated down the class hierarchy, which allows incremental reasoning. The proof method has two parts, a conventional program logic (e.g., [35, 20, 5, 33]) and, on top of that, a *proof environment* which incrementally tracks method specifications and requirements.

3.1. Proof Outlines

Apart from the treatment of late bound method calls, our initial reasoning system follows standard proof rules [4, 5] 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 [13]. We present the proof system using Hoare triples $\{p\}\, t\, \{q\}$ [20], for assertions p and q, and statement sequence t. Here, p is the precondition and q is the postcondition to t. The meaning $\models \{p\}\, t\, \{q\}$ of a triple $\{p\}\, t\, \{q\}$ is standard: if t is executed in a state where p holds and the execution terminates, then q holds after t 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* [33] for a triple $\{p\}\, t\, \{q\}$ is the statement sequence decorated with assertions. The main idea is to decorate different points in the program t with assertions such that the analysis between the different program points can be done mechanically, ensuring $\vdash_{PL} \{p\}\, t\, \{q\}$. A classical example is to decorate loops in the program with loop invariants. For the purposes of this paper, we are mainly interested in decoration of method calls with pre- and postconditions.

Let the notation $O \vdash_{PL} t : (p, q)$ mean that O is a proof 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 pre- and postconditions for the internal late bound method calls are called *requirements*. Thus, for a decorated call

$\{r\}\, n()\, \{s\}$ in O, (r, s) is a requirement for n. Examples of proof outlines can be found in Example 5.

3.2. Method Specifications and Requirements

The verification technique distinguishes between a method's declared *specification* (its contract) and its *requirement*. Roughly, the first captures its announced behavior as declared in the specification list $\overline{MS}$ of class definitions. For a method m defined in class B and a user given specification $m@B(\overline{x}) : (p, q)$ given in the context of class C, the assertion pair (p, q) is remembered as a specification of the implementation $body(B, m)$. In contrast, the requirements stem from the analysis of internal late bound method calls and represent properties needed to verify the call-site of a method, namely to satisfy the specification of the call-site. Inside a class hierarchy, a method with a given name may be available in more than one class due to inheritance, and can be called internally from different call-sites. Consequently, the properties related to a method definition are considered for each class and its position in the class hierarchy. If, furthermore, the class hierarchy is incrementally extended, new specifications and requirements may be added. This bookkeeping of properties is done in a proof environment, by means of two mappings S and R. Method specifications and requirements are written as assertion pairs (p, q) of type $APair$.

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 $L_{\mathcal{E}} : Cid \to Class$ is a partial mapping and $S_{\mathcal{E}}, R_{\mathcal{E}} : Cid \times Cid \times Mid \to Set[APair]$ are total mappings.

In a proof environment $\mathcal{E}$, the mapping $L_{\mathcal{E}}$ reflects the class hierarchy and the two mappings $S_{\mathcal{E}}$ and $R_{\mathcal{E}}$ organize the properties collected so far during analysis.

Assume that m is defined in B and that the user gives a specification $m@B(\overline{x}) : (p, q)$ in the context of class C. During the analysis of class C, the specification (p, q) is included in the *specification set* $S(C, B, m)$. We use the notation $S(C, B.m)$ for $S(C, B, m)$ to emphasize that m is defined in B. A proof outline for the method body must then be supplied by the developer, where method calls are decorated with pre- and postconditions. For each internal late bound call $\{r\}\, n(\overline{y})\, \{s\}$ in this outline, the requirement (r, s) is included in the *requirement set* $R(C, B, n)$ by the analysis of the outline. Here, C denotes the class that *imposes* the requirement, as the original specification (p, q) is given by C, and B is the call-site. We use the notation

$R(C, B\#n)$ for $R(C, B, n)$ to emphasize that B is the call-site. Remark that during the analysis of C, the requirement (r, s) is verified for the method that the call will bind to in the context of class C. The inclusion of (r, s) in $R(C, B\#n)$ acts a restriction to future subclasses of C: the requirements made by late bound calls in the proof outline for the body of m are imposed on subclasses of C. Especially, if the method n is overridden in a subclass D, the requirements contained in $R(C, B\#n)$ are verified for the new definition of n when D is analyzed. For the call $\{r\}\, n(\overline{y})\, \{s\}$, the requirement (r, s) then holds for all method implementations that the call can bind to in the context of any class below C. This means that we may rely on the specification (p, q) of $\mathit{body}(B, m)$ also when the method is executed on a subclass instance, i.e., where the internal late bound calls in m are bound in context of the subclass.

If $(p, q) \in S(C, B.m)$, we may assume this assertion pair when reasoning about calls that can bind to $\mathit{body}(B, m)$. Especially, we may assume (p, q) when reasoning about static calls $m@B(\overline{e})$ and when reasoning about internal late bound calls that can bind to $\mathit{body}(B, m)$. As an important property of our reasoning system, we remark that *overriding* implementations of m in subclasses may satisfy different contracts than the definition in the superclass. Especially, if m is overridden in a subclass D of C, the specification $S(C, B.m)$ is not inherited, i.e., it is not imposed on the new version of m. However, requirements made by superclasses, e.g., $R(C, B\#m)$, are inherited and imposed on the overriding version of m in D.

In general, if the set $S(C, B.m)$ is non-empty, the set was extended during the analysis of C, and $C \leq B$. Likewise, if $R(C, B\#m)$ is non-empty, the set was extended during the analysis of C, and $C \leq B$. Let $C \in \mathcal{E}$ denote that $L_{\mathcal{E}}(C)$ is defined, and $\overline{C} \in \mathcal{E}$ denote $C \in \mathcal{E}$ for each C in $\overline{C}$. For the *empty environment* $\mathcal{E}_\emptyset$, $L_{\mathcal{E}_\emptyset}(C)$ is undefined and $S_{\mathcal{E}_\emptyset}(C, B.m) = R_{\mathcal{E}_\emptyset}(C, B\#m) = \emptyset$ for all $C, B : \mathit{Cid}$ and $m : \mathit{Mid}$. In the following, we let functions indexed by $\mathcal{E}$, e.g., $\mathit{bind}_{\mathcal{E}}(C, B\#m)$, denote that the functions are evaluated over the classes defined in $\mathcal{E}$.

Consider a method m defined in a class B. By the analysis of B declared specification of m given in B will be included in the set $S(B, B.m)$. Subclasses of B may give *additional* specification s for the implementation of m. For example, if a method n is overridden by a subclass C of B, and m calls n, a specification of $\mathit{body}(B, m)$ given in C may account for m's behavior with the overriding version of n. Hence, the specification of m as given in the context of class C is included in the set $S(C, B.m)$.

Example 4. We consider the method `withdraw`, as implemented in class `Account` in Figure 4. By the analysis of that class, the following specification is included in the set $S(\texttt{Account}, \texttt{Account.withdraw})$:

$$(bal = b_0 \ \wedge \ owner = id, bal = b_0 - x)$$

The method `validate` is overridden in subclass `AuthAccount`, which means that withdrawals will succeed if the client id is contained in the attributes of the class `Auth`. The following additional specification of method `withdraw` can then be given in the context of the subclass:

$$(bal = b_0 \ \wedge \ (id = a1 \vee id = a2), bal = b_0 - x)$$

By the analysis of `AuthAccount`, this specification is recorded in the set $S(\texttt{AuthAccount}, \texttt{Account.withdraw})$.

3.3. Entailment

Since we deal with sets of assertion pairs, the standard consequence rule of Hoare Logic [4] is insufficient. We need an entailment relation which allows us to combine information from several assertion pairs. Let p^o denote an expression p with all occurrences of program fields f substituted by the corresponding variables f^o, avoiding name capture. The assertion pair (p, q) is understood as an input/output relation $\forall \overline{z} \ . \ p \Rightarrow q^o$, where f and f^o denotes the input and output values of f, respectively, and $\overline{z}$ are the logical variables in p and q. The entailment relation is defined for assertion pairs and for sets of assertion pairs as follows:

Definition 3. (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 by

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

The relation $\mathcal{U} \rightarrowtriangle (r, s)$ 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 [4]. Note that when proving entailment, program fields (primed and unprimed) are implicitly universally quantified. Furthermore, entailment is reflexive and transitive, and $\mathcal{V} \subseteq \mathcal{U}$ implies $\mathcal{U} \rightarrowtriangle \mathcal{V}$.

3.4. Soundness

In order to preserve the validity of previous proofs, it is crucial for incremental reasoning to preserve the declared specifications for *inherited* methods: for a specification (p, q) included in $S(C, B.m)$ it is safe to rely on (p, q) when $body(B, m)$ is executed on an instance of C or a subclasses of C. With the open world assumption the subclasses of C are unknown when C is analyzed, so soundness is ensured by tracking the requirements that (p, q) imposes on late bound calls in $body(B, m)$. If a method n is overridden in a class D below C, all requirements towards n made by classes above D must be satisfied by $body(D, n)$. This is expressed by $S(D, D.n) \rightarrowtriangle R{\uparrow}(D, n)$, where $R{\uparrow}(D, n)$ is defined as the union of all requirements towards n made above D; i.e., the union of $R(C, B\#n)$ for all $D \leq C \leq B$.

In general soundness means that if $body(B, m)$ is executed on an instance of class D, it must be safe to rely on the specifications in the set $S{\uparrow}(D, B.m)$, which is defined as the union of $S(C, B.m)$ for all classes C where $D \leq C \leq B$. Soundness is formalized by the following definition of sound proof environments and Lemma 1. Let $e : C.m$ denote the external call $e.m$ where C is the type of e.

Definition 4. (Sound environments.) Let $B, C, D : Cid$ and $m, n : Mid$. A *sound environment* $\mathcal{E} : Env$ satisfies the following two conditions for all $B, C \in \mathcal{E}$ and m:

1. $\forall (p, q) \in S_{\mathcal{E}}(C, B.m)\ .\ \exists O\ .\ O \vdash_{PL} body_{\mathcal{E}}(B, m) : (p, q)$
   $\wedge\ Internal_{\mathcal{E}}(C, B, O) \wedge External_{\mathcal{E}}(O) \wedge Static_{\mathcal{E}}(C, O)$
2. $m \in C.mtds \Rightarrow S_{\mathcal{E}}(C, C.m) \rightarrowtriangle R_{\mathcal{E}}{\uparrow}(C, m)$

where

$$
\begin{array}{l}
Internal_{\mathcal{E}}(C, B, O) \triangleq \forall (\{r\}\, v := n(\overline{e})\, \{s\}) \in O\ .\ \forall D \leq_{\mathcal{E}} C\ . \\
\qquad\qquad S_{\mathcal{E}}{\uparrow}(D, bind_{\mathcal{E}}(D, B\#n).n) \rightarrowtriangle (r', s') \\
External_{\mathcal{E}}(O) \triangleq \forall (\{r\}\, v := e : D.n(\overline{e})\, \{s\}) \in O\ . \\
\qquad\qquad S_{\mathcal{E}}{\uparrow}(D, bind_{\mathcal{E}}(D, D\#n).n) \rightarrowtriangle (r', s') \\
\qquad\qquad \wedge\ R_{\mathcal{E}}{\uparrow}(D, n) \rightarrowtriangle (r', s') \\
Static_{\mathcal{E}}(C, O) \triangleq \forall (\{r\}\, v := n@B(\overline{e})\, \{s\}) \in O\ . \\
\qquad\qquad S_{\mathcal{E}}{\uparrow}(C, bind_{\mathcal{E}}(B, B\#n).n) \rightarrowtriangle (r', s')
\end{array}
$$

where $r' = (r[\overline{z}_0/\overline{z}] \wedge \overline{x} = \overline{e}[\overline{z}_0/\overline{z}])$, and $s' = s[\mathbf{return}/v][\overline{z}_0/\overline{z}]$, and $\overline{x}$ are

the formal parameters of n[1]. Here, the substitution $[\overline{z}_0/\overline{z}]$, where $\overline{z}_0$ is a list of fresh logical variables, hides the local variables of m (including formal parameters).

The soundness of a proof environment can be explained informally as follows: Assume that $(p, q) \in S_{\mathcal{E}}(C, B.m)$ and that there is a proof outline O of $body(B, m)$ for (p, q). For each *internal call* $\{r\}\, n\, \{s\}$ in O and for each subclass D of C, the requirement (r', s') must follow from the specifications of the method definition to which a call is bound for search class D. The assertion pair (r', s') is derived from (r, s) as in Definition 4. For each *external call* $\{r\}\, e : E.n\, \{s\}$ in O, the assertion pair (r', s') must follow from the specification of the method provided by E, and it must be imposed on redefinitions below this type. (Remember that by the healthy binding strategy, the external call will bind to n as found above E or to a redefinition below E.) For each *static call* $\{r\}\, n@A\, \{s\}$ in O, the assertion pair (r', s') must follow from the specification of the method implementation to which the call will bind. The assertion pair is not imposed on method overridings since the call is bound at compile time.

Let $\models_C \{p\}\, t\, \{q\}$ denote $\models \{p\}\, t\, \{q\}$ provided that late bound internal calls in t are bound for search class C, and let $\models_C m(\overline{x}) : (p, q)\{t\}$ be given by $\models_C \{p\}\, t\, \{q\}$. If t is without calls and $\vdash_{PL} \{p\}\, t\, \{q\}$, then $\models \{p\}\, t\, \{q\}$ follows by the soundness of PL. Lemma 1 states that if $(p, q) \in S_{\mathcal{E}}(C, B.m)$ and $body(B, m)$ is executed in an instance of a subclass D of C, a sound environment guarantees that (p, q) is a valid specification. The proof of this lemma can be found in Appendix A.3.

Lemma 1. *Assume a sound environment $\mathcal{E}$: Env and a sound program logic PL. Let B, D : Cid, m : Mid, and (p, q) : APair such that $B, D \in \mathcal{E}$ and $(p, q) \in S_{\mathcal{E}}\uparrow(D, B.m)$. Then $\models_D m(\overline{x}) : (p, q)\{body(B, m)\}$.*

4. The Inference System for Incremental Reasoning

In this section, the incremental reasoning strategy outlined above is formalized as a calculus *LBS(PL)*. Given a sound program logic PL, the calculus builds a proof environment which reflects the class hierarchy and captures

[1]Remark that in the case of external calls where r, s range over **this**, we also need to replace **this** with a fresh name in r' and s'.

$$
\begin{array}{lll}
\mathcal{A} & ::= & \mathcal{M} \mid [\epsilon\,;\,\mathcal{L}] \cdot \mathcal{M} \mid [\langle C :\, \mathcal{O}\rangle\,;\,\mathcal{L}] \cdot \mathcal{M} \\
\mathcal{M} & ::= & \mathit{module}(\overline{L}) \mid \mathcal{M} \cdot \mathcal{M} \\
\mathcal{L} & ::= & \emptyset \mid L \mid \mathit{require}(C, m, (p, q)) \mid \mathcal{L} \cup \mathcal{L} \\
\mathcal{O} & ::= & \epsilon \mid \mathit{anReq}(\overline{M}) \mid \mathit{anSpec}(\overline{MS}) \mid \mathit{anCalls}(C, O) \mid \mathit{verify}(C, m, \overline{R}) \\
 & & \quad \mid \mathit{supCls}(\overline{C}) \mid \mathit{supMtd}(C, \overline{m}) \mid \mathcal{O} \cdot \mathcal{O}
\end{array}
$$

Figure 5: Syntax for the analysis operations. Here, M, MS, and L are as in Figure 1. $\overline{R}$ is a set of assertion pairs, and O is a statement decorated with pre- and post conditions to method calls.

method specifications and requirements. Initially, the environment is empty and the class hierarchy is analyzed in a top-down manner, starting with classes without superclasses. During class analysis, the proof environment is extended in order to keep track of the currently analyzed class hierarchy and the associated method specifications and requirements. Each class is analyzed after all of its superclasses, based on the environment resulting from the analysis of previous classes. Establishing a proof outline for one method body at a given stage of the overall analysis gives rise to (further) proof-obligations, which are tracked by the proof system. The system itself is formalized as a set of derivation rules (cf. Section 4.1), whose traversal through the class hierarchy is driven by a list of analysis operations.

4.1. The Inference System LBS(PL)

An open program may be extended with new classes, and there may be mutual dependencies between these 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, the judgments are of the form $\mathcal{E} \vdash \mathcal{A}$, where $\mathcal{E} : \mathit{Env}$ is a proof environment and $\mathcal{A}$ is a list of *analysis operations*. The syntax for analysis operations is summarized in Figure 5, and the different operations are explained below. For convenience, we let $\overline{L}$ denote both a list and set of

classes. 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 6 and Figure 8. *LBS(PL)* contains in addition structural rules for flattening $\mathcal{O}$ operations on their last argument and for discarding operations with empty arguments. These rules can be found in Appendix A.2. The rules are read from bottom to top, e.g., application of rule NewModule in Figure 6 on the judgement $\mathcal{E} \vdash module(\overline{L})$ leads to the judgement $\mathcal{E} \vdash [\epsilon\,;\,\overline{L}]$.

Environment updates are formalized by the operator $\_ \oplus \_ : Env \times Update \rightarrow Env$, where the second argument represents the update. There are three different environment updates; extending the environment with a new class and extending the specifications or the requirements of a method in a class. The updates are defined as follows, where the notation $M[A \mapsto B]$ means the mapping M where A maps to B:

$$\mathcal{E} \oplus extL(C, \overline{D}, \overline{f}, \overline{M}) = \langle L_{\mathcal{E}}[C \mapsto \langle \overline{D}, \overline{f}, \overline{M} \rangle], S_{\mathcal{E}}, R_{\mathcal{E}} \rangle$$
$$\mathcal{E} \oplus extS(C, D, m, (p, q)) = \langle L_{\mathcal{E}}, S_{\mathcal{E}}[(C, D, m) \mapsto S_{\mathcal{E}}(C, D.m) \cup \{(p, q)\}], R_{\mathcal{E}} \rangle$$
$$\mathcal{E} \oplus extR(C, D, m, (p, q)) = \langle L_{\mathcal{E}}, S_{\mathcal{E}}, R_{\mathcal{E}}[(C, D, m) \mapsto R_{\mathcal{E}}(C, D\#m) \cup \{(p, q)\}] \rangle$$

A new module $module(\overline{L})$ is initiated for analysis by the rule NewModule given in Figure 6. This rule generates an operation of the form $[\epsilon; \mathcal{L}]$, where $\mathcal{L}$ is initialized to $\overline{L}$. The module is analyzed by analyzing the classes in $\overline{L}$ one by one; whenever a class is selected for analysis, the analysis of this class is completed before a new class from $\overline{L}$ is selected. The set $\mathcal{L}$ may contain class definitions and *require* operations as indicated by the production for $\mathcal{L}$ in Figure 5. During module analysis, the set $\mathcal{L}$ contains the unanalyzed classes of $\overline{L}$, and *require* operations that originate from the analysis of external calls in the analyzed classes, as explained below. The rules of *LBS(PL)* ensure that the analysis of this module is completed before the remaining modules are considered.

The rule NewClass selects a class **`class`** C **`extends`** $\overline{D}$ $\{\overline{f}\ \overline{M}\ \overline{MS}\}$ from the current module, and removes this class from the set of unanalyzed classes. The premise $C \notin \mathcal{E}$ ensures that the class has not been previously analyzed, and the premise $\overline{D} \in \mathcal{E}$ ensures that C is analyzed after all of its superclasses.

This rule generates an operation of the form $[\langle C : \mathcal{O} \rangle\,;\,\mathcal{L}]$, where the operations $\mathcal{O}$ are analyzed in the context of class C *before* other classes in the module are analyzed. The selection of a class C initiates three analysis operations in the context of C. The operation $anSpec(\overline{MS})$ initiates the analysis

(NewClass)

$$\frac{C \notin \mathcal{E} \qquad \overline{D} \in \mathcal{E} \qquad \overline{E} = commSup_{\mathcal{E}}(C) \qquad \mathcal{E} \oplus extL(C, \overline{D}, \overline{f}, \overline{M}) \vdash [\langle C : \ anSpec(\overline{MS}) \cdot anReq(\overline{M}) \cdot supCls(\overline{E}) \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\epsilon \ ; \ \{\textbf{class}\ C\ \textbf{extends}\ \overline{D}\ \{\overline{f}\ \overline{M}\ \overline{MS}\}\} \cup \mathcal{L}] \cdot \mathcal{M}}$$

(EmpModule)

$$\frac{\mathcal{E} \vdash \mathcal{M}}{\mathcal{E} \vdash [\epsilon \ ; \ \emptyset] \cdot \mathcal{M}}$$

(NewSpec)

$$\frac{\mathcal{E} \vdash [\langle C : \ verify(bind_{\mathcal{E}}(D, D\#m), m, (p, q)) \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C : \ anSpec(m@D(\overline{x}) : (p, q)) \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}$$

(NewModule)

$$\frac{\mathcal{E} \vdash [\epsilon \ ; \ \overline{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash module(\overline{L}) \cdot \mathcal{M}}$$

(NewMtd)

$$\frac{\mathcal{E} \vdash [\langle C : \ verify(C, m, R_{\mathcal{E}}\uparrow(C.inh, m)) \cdot \mathcal{O} \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C : \ anReq(m(\overline{x})\{t\}) \cdot \mathcal{O} \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}$$

(EmpClass)

$$\frac{\mathcal{E} \vdash [\epsilon \ ; \ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C : \ \epsilon \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}$$

(ReqDer)

$$\frac{S_{\mathcal{E}}\uparrow(C, D.m) \rightarrowtriangle (p, q) \qquad \mathcal{E} \vdash [\langle C : \ \mathcal{O} \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C : \ verify(D, m, (p, q)) \cdot \mathcal{O} \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}$$

(ReqNotDer)

$$\frac{O \vdash_{PL} body_{\mathcal{E}}(D, m) : (p, q) \qquad \mathcal{E} \oplus extS(C, D, m, (p, q)) \vdash [\langle C : \ anCalls(D, O) \cdot \mathcal{O} \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C : \ verify(D, m, (p, q)) \cdot \mathcal{O} \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}$$

(Internal)

$$\frac{E = bind_{\mathcal{E}}(C, D\#m) \qquad \mathcal{E} \oplus extR(C, D, m, (r', s')) \vdash [\langle C : \ verify(E, m, (r', s')) \cdot \mathcal{O} \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C : \ anCalls(D, \{r\}\, v := m(\overline{e})\, \{s\}) \cdot \mathcal{O} \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}$$

(Static)

$$\frac{\mathcal{E} \vdash [\langle C : \ verify(bind_{\mathcal{E}}(B, B\#m), m, (r', s')) \cdot \mathcal{O} \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C : \ anCalls(D, \{r\}\, v := m@B(\overline{e})\, \{s\}) \cdot \mathcal{O} \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}$$

(External)

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

(extReq)

$$\frac{C \in \mathcal{E} \qquad R_{\mathcal{E}}\uparrow(C, m) \rightarrowtriangle (p, q) \qquad S_{\mathcal{E}}\uparrow(C, bind(C, C\#m).m) \rightarrowtriangle (p, q) \qquad \mathcal{E} \vdash [\epsilon \ ; \ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\epsilon \ ; \ \{require(C, m, (p, q))\} \cup \mathcal{L}] \cdot \mathcal{M}}$$

Figure 6: The inference system. In the call rules, we have as before that $r' = (r[\overline{z}_0/\overline{z}] \wedge \overline{x} = \overline{e}[\overline{z}_0/\overline{z}])$, and $s' = s[\textbf{return}/v][\overline{z}_0/\overline{z}]$, where $\overline{x}$ are the formal parameters of n, and $\overline{z}$ are the local variables of the calling method.

of the different method specifications given in C; the operation $\mathit{anReq}(\overline{M})$ initiates the analysis of the requirements imposed by the superclasses of C on method definitions in C. The operation $\mathit{supCls}(\overline{E})$ initiates the analysis of *delayed requirements*, as described below.

For each method definition $m(\overline{x})\{t\}$ in $\overline{M}$, flattening of $\mathit{anReq}(\overline{M})$ leads to an operation $\mathit{anReq}(m(\overline{x})\{t\})$. This operation is analyzed by rule NewMtd, leading to an operation $\mathit{verify}(C, m, R{\uparrow}(C.\mathit{inh}, m))$, where $R{\uparrow}(C.\mathit{inh}, m)$ contains the requirements towards m that are imposed by superclasses of C. For each method specification $m@D(x) : (p, q)$, contained in $\overline{MS}$, application of rule NewSpec leads to an operation $\mathit{verify}(\mathit{bind}(D, D\#m), m, (p, q))$. These *verify* operations are analyzed in the context of class C. In general, an operation $\mathit{verify}(B, n, (p, q))$, for some class $B \geq C$ and method n defined in B, means that the assertion pair (p, q) will be analyzed for the implementation of n as found in class B.

The two operations $\mathit{anReq}(\overline{M})$ and $\mathit{anSpec}(\overline{MS})$ generated by NewClass thereby ensures that: (1) If method m is defined in C and some superclass of C imposes some requirement (r, s) towards m, i.e., m overrides a superclass definition, then an operation $\mathit{verify}(C, m, (r, s))$ is generated, and (2) for each method specification $m@D(\overline{x}) : (p, q)$ given in the context of C, an operation $\mathit{verify}(\mathit{bind}(D, D\#m), m, (p, q))$ is generated.

The generated *verify* operations are analyzed by either rule ReqDer or rule ReqNotDer. For a method m defined in a class B above C, the set $S(C, B.m)$ is initially empty. This set can only be extended during the analysis of class C, by rule ReqNotDer. This rule requires that if $S(C, B.m)$ is extended with (p, q), then a new proof outline O must be provided for the body of m such that $O \vdash_{PL} \mathit{body}(B, m) : (p, q)$. The analysis then continues by considering the decorated method body by means of an $\mathit{anCalls}(B, O)$ operation as described below. The assertion pair (p, q) then becomes a new specification of $B.m$ in the context of C, and (p, q) is itself assumed when analyzing the method body. This captures the standard approach to reasoning about recursive method calls [21].

Consider next the analysis of some operation $\mathit{verify}(B, m, (p, q))$. The set $S(C, B.m)$ is incrementally extended during the analysis of C, and it might therefore be the case that (p, q) follows by entailment from the already verified assertion pairs for this method, i.e., $S(C, B.m) \twoheadrightarrow (p, q)$. In this case, no further analysis of (p, q) is needed, and the operation is then discarded by rule ReqDer. Otherwise, a proof outline must be provided for the method body, and the operation is verified by rule ReqNotDer as described above.

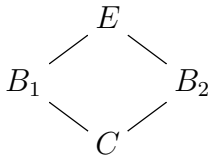

Figure 7: Example of a class hierarchy with diamond inheritance.

In general, B may be a superclass of C, which means that (p, q) may follow from already verified assertion pairs in supeclasses. In rule ReqDer, this is captured by the relation $S\!\uparrow\!(C, B.m) \twoheadrightarrow (p, q)$. The definition of $S\!\uparrow\!(C, B.m)$ can be found in Appendix A.1, note that $S\!\uparrow\!(C, B.m)$ reduces to $S(C, B.m)$ when $B = C$.

Next we consider an operation $anCalls(B, O)$ generated by ReqNotDer. Here, O is a proof outline for the body of some method in class B, where method call statements are decorated with pre- and postconditions. By the structural rules of Appendix A.2, we assume that $anCalls(C, t_1; t_2)$ is decomposed to $anCalls(C, t_1) \cdot anCalls(C, t_2)$, and that $anCalls(C, t)$ is discarded if there are no call statements in t. The remaining statements are analyzed by rules Internal, Static, and External.

An internal late bound call $\{r\}\, v := n(\overline{e})\, \{s\}$ is handled by rule Internal. Two steps are taken for the analysis of the requirment to the call, where (r', s') is given by (r, s) as defined in Figure 6:

1. An operation $verify(E, n, (r', s'))$ is generated, where E the class that the call will bind to for search class C, i.e., $E = bind(C, B\#n)$.
2. (r', s') is included in $R(C, B\#n)$.

For the generated *verify* operation, (r', s') either follows from the already established assertion pairs $S\!\uparrow\!(C, E.n)$, which means that the requirement can be discarded by rule ReqDer. Otherwise, the operation is analyzed by rule ReqNotDer, requiring that a proof outline O' is provided such that $O' \vdash_{PL} body(E, n) : (r', s')$. This proof outline is analyzed in the same manner as the original proof outline for m. This adds (r', s') to $S(C, E.n)$, trivializing the proof of $S\!\uparrow\!(C, E.n) \twoheadrightarrow (r', s')$. The extension of the R mapping in step 2 ensures that future redefinitions of n must respect the new requirement (r', s'), i.e., the requirement is imposed whenever redefinitions are considered by NewMtd.

A static call $\{r\}\, v := n@B(\overline{e})\, \{s\}$ is handled by rule Static. The *verify* operation generated by this rule ensures that (r', s') follows from the specifi-

$$\frac{\mathcal{E} \vdash [\langle C : \ supMtd(D, called_{\mathcal{E}}(D) \setminus C.mtds) \cdot \mathcal{O} \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C : \ supCls(D) \cdot \mathcal{O} \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}} \ (\textsc{SupMtd})$$

$$\frac{E = bind_{\mathcal{E}}(C, D\#m) \quad \mathcal{E} \vdash [\langle C : \ verify(E, m, delReq_{\mathcal{E}}(C, D\#m)) \cdot \mathcal{O} \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C : \ supMtd(D, m) \cdot \mathcal{O} \rangle \ ; \ \mathcal{L}] \cdot \mathcal{M}} \ (\textsc{SupReq})$$

Figure 8: The inference rules of *LBS(PL)* for delayed requirements.

cation of the method that the call will bind to. Note that this rule does not extend the R mapping, since the call is bound at compile time.

External calls $\{r\}\, v := e.n(\overline{e})\, \{s\}$, with $e : E$, are analyzed by rule External. The assertion pair (r', s') is lifted outside the context of the analyzed class by this operation, and shifted into the set $\mathcal{L}$ of analysis operations as a *require* operation. Rule extReq analyzes a *require* operation, and can be applied after the initial analysis of the declared class of the callee. Rule extReq ensures that (r', s') follows from the specification of n given by the static type E of e, and it is imposed on redefinitions of n below E.

The $supCls(\overline{E})$ operation that is generated by rule NewClass is used to take special care of the case where the new class C introduces a *diamond* in the class hierarchy. An example of a small class hierarchy with diamond inheritance can be found in Figure 7. Here, class C extends two different classes B_1 and B_2, and both these classes inherit class E. We then say that E is a *common superclass* of C. In general, we say that C introduces a diamond with common superclass E if there exists two different classes in $C.inh$ and E is above both these classes. There may be more than one common superclass of C, and we let their union be denoted by $commSup(C)$. A formal definition of *commSup* can be found in Appendix A.1.

The purpose of the $supCls(\overline{E})$ operation, where $\overline{E}$ is $commSup(C)$ as given by rule NewClass, is explained by considering the class hierarchy in Figure 7. Consider first the situation before introduction of class C. Assume that the class E implements two methods m and n, and that there is an internal late bound call to n in the body of m. Assume furthermore that none of these methods are overridden in B_2, and that some specification (p, q) of m is given in the context of this class. As explained above, a proof

outline for $body(E, m)$ is then analyzed in the context of class B_2 which leads to an inclusion of (p, q) in $S(B_2, E.m)$ by rule ReqNotDer. Due to the internal late bound call to n in the body of m, some requirement (r, s) is recorded in $R(B_2, E\#n)$. We assume that the analysis of this requirement succeeds for B_2, i.e., $S\!\uparrow\!(B_2, E.n) \twoheadrightarrow (r, s)$. Assume next that n is overridden in class B_1. As B_1 and B_2 are two unrelated subclasses of E, they can be analyzed in any order, and the requirements imposed by one of them are not imposed on the other. Especially, requirements contained in $R(B_2, E\#n)$ are not imposed on n in E as a method call made on an instance of B_2 cannot be bound to a definition in class E. Consider next the analysis of class C, assuming that the method n is not overridden by C. When m is executed on an instance of C, the internal late bound call to n is bound to the definition in B_1, i.e., $bind(C, E\#n) = B_1$. In order to rely on the already verified specification $S(B_2, E.m)$ of m, the analysis of C must ensure that (r, s) follows from the specification for $B_1.n$. In this manner, the verification of $R(B_2, E\#n)$ is *delayed* until a method call with these requirements can actually be bound to $B_1.n$. We then refer to $R(B_2, E\#n)$ as a set of *delayed requirements*. In a more generalized setting, the same argument applies to all such requirements made by any class between C and the common superclass E. For a common superclass E of C and method n called internally in E, the function $delReq(C, E\#n)$ returns the union of the delayed requirements. The definition of this function can be found in Appendix A.1. The rules for analyzing the $supCls(\overline{E})$ operation are displayed in Figure 8. If C does not introduce any diamonds, this operation is discarded as the argument will be empty. Otherwise, SupMtd generates a *supMtd* operation for each common superclass. For each method called by a common superclass and not overridden by C, SupReq generates a *verify* operation for the delayed requirements of these calls. The delayed requirements are thereby verified with regard to the implementations that the call will bind to in the context of C.

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 is completed by rule EmpModule. Thus, the analysis of a module is completed after the analysis of the module classes and the *require* operations generated by the analysis of external external calls in these classes have succeeded. Note that a successful analysis of $\mathcal{E} \vdash module(\overline{L})$ has exactly one leaf node $\mathcal{E}' \vdash [\epsilon ; \emptyset]$, and we call $\mathcal{E}'$ the *environment resulting* from the

analysis of $module(\overline{L})$.

The analysis of a program is initiated by the judgment $\mathcal{E}_\emptyset \vdash module(\overline{L})$, where $\overline{L}$ is a set of classes that are self-contained with respect to 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, independently of subsequent modules.

With lazy behavioral subtyping, a programmer typically provides S specifications for each class B. Their verification generates R requirements for the internal late bound calls occurring in B, which are imposed on subclass redefinitions of the called methods. In a subclass C, a redefined method m can violate the S specifications of a superclass, but not the R requirements. Note that behavioral subtyping is not implied by this approach. Still, lazy behavioral subtyping supports incremental reasoning under an open world assumption. Class C may provide additional specifications for inherited methods, resulting in additional verification of such methods, which may generate additional R requirements for the future subclasses of C. This means that unrelated subclasses of B may have different R requirements to the same method.

4.2. Soundness of LBS(PL)

The following theorem establishes the soundness of the inference rules, the proof can be found in Appendix A.4.

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

By Lemma 1 and Theorem 1 above, we conclude this section with the following soundness theorem:

Theorem 2 (**Soundness**). *If PL is a sound program logic, then LBS(PL) constitutes a sound proof system, in the sense that the environment resulting from the analysis of a program is sound.*

Proof. In *LBS(PL)*, a program is analyzed as a sequence of *module* operations. 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. □

Example 5. We consider the analysis of the classes in Figure 3. To keep the notation below compact, we sometimes use `A`, `AA`, `FA`, and `MA` as abbreviations for `Account`, `AuthAccount`, `FeeAccount`, and `MyAccount` respectively. Let $\overline{L}$ be the classes in Figure 4, and assume that the analysis of these classes starts with an empty proof environment, i.e., the judgement $\mathcal{E}_\emptyset \vdash \mathit{module}(\overline{L})$ is analyzed. Application of rule NewModule then leads to $\mathcal{E}_\emptyset \vdash [\epsilon\,;\,\overline{L}]$. Let `Auth` be the first class that is selected for analysis by rule NewClass.

Analysis of `Auth`*.* When applying rule NewClass, we arrive at the judgement

$$\mathcal{E}_1 \vdash [\langle \texttt{Auth} :\ \mathit{anSpec}(\texttt{validate@Auth}(a) : (\mathit{true}, \textbf{return} = (a = \mathit{a1} \vee a = \mathit{a2})))\rangle\,;\,\overline{L}']$$

where $\mathcal{E}_1$ is $\mathcal{E}_\emptyset$ extended with the declaration of `Auth`, and $\overline{L}'$ are the remaining classes in Figure 4. For brevity, we here focus on the specification of method `validate`, as declared in Figure 4. We ignore the *anReq* and *supCls* operations generated by NewClass since `Auth` is without superclasses. For the generated *anSpec* operation, rule NewSpec leads to an operation $\mathit{verify}(\texttt{Auth}, \texttt{validate}, (\mathit{true}, \textbf{return} = (a = \mathit{a1} \vee a = \mathit{a2})))$. This operation is analyzed by rule ReqNotDer, leading to the judgement $\mathcal{E}_2 \vdash [\langle \texttt{Auth} :\ \epsilon\rangle\,;\,\overline{L}']$, where

$$\mathcal{E}_2 = \mathcal{E}_1 \oplus \mathit{extS}(\texttt{Auth}, \texttt{Auth}, \texttt{validate}, (\mathit{true}, \textbf{return} = (a = \mathit{a1} \vee a = \mathit{a2})))$$

Thus, the specification is included in $S_{\mathcal{E}_2}(\texttt{Auth}, \texttt{Auth.validate})$. The proof outline for the method body is trivial, and we ignore the generated *anCalls* operation since there are no method calls in the method body. By application of rule EmpClass, we then arrive at $\mathcal{E}_2 \vdash [\epsilon\,;\,\overline{L}']$. The only rule that can be applied now is NewClass, and this rule will select class `Account` for analysis.

Analysis of `Account`*.* As for `Auth`, we ignore the *anReq* and *supCls* operations generated by NewClass. The rule NewClass thereby leads to the judgement

$$\mathcal{E}_3 \vdash [\langle \texttt{A} :\ \mathit{anSpec}(\overline{MS})\rangle\,;\,L]$$

where $\overline{MS}$ is the specifications of `update`, `validate`, and `withdraw` as given in the context of class `Account`, $\mathcal{E}_3$ extends the class mapping of $\mathcal{E}_2$ with `Account`, and L is the remaining class `AuthAccount` of the original module. Remember that `A` is used as shorthand for `Account`.

The specifications of methods `update` and `validate` are analyzed as described above for the `validate` method of `Auth`. After analysis of these two methods, we arrive at the following judgement:

$$\mathcal{E}_4 \vdash [\langle \texttt{A} : \mathit{anSpec}(\texttt{withdraw@A}(\mathit{id}, x) : (\mathit{bal} = b_0 \wedge \mathit{owner} = \mathit{id}, \mathit{bal} = b_0 - x))\rangle ; L]$$

where

$$\mathcal{E}_4 = \mathcal{E}_3 \oplus \mathit{extS}(\texttt{A}, \texttt{A}, \texttt{update}, (\mathit{bal} = b_0, \mathit{bal} = b_0 + y)) \oplus \mathit{extS}(\texttt{A}, \texttt{A}, \texttt{validate}, (\mathit{true}, \textbf{return} = (\mathit{owner} = \mathit{id})))$$

The specification of `withdraw` can be verified for the method body by the following proof outline:

$\{\mathit{bal} = b_0 \wedge \mathit{owner} = \mathit{id}\}$ `v := validate(id)` $\{\mathit{bal} = b_0 \wedge v = \mathit{true}\}$
if `v` **then** $\{\mathit{bal} = b_0\}$`update(-x)`$\{\mathit{bal} = b_0 - x\}$ **fi**

By application of NewSpec and ReqNotDer followed by decomposition of the proof outline, we thereby reach the following judgement:

$$\mathcal{E}_5 \vdash [\langle \texttt{A} : \mathit{anCalls}(\texttt{A}, \{\mathit{bal} = b_0 \wedge \mathit{owner} = \mathit{id}\}\ \texttt{v:=validate(id)}\ \{\mathit{bal} = b_0 \wedge v = \mathit{true}\}) \cdot \mathit{anCalls}(\texttt{A}, \{\mathit{bal} = b_0\}\ \texttt{update(-x)}\ \{\mathit{bal} = b_0 - x\})\rangle ; L]$$

where

$$\mathcal{E}_5 = \mathcal{E}_4 \oplus \mathit{extS}(\texttt{A}, \texttt{A}, \texttt{withdraw}, (\mathit{bal} = b_0 \wedge \mathit{owner} = \mathit{id}, \mathit{bal} = b_0 - x))$$

The two *anCalls* operations are analyzed by rule Internal. In each case, the generated *verify* operation follows by entailment from the specification of the called method, and is discarded by rule ReqDer. (For brevity we ignore the trivial specification of `validate`, expressing that `bal` is not changed by the method.) After application of rule EmpClass, we then arrive at the following judgement:

$$\mathcal{E}_6 \vdash [\epsilon ; L] \tag{1}$$

where

$$\mathcal{E}_6 = \mathcal{E}_5 \oplus \mathit{extR}(\texttt{A}, \texttt{A}, \texttt{update}, (\mathit{bal} = b_0 \wedge y = -x, \mathit{bal} = b_0 - x)) \oplus \mathit{extR}(\texttt{A}, \texttt{A}, \texttt{validate}, (\mathit{bal} = b_0 \wedge \mathit{owner} = \mathit{id}, \mathit{bal} = b_0 \wedge \textbf{return} = \mathit{true}))$$

Then non-empty S and R sets of environment $\mathcal{E}_6$ are displayed in Figure 9.

$S_{\mathcal{E}_6}(\texttt{Auth}, \texttt{Auth.validate}) = (\mathit{true}, \textbf{return} = (a = a1 \vee a = a2))$
$S_{\mathcal{E}_6}(\texttt{A}, \texttt{A.update}) = (\mathit{bal} = b_0, \mathit{bal} = b_0 + y)$
$S_{\mathcal{E}_6}(\texttt{A}, \texttt{A.validate}) = (\mathit{true}, \textbf{return} = (\mathit{owner} = \mathit{id}))$
$S_{\mathcal{E}_6}(\texttt{A}, \texttt{A.withdraw}) = (\mathit{bal} = b_0 \wedge \mathit{owner} = \mathit{id}, \mathit{bal} = b_0 - x)$
$R_{\mathcal{E}_6}(\texttt{A}, \texttt{A\#update}) = (\mathit{bal} = b_0 \wedge y = -x, \mathit{bal} = b_0 - x)$
$R_{\mathcal{E}_6}(\texttt{A}, \texttt{A\#validate}) = (\mathit{bal} = b_0 \wedge \mathit{owner} = \mathit{id}, \mathit{bal} = b_0 \wedge \textbf{return} = \mathit{true})$

Figure 9: The non-empty S and R sets of proof environment $\mathcal{E}_6$. In addition, the L mapping of $\mathcal{E}_6$ contains the implementation of the two classes `Auth` and `Account`.

Analysis of `AuthAccount`. By application of NewClass to Equation (1), we arrive at the judgement

$$\mathcal{E}_7 \vdash [\langle \texttt{AA} : \ \mathit{anSpec}(\overline{MS}) \cdot \mathit{anReq}(\overline{M}) \cdot \mathit{supCls}(\emptyset) \rangle \ ; \ \emptyset]$$

where $\mathcal{E}_7$ extends $\mathcal{E}_6$ with the implementation of `AuthAccount`, $\overline{MS}$ contains the class specifications, and $\overline{M}$ contains the methods defined by the class. Since there are no common superclasses, we hereafter ignore the *supCls* operation. For method `validate`, the user given specification leads to the operation

$$\mathit{verify}(\texttt{AA}, \texttt{validate}, (\mathit{true}, \textbf{return} = (\mathit{owner} = \mathit{id} \vee a1 = \mathit{id} \vee a2 = \mathit{id})))$$

which is verified by rule ReqNotDer. The method body can be analyzed by the following proof outline:

$\{\mathit{true}\}$ `r:=validate@Account(id)` $\{r = (\mathit{owner} = \mathit{id})\}$
if `(!r)` **then** $\{\mathit{owner} \neq \mathit{id}\}$
`validate@Auth(id)` $\{\mathit{owner} \neq \mathit{id} \wedge r = (a1 = \mathit{id} \vee a2 = \mathit{id})\}$
fi $\{r = (\mathit{owner} = \mathit{id} \vee a1 = \mathit{id} \vee a2 = \mathit{id})\}$; **return** `r`

The pre- and postconditions for the two static calls follow by entailment of the corresponding superclass specification of `validate` (assuming the trivial specification that `validate` in `Auth` do not modify `owner`). Since the calls are static, none of them leads to an extension of the R mapping. By application of rule ReqNotDer followed by analysis of the proof outline, we thereby arrive at the judgement

$$\mathcal{E}_8 \vdash [\langle \texttt{AA} : \ \mathit{anSpec}(\mathit{MS}) \cdot \mathit{anReq}(\overline{M}) \rangle \ ; \ \emptyset]$$

where MS is the user given specification of `withdraw`, $\overline{M}$ is as above, and

$$\mathcal{E}_8 = \mathcal{E}_7 \oplus extS(\texttt{AA}, \texttt{AA}, \texttt{validate}, \\ (true, \textbf{return} = (owner = id \vee a1 = id \vee a2 = id)))$$

For method `withdraw`, the given specification leads to the following operation, which is verified by rule ReqNotDer:

$$verify(\texttt{A}, \texttt{withdraw}, (bal = b_0 \wedge (id = a1 \vee id = a2), bal = b_0 - x))$$

The specification can be verified by the following proof outline:

$\{bal = b_0 \wedge (a1 = id \vee a2 = id)\}$
`v := validate(id)` $\{bal = b_0 \wedge v = true\}$
if `v` **then** $\{bal = b_0\}$`update(-x)`$\{bal = b_0 - x\}$ **fi**

The internal late bound call to `validate` leads to the requirement $(bal = b_0 \wedge (a1 = id \vee a2 = id), bal = b_0 \wedge \textbf{return} = true)$ which is included in $R(\texttt{AA}, \texttt{A\#validate})$. The requirement follows from $S(\texttt{AA}, \texttt{AA.validate})$ by entailment. Correspondingly, internal late bound call to `update` leads to the requirement $(bal = b_0 \wedge y = -x, bal = b_0 - x)$ which is included in $R(\texttt{AA}, \texttt{A\#update})$. For class `AuthAccount`, this call binds to the inherited implementation of `update`, and the requirement follows from the superclass specification of this method. The successful analysis of the *verify* operation above thereby leads to the judgement:

$$\mathcal{E}_9 \vdash [\langle \texttt{AA} : \ anReq(\overline{M}) \rangle \ ; \ \emptyset]$$

where

$$\begin{aligned} \mathcal{E}_9 = \mathcal{E}_8 \\ &\oplus extS(\texttt{AA}, \texttt{A}, \texttt{withdraw}, (bal = b_0 \wedge (id = a1 \vee id = a2), bal = b_0 - x)) \\ &\oplus extR(\texttt{AA}, \texttt{A}, \texttt{validate}, \\ &\qquad (bal = b_0 \wedge (a1 = id \vee a2 = id), bal = b_0 \wedge \textbf{return} = true)) \\ &\oplus extR(\texttt{AA}, \texttt{A}, \texttt{update}, (bal = b_0 \wedge y = -x, bal = b_0 - x)) \end{aligned}$$

It then remains to analyze the *anReq* operation. Since `validate` is the only method defined in `AuthAccount`, it suffices to consider this method by rule NewMtd. This rule generates a *verify* operation for the inherited

$S_{\mathcal{E}_9}(\texttt{AA}, \texttt{AA.validate}) = (\mathit{true}, \textbf{return} = (\mathit{owner} = \mathit{id} \vee \mathit{a1} = \mathit{id} \vee \mathit{a2} = \mathit{id}))$
$S_{\mathcal{E}_9}(\texttt{AA}, \texttt{A.withdraw}) = (\mathit{bal} = b_0 \wedge (\mathit{id} = \mathit{a1} \vee \mathit{id} = \mathit{a2}), \mathit{bal} = b_0 - x)$
$R_{\mathcal{E}_9}(\texttt{AA}, \texttt{A\#validate}) =$
$\qquad (\mathit{bal} = b_0 \wedge (\mathit{a1} = \mathit{id} \vee \mathit{a2} = \mathit{id}), \mathit{bal} = b_0 \wedge \textbf{return} = \mathit{true})$
$R_{\mathcal{E}_9}(\texttt{AA}, \texttt{A\#update}) = (\mathit{bal} = b_0 \wedge y = -x, \mathit{bal} = b_0 - x)$

Figure 10: The non-empty S and R sets of proof environment $\mathcal{E}_9$. The environment contains in addition the implementation of `AuthAccount`, and the assertion pairs of $\mathcal{E}_6$ as displayed in Figure 9.

requirements towards `validate`. The only requirement imposed by superclasses of `AuthAccount` is $R(\texttt{A}, \texttt{A\#validate})$ as displayed in Figure 9, which means that we arrive at the judgement:

$$\mathcal{E}_9 \vdash [\langle \texttt{AA} : \mathit{verify}(\texttt{AA}, \texttt{validate}, R_{\mathcal{E}_9}(\texttt{A}, \texttt{A\#validate}))\rangle ; \emptyset]$$

The verification of this operation succeeds by ReqDer, since the requirement follows by entailment from the specification of `validate` in `AuthAccount`. The analysis of `AuthAccount` is then completed by rule EmpClass. Since this was the last class of the original module operation, analysis of the module is completed by rule EmpModule. Environment $\mathcal{E}_9$ is thereby the environment resulting from the analysis of the module. The non-empty specification and requirement sets of $\mathcal{E}_9$ are summarized in Figure 10.

As a further illustration of the proof system, we consider the implementation in Figure 11. This figure provides an implementation of the two remaining classes `FeeAccount` and `MyAccount` of Figure 3.

Assume that a module operation consisting of these two classes are analyzed based on the environment resulting from the analysis of the previous classes, i.e., the judgement

$$\mathcal{E}_9 \vdash \mathit{module}(\overline{L}_1)$$

is analyzed, where $\overline{L}_1$ denotes the classes in Figure 11. Class `FeeAccount` is analyzed before `MyAccount`.

Analysis of `FeeAccount`. By application of NewModule and NewClass, we arrive at the judgement

$$\mathcal{E}_{10} \vdash [\langle \texttt{FA} : \mathit{anSpec}(\mathit{MS}) \cdot \mathit{anReq}(M)\rangle ; L_1]$$

```
class FeeAccount extends Account { nat fee; nat accFee;
  update(int y) {update@Account(y);
    if y < 0 then accFee := accFee + fee fi}
  monthly() {update@Account(-accFee); accFee:=0}
  withdraw@Account(nat id, nat x):
            (x > 0 ∧ accFee = a0 ∧ id = owner, accFee = a0 + fee)
}
class MyAccount extends FeeAccount AuthAccount {
  bool validate(nat id) {validate@AuthAccount(id)}
}
```

Figure 11: Implementation of the remaining classes in Figure 3.

where *MS* is the specification of `withdraw`, M is method `update`, and L_1 is the remaining class `MyAccount`. We ignore operation *supCls* since `FeeAccount` has no common superclasses, and the *anReq* operation generated for method `monthly` since there are no superclass requirements towards this method. By rule NewSpec, the *anSpec* operation leads to the operation

$$\mathit{verify}(\texttt{A}, \texttt{withdraw}, (x > 0 \wedge \mathit{accFee} = a_0 \wedge \mathit{id} = \mathit{owner}, \mathit{accFee} = a_0 + \mathit{fee}))$$

which is verified by ReqNotDer. We assume a proof outline for `withdraw` with the requirement $(x > 0 \wedge \mathit{accFee} = a_0 \wedge \mathit{id} = \mathit{owner}, x > 0 \wedge \mathit{accFee} = a_0 \wedge \mathbf{return} = \mathit{true})$ towards `validate` and $(\mathit{accFee} = a_0 \wedge y < 0, \mathit{accFee} = a_0 + \mathit{fee})$ towards `update`. The requirement towards `validate` follows by entailment from the superclass specification of the method (assuming the trivial specification that *accFee* is not modified), whereas the requirement towards update needs verification since `update` is overridden by `FeeAccount` and the new implementation of `update` is left unspecified by the developer. This means that the specifiation set $S(\texttt{FA}, \texttt{FA.update})$ is extended with $(\mathit{accFee} = a_0 \wedge y < 0, \mathit{accFee} = a_0 + \mathit{fee})$ which is trivially verified over the method body. The successful analysis of the *verify* operation thereby leads to the following judgement:

$$\mathcal{E}_{11} \vdash [\langle \texttt{FA} : \ \mathit{anReq}(M) \rangle \ ; \ L_1]$$

where

$$\begin{aligned}\mathcal{E}_{11} = \mathcal{E}_{10}\\ \quad \oplus\ extS(\texttt{FA}, \texttt{A}, \texttt{withdraw},\\ \qquad (x > 0 \wedge accFee = a_0 \wedge id = owner, accFee = a_0 + fee))\\ \quad \oplus\ extR(\texttt{FA}, \texttt{A}, \texttt{validate},\\ \qquad (x > 0 \wedge accFee = a_0 \wedge id = owner, x > 0 \wedge accFee = a_0 \wedge \textbf{return} = true))\\ \quad \oplus\ extR(\texttt{FA}, \texttt{A}, \texttt{update}, (accFee = a_0 \wedge y < 0, accFee = a_0 + fee))\\ \quad \oplus\ extS(\texttt{FA}, \texttt{FA}, \texttt{update}, (accFee = a_0 \wedge y < 0, accFee = a_0 + fee))\end{aligned}$$

For the *anReq* operation, we need to consider requirement $R(\texttt{A}, \texttt{A\#update})$ in Figure 9, which is analyzed by a *verify* operation in the context of class `FeeAccount`. At this point in the analysis, the requirement does not follow from the specification of the overriding `update` in `FeeAccount`, and the *verify* operation is analyzed by rule ReqNotDer. In a proof outline for the method body, we may use the requirement itself as a pre/post specification for the static call, and the analysis of the proof outline thereby succeeds by rule Static. Since the requirement imposed by the superclass can be verified also for the subclass, we can rely on the original superclass specification of `withdraw` also when the method is executed on an instance of `FeeAccount`. The analysis of `FeeAccount` then leads to the following judgement:

$$\mathcal{E}_{12} \vdash [\epsilon\ ;\ L_1]$$

where

$$\mathcal{E}_{12} = \mathcal{E}_{11} \oplus extS(\texttt{FA}, \texttt{FA}, \texttt{update}, (bal = b_0 \wedge y = -x, bal = b_0 - x))$$

Analysis of `MyAccount`. In class `MyAccount`, the method `validate` is overridden in order to direct the internal late bound call in `Account` to the version found in `AuthAccount`. By selecting `MyAccount` for analysis, we arrive at the judgement:

$$\mathcal{E}_{13} \vdash [\langle \texttt{MA} :\ anReq(V) \cdot supCls(\texttt{A})\rangle\ ;\ \emptyset]$$

where V is the method `validate`. By rule NewMtd, this operation leads to a *verify* operation for each requirement imposed by the superclasses. As `validate` is implemented by a static call to `validate` in `AuthAccount`, the analysis of all these requirements succeed, and the are included in the set $S(\texttt{MA}, \texttt{MA.validate})$. Let $\mathcal{E}_{14}$ the environment after this extension of

the S mapping. Next we consider the *supCls*(A) operation. By rule SupMtd, this operation leads to a *supMtd* operation for each method called internally by Account, but not overridden by MyAccount. The methods called by Account are update and validate. Since validate is overridden by MyAccount, we are left with the operation *supMtd*(A, update), which is analyzed by rule SupReq. Since *bind*(MA, A#update) = FA, application of this rule gives the operation *verify*(FA, update, $delReq_{\mathcal{E}_{14}}$(MA, A#update)) By applying the definition of *delReq* as found in Appendix A.1, function $delReq_{\mathcal{E}_{14}}$(MA, A#update) evaluates to $R_{\mathcal{E}_{14}}$(AA, A#update). During the analysis of AuthAccount, the call A#update was bound to Account. However, as MyAccount introduces a diamond in the class hierarchy, the call is bound to FeeAccount for instances of MyAccount. In order to rely on the specification S(AA, A.withdraw) we therefore need to ensure that the requirement R(AA, A#update) holds for update as implemented by FeeAccount. The requirement follows by entailment from specification S(FA, FA.update), which means that the *verify* operation succeeds by rule ReqDer. This concludes the verification of *module*($\overline{L}_1$), and $\mathcal{E}_{14}$ is the resulting environment.

The analysis of imposed requirements means that when withdraw is executed on an instance of MyAccount, we may rely on the specifications of this method given by the different superclasses of MyAccount, i.e.

$$S(\texttt{A}, \texttt{A.withdraw}) \cup S(\texttt{AA}, \texttt{A.withdraw}) \cup S(\texttt{FA}, \texttt{A.withdraw}).$$

5. External Specification by Interfaces

Whereas lazy behavioral subtyping provides a flexible framework for reasoning about open object-oriented programs. However, when reasoning about an external call $e.m(\overline{e})$ with $e : E$, the pre/post assertion pair of the call must follow from the R requirements for m that have been established for class E (rules External and extReq in Figure 6). As these R requirements are generated by the internal analysis, they may not provide a suitable basis for reasoning about external properties. One solution to this problem is to extend the R requirements of E by analysis of the external call. However, this can trigger new verification tasks, which make the approach less modular; it will be necessary to verify that the new pre/post assertion pair of the external call holds for E and all subclasses of E. Another approach is to let the programmer provide R requirements for methods which suffice for reasoning

$$
\begin{array}{lcl}
P & ::= & \overline{KL}\{t\} \\
KL & ::= & K \mid L \\
L & ::= & \texttt{class}\ C\ \texttt{extends}\ \overline{C}\ \texttt{implements}\ I\ \{\overline{f}\ \overline{M}\ \overline{MS}\} \\
K & ::= & \texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{\overline{IS}\} \\
IS & ::= & m(\overline{x}) : (p, q)
\end{array}
$$

Figure 12: Syntax for the language *MII*. The syntactic category P of *MI* (see Figure 1) is extended with interfaces, and class definitions are extended with an **implements** clause. The other syntactic categories of Figure 1 remain unchanged. Here, I denotes interface names of type *Iid*.

about external calls. This approach results in a more modular version of lazy behavioral subtyping in the sense that a class need only be analyzed once, but slightly restricts method redefinition.

In this section we use *behavioral interfaces* as a means to specify and reason about external calls. A behavioral interface describes the visible methods of a class and their specifications, and inheritance may be used to form new interfaces from old ones. Behavioral interfaces are used to type object variables (references). Subtyping follows the inheritance hierarchy of interfaces, but need not follow the class hierarchy. A class definition explicitly declares which interface it implements.

This section develops this approach in terms of a language *MII*, which extends *MI*, in which objects and object references are typed by interfaces and each class implements a single interface. We develop the corresponding reasoning framework *LBSI(PL)* for reasoning about *MII* programs, based on proof environments *IEnv* which include interface information and, as before, an underlying sound program logic *PL*.

5.1. A Language with Behavioral Interfaces

The programming language *MII* extends *MI* with interfaces and has the syntax given in Figure 12. The syntax for classes is modified such that a class implements a single interface. An interface I may extend a list $\overline{I}$ of superinterfaces, and declare a set $\overline{IS}$ of method signatures, where behavioral constraints are given as $(pre, post)$ specifications. An interface may provide specifications of methods which are not found in its superinterfaces, and it may declare additional specifications for methods found in its superinterfaces. Interfaces form an inheritance hierarchy in which the relationship between

interfaces is restricted to a form of behavioral subtyping: if I' extends I, then I' is a *subtype* of I and I is a *supertype* of I'. Thus, an interface may not declare method specifications that are in conflict with specifications declared by its superinterfaces. Let $\preceq$ denote the reflexive and transitive subtype relation, which is given by the nominal extends-relation over interfaces. Thus, $I' \preceq I$ if I' equals I or if I' (directly or indirectly) extends I. We say that an interface I *exports* the methods which are declared in I or inherited from the superinterfaces of I, with the associated constraints on method use.

A class C *implements* the interface I given by the **`implements`** clause in the class definition. All the methods exported by I must be defined, satisfying the specifications of I. The analysis of the class must ensure that this requirement holds. 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. An instance of C *supports* I and all superinterfaces of I, ensuring that the object provides the methods exported in 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 C implements I, some class D may extend C without implementing the behavior specified by I: If D implements interface J, then $J \preceq I$ need *not* hold. If J is not a subtype of I, the specifications declared by I will not be imposed on D, and instances of D will not support I. As a consequence, a subclass may reuse and redefine superclass methods within in the framework of lazy behavioral subtyping, since it is free to violate the interface specifications of its superclasses. In this manner, the type and class inheritance hierarchies are separated. Fields in *MII* are typed by interfaces; if an object supports I (or a subtype of I) then the object may be referenced by a field v typed by I, i.e., v may refer to an instance of class C. However, if J is not a subtype of I, the field may not refer to an instance of class D. Static type checking of an assignment $v := e$ must then ensure that the expression e denotes an object supporting the declared interface of v. In this setting, the *substitution principle* for objects can be reformulated as follows:

> *For an object variable v with declared interface I, the object that v refers to at run-time will satisfy the behavioral specification I.*

Reasoning about an external call $e.m(\overline{e})$ can then be based on the declared interface type of the object expression e; the interface hides the actual class of the object referred to by e. This simplifies External to simply check

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

Figure 13: Auxiliary function definitions, using space as the list separator.

interface contracts, and *require* operations are no longer needed in the proof system. Observe that internal method calls $m(\overline{e})$ and $m@C(\overline{e})$ circumvent the interface mechanism, whereas $x.m(\overline{e})$ depends on the declared interface of x (even if x can be reduced to **`this`**).

5.2. The Proof Environment of LBSI(PL)

In *LBSI(PL)*, a class name is bound to a tuple $\langle \overline{D}, I, \overline{f}, \overline{M} \rangle$ of type *IClass*, and the interface implemented by a class is accessible by the observer function *impl*. An interface name is bound to a tuple $\langle \overline{I}, \overline{IS} \rangle$ of type *Interface*, where the list of superinterfaces $\overline{I}$ and the method specifications $\overline{IS}$ are accessible by the observer functions *inh* and *specs*, respectively. The proof environments of *LBSI(PL)* 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 $S_{\mathcal{E}}, R_{\mathcal{E}} : \mathit{Cid} \times \mathit{Cid} \times \mathit{Mid} \to \mathit{Set}[\mathit{APair}]$ are total mappings. Below, we use the notation $S_{\mathcal{E}}(C, D.m)$ for $S_{\mathcal{E}}(C, D, m)$ and $R_{\mathcal{E}}(C, D\#m)$ for $R_{\mathcal{E}}(C, D, m)$.

In *LBSI(PL)*, $I \in \mathcal{E}$ denotes that $K_{\mathcal{E}}(I)$ is defined. We introduce some auxiliary functions, which are formally defined in Figure 13. Let the function $public_{\mathcal{E}}(I)$ denote the set of method identifiers exported by I, so $m \in public_{\mathcal{E}}(I)$ if m is declared in I or inherited from a supertype of I. If $I' \preceq_{\mathcal{E}} I$ then $public_{\mathcal{E}}(I) \subseteq public_{\mathcal{E}}(I')$, since a subtype can only add methods to those of a supertype. Let the function $spec_{\mathcal{E}}(I, m)$ return a *set* of type $Set[APair]$ with the behavioral constraints imposed on m by I. Note that these constraints may stem from I or from a supertype of I and that a subinterface may provide additional specifications of methods inherited from superinterfaces. If $m \in public_{\mathcal{E}}(I)$ and $I' \preceq_{\mathcal{E}} I$, then $spec_{\mathcal{E}}(I, m) \subseteq spec_{\mathcal{E}}(I', m)$.

The binding of method calls. The internal calls discard the interfaces implemented by the different classes, so the binding of internal late bound and static calls remain as defined in Section 2.4. For the binding of external calls, the definition from Section 2.4 is no longer suitable, since the binding is not restricted by the declared class of the callee in *MII*. However, to obtain a healthy binding strategy as explained in Section 2.4, an external call to m on an instance of class C will be bound by $bind(C, C\#m)$. For class C, the specifications of this implementation is given by $S\!\uparrow\!(C, bind(C, C\#m).m)$. For convenience, we let $S\!\uparrow\!(C, m)$ denote this set.

Sound environments. The definition of sound environments is revised to account for interfaces. In Condition 1 below, the pre/post assertion pair of an external call must now follow from the *interface specification* of the called object. Consider a pre/post assertion pair (r, s) 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. The assertion pair (r, s) must thereby follow from the specification of m given by type I, expressed by $spec_{\mathcal{E}}(I, m) \twoheadrightarrow (r, s)$. Condition 2 for sound environments is unchanged from Definition 4, but a third condition 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 $m \in public_{\mathcal{E}}(I)$. For each such method, the behavioral specification declared by I must follow from the specification of the method to which external calls will be bound.

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

$m : Mid$.

1. $\forall (p,q) \in S_{\mathcal{E}}(C, B.m)\ .\ \exists O\ .\ O \vdash_{PL} body_{\mathcal{E}}(B,m) : (p,q)$
   $\wedge\ Internal_{\mathcal{E}}(C,B,O) \wedge External_{\mathcal{E}}(O) \wedge Static_{\mathcal{E}}(C,O)$
2. $m \in C.mtds \Rightarrow S_{\mathcal{E}}(C, C.m) \rightarrowtriangle R_{\mathcal{E}}{\uparrow}(C,m)$
3. $\forall m \in public_{\mathcal{E}}(I)\ .\ S_{\mathcal{E}}{\uparrow}(C,m) \rightarrowtriangle spec_{\mathcal{E}}(I,m)$, where $I = C.impl$

where

$$
\begin{array}{l}
Internal_{\mathcal{E}}(C,B,O) \triangleq \forall (\{r\}\, v := n(\overline{e})\, \{s\}) \in O\ .\ \forall D \leq_{\mathcal{E}} C\ . \\
\qquad\qquad S_{\mathcal{E}}{\uparrow}(D, bind_{\mathcal{E}}(D, B\#n).n) \rightarrowtriangle (r', s') \\
External_{\mathcal{E}}(O) \triangleq \forall (\{r\}\, v := e : I.n(\overline{e})\, \{s\}) \in O\ .\ spec_{\mathcal{E}}(I,n) \rightarrowtriangle (r', s') \\
Static_{\mathcal{E}}(C,O) \triangleq \forall (\{r\}\, v := n@B(\overline{e})\, \{s\}) \in O\ . \\
\qquad\qquad S_{\mathcal{E}}{\uparrow}(C, bind_{\mathcal{E}}(B, B\#n).n) \rightarrowtriangle (r', s')
\end{array}
$$

and $r' = (r[\overline{z}_0/\overline{z}] \wedge \overline{x} = \overline{e}[\overline{z}_0/\overline{z}])$, $s' = s[\mathbf{return}/v][\overline{z}_0/\overline{z}]$, $\overline{x}$ are the formal parameters of n, and $\overline{z}$ are the local variables of the calling method m.[2] Here, $S{\uparrow}(C,m)$ is used as an abbreviation for $S{\uparrow}(C, bind(C, C\#m).m)$.

Lemma 1 is adapted to environments of type *IEnv*. The proof is given in Appendix C.1.

Lemma 2. *Assume sound environment $\mathcal{E}$: IEnv and a sound program logic PL. Let B, D : Cid, m : Mid, and (p,q) : APair such that $B, D \in \mathcal{E}$ and $(p,q) \in S_{\mathcal{E}}{\uparrow}(D, B.m)$. Then $\models_D m(\overline{x}) : (p,q)\{body(B,m)\}$.*

For *environment updates*, we define an operation to update a proof environment with a new interface, and redefine the operation for updating a proof environment with a new class:

$$
\begin{array}{rcl}
\mathcal{E} \oplus extL(C, \overline{D}, I, \overline{f}, \overline{M}) & \triangleq & \langle L_{\mathcal{E}}[C \mapsto \langle \overline{D}, I, \overline{f}, \overline{M} \rangle], K_{\mathcal{E}}, S_{\mathcal{E}}, R_{\mathcal{E}} \rangle \\
\mathcal{E} \oplus extK(I, \overline{I}, \overline{IS}) & \triangleq & \langle L_{\mathcal{E}}, K_{\mathcal{E}}[I \mapsto \langle \overline{I}, \overline{IS} \rangle], S_{\mathcal{E}}, R_{\mathcal{E}} \rangle
\end{array}
$$

5.3. The Calculus LBSI(PL)

In *LBSI(PL)*, judgments have the form $\mathcal{E} \vdash \mathcal{A}$, where $\mathcal{E}$ is the proof environment and $\mathcal{A}$ is a sequence of interfaces and classes. As before we require that superclasses are analyzed before subclasses, and in addition that

[2] For external calls where r, s range over **this**, we also need to replace **this** with a fresh name in r' and s'.

superinterfaces are analyzed before subinterfaces. Furthermore, we assume that an interface is analyzed before it is used by a class. Consequently, whenever a class is analyzed, its implemented 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.

External calls are now verified against the interface specifications of the called methods, so the analysis of a class can be completed without imposing *require* operations on other classes. Consequently, it suffices to consider individual classes and interfaces as the granularity of program analysis in the revised calculus, and the module layer of *LBS(PL)* is omitted. The syntax for analysis operations in *LBSI(PL)* is given by:

$$\begin{aligned}
\mathcal{A} &::= \mathcal{P} \mid \langle C : \mathcal{O} \rangle \cdot \mathcal{P} \\
\mathcal{P} &::= K \mid L \mid \mathcal{P} \cdot \mathcal{P} \\
\mathcal{O} &::= \epsilon \mid \mathit{anReq}(\overline{M}) \mid \mathit{anSpec}(\overline{MS}) \mid \mathit{anCalls}(C, O) \mid \mathit{verify}(C, m, \overline{R}) \\
&\quad \mid \mathit{supCls}(\overline{C}) \mid \mathit{supMtd}(C, \overline{m}) \mid \mathit{intSpec}(\overline{m}) \mid \mathcal{O} \cdot \mathcal{O}
\end{aligned}$$

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

The calculus *LBSI(PL)* for *MII* consists of a (sound) program logic *PL*, a proof environment $\mathcal{E} : \mathit{IEnv}$, the inference rules listed in Figure 14, and modified versions of the inference rules of *LBS(PL)*, except NewClass, External, extReq, NewModule, and EmpModule. Rules NewClass and External are replaced by NewClass′ and External′ as shown in Figure 14. Rule extReq is superfluous because external calls are analyzed in terms of interface specifications, and rules NewModule and EmpModule are redundant as *LBSI(PL)* is without modules. The remaining rules of *LBS(PL)* are modified by removing the module operations as illustrated by NewClass′ and External′. The complete set of rules for *LBSI(PL)* can be found in Appendix B, including structural rules.

The main differences between *LBS(PL)* and *LBSI(PL)* are captured by the rules in Figure 14. Rule NewInt extends the environment with a new interface, but no further analysis of the interface is required. The specifications of the interface will be analyzed with respect to each class that implements the interface. (Recall that interfaces are assumed to appear in the sequence $\mathcal{P}$ before they are used.) Rule NewClass′ is similar to the rule from *LBS(PL)*,

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

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

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

$$\frac{S_{\mathcal{E}}{\uparrow}(C, m) \twoheadrightarrow spec_{\mathcal{E}}(C.impl, m) \qquad \mathcal{E} \vdash \langle C :\ \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C :\ intSpec(m) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{IntSpec})$$

Figure 14: The extended inference system *LBSI(PL)*, where $\mathcal{P}$ is a (possibly empty) sequence of classes and interfaces. Rules NewClass′ and External′ replace NewClass and External from *LBS(PL)*. The other rules of *LBS(PL)* are preserved.

but an operation *intSpec* is introduced to analyze the specifications of the implemented interface. Rule External′ is used for external calls; here, the call is analyzed based on the interface specification of the callee. For each public method of the class, the rule IntSpec is used to verify the interface specification follows from the specification of the method implementation to which external calls will be bound.

Remark that in *LBSI(PL)*, the method specifications play a more active role when analyzing classes. Method specifications are used to establish interface properties, which are used for the analysis of external calls. Thus, external calls are no longer analyzed based on knowledge from the R mapping of the callee. This mapping is only used to analyze internal calls in *LBSI(PL)*.

Example 6. Reconsider the classes in Figure 4 and Figure 11. As a subclass need not satisfy the superclass type, the different classes may implement different interfaces. Assuming a suitably expressive interface specification language, for instance using invariants over communication histories as in

[15], one may provide abstract specifications of how class instances interact with their environment. The class `Account` may implement an interface I defining exactly how the balance is calculated by `deposit` and `withdraw`. One may then specify the result of method `getbal` as a function over the history restricted to completed executions of `deposit` and `withdraw`. The interface I will be violated by the subclass `FeeAccount`, where also executions of method `monthly` will change the balance. This means that this class implement a different interface than I, since the result returned by `getbal` will depend on previous executions of `monthly` (in addition to `deposit` and `withdraw`). However, the internal analysis of the classes is performed as before, and the different S specifications are used to establish the interface properties.

Soundness. In order to show the soundness of *LBSI(PL)*, Theorem 1 is first modified as follows:

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

The proof of this theorem is given in Appendix C.2. We now show soundness for *LBSI(PL)*:

Theorem 4 (**Soundness**)**.** *If PL is a sound program logic, then LBSI(PL) constitutes a sound proof system, in the sence that the environment resulting from the analysis of a program is sound.*

Proof. In *LBSI(PL)*, a program is analyzed as a sequence of classes and interfaces. It follows directly from Definition 6 that the empty environment is sound. Theorem 3 and Lemma 2 guarantee that the environment remains sound during the analysis of classes and interfaces. □

6. Related Work

Multiple inheritance is commonly used in modeling notations such as UML [7], as a concept naturally inherits from several other concepts. However, it has not found its way into prominent programming languages such as Java. This may be due to the complexity of resolving method binding, as discussed in Section 2, which may easily cause ambiguities. However, multiple inheritance is supported in, e.g., C++ [39], CLOS [14], Eiffel [30], Ocaml

[27], POOL [3], Self [11], and Creol [23]. Horizontal name conflicts in C++, POOL, and Eiffel are removed by explicit resolution, after which the inheritance graph may be linearized. Name conflicts also occur in the context of multiple dispatch, or multi-methods [14]. Multi-methods gives a powerful binding mechanism, but reasoning about multi-methods and redefinition is difficult. For the prototype-based language Self [40], which supports the even more flexible mechanism of dynamic inheritance, an elegant *prioritized* binding strategy for multiple inheritance has been proposed in [11]. Each superclass is given a priority. With equal priority, the superclass related to the caller class is preferred. However, explicit class priorities may cause surprises in large class hierarchies: names may become ambiguous through inheritance. If neither class is related to the caller, binding fails.

The approach presented here is directly applicable to Creol since methods are the code structuring mechanism of Creol. The additional features of concurrency control and processor release points can be treated as in [15] (where inheritance is not considered).

There are surprisingly few high-level formal models of multiple inheritance in the literature. Such formalizations have traditionally used the *objects-as-records* paradigm, especially when dealing with (statically) typed languages. Cardelli [10] gives a denotational semantics of "multiple inheritance", concentrating on typing aspects; i.e., the proposed notion of inheritance corresponds to subtyping in modern terminology. Rossie, Friedman, and Wand [37] formalize multiple inheritance using *subobjects*, a run-time data structure used for virtual pointer tables [25, 39]. This work focuses on compile-time issues and does not clarify multiple inheritance at the abstraction level of the programming language. A natural semantics for late binding in Eiffel models the binding mechanism at the abstraction level of the program [6]. Recently, an operational semantics and a type safety proof inspired by C++ have been formalized in Isabelle/HOL [41]. Due to its relative complexity compared to single inheritance, multiple inheritance has been seen as a mixed blessing: one the one hand desirable, on the other hand ambiguous. Thus, a number of proposals have been put forward to allow more flexible modes of code reuse, without being based on traditional multiple inheritance. These include *dynamic* inheritance [11], *nested* inheritance [31], and, perhaps most prominently, *mixin*- or *trait*-based inheritance [8]; The latter is used instead of multiple inheritance in, e.g., Scala [32]. The notion of healthiness proposed in this paper also serves to remove ambiguities from multiple inheritance by avoiding accidental overridings, making multiple inheritance easier to use.

Work on behavioral reasoning for object-oriented programs address languages with single inheritance (e.g., [35, 36, 9]). For late binding, different variations of behavioral subtyping are most common [28, 2, 26], as discussed above. Pierik and de Boer [35] present a sound and complete reasoning system for late bound calls which does not rely on behavioral subtyping. This work, also for single inheritance, is based on a closed world assumption, meaning that the class hierarchy is not open for incremental extensions. To support object-oriented design, proof systems should be constructed for incremental reasoning.

Lately, incremental reasoning, both for single and multiple inheritance, has been considered in the setting of *separation logic* [29, 12, 34]. These approaches support a distinction between static specifications, given for each method implementation, and dynamic specifications used to verify late bound calls. The dynamic specifications are related to our R requirements in the sence that both are imposed on subclass overridings. The dynamic specifications are given at the definition site, in contrast to our work where R requirements are generated by call-site analysis.

7. Conclusion and Future Work

This paper extends the framework of lazy behavioral subtyping from languages with single inheritance to languages with multiple inheritance. Thus, a contribution of the paper is a proof system for an object-oriented kernel language with multiple inheritance. Another contribution is to show that lazy behavioral subtyping can be adapted from single to multiple inheritance in a natural way, making it well-suited for different object-oriented systems. Lazy behavioral subtyping supports incremental reasoning under an open world assumption, where class hierarchies can be gradually extended by inheritance. The approach is more flexible than traditional behavioral subtyping, since user given S specifications are not imposed on overriding methods in subclasses. Only the minimal R requirements resulting from the analysis of internal late bound calls must be preseved by overriding methods. This is demonstrated by the main example. The S specifications of a method definition are used in the verification of static calls to the method and to establish interface properties if the method is public. S specifications may in addition be used in order to establish class invariants.

The paper investigates two versions of the reasoning framework, one for a purely class-based language and another for a language with behavioral inter-

faces. We have presented our formalisms in terms of a small kernel language which captures the main features of object-oriented programming, excluding language features not central to the discussion. Therefore, the results of the paper are applicable to many languages, assuming a healthy binding strategy. We show soundness of the reasoning framework in both cases. A running example illustrates how the approach can be used for formal reasoning in a setting which is more flexible than traditional behavioral subtyping, and how independent class and interface hierarchies can be used for flexible reuse of code.

It has been argued that multiple inheritance is too complex to be applicable in a safe manner. This is mainly due to horizontal name conflicts and method binding which behaves in unexpected ways, causing ambiguities. This paper proposes healthiness requirements on method binding and the use of static calls to reduce these factors. The paper gives a healthy binding mechanism where renaming is not needed, thereby avoiding theoretical and practical problems related to explicit renaming. In this way, our formalism contributes to making the concept of multiple inheritance more attractive. The motivation behind multiple inheritance is to support flexible code reuse. This coincides with the motivation for lazy behavioral subtyping, which is to reason about flexible code reuse. It is therefore interesting to combine these mechanisms. Combining healthiness and lazy behavioral subtyping, two independent class hierarchies will not accidentally interfere with each other when the hierarchies are merged in a common subclass. This suffices to allow incremental reasoning for external calls as well as internal calls.

References

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

[2] P. America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 60–90. Springer-Verlag, 1991.

[3] P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems,*

Languages, and Applications (OOPSLA), volume 25(10), pages 161–168. ACM Press, Oct. 1990.

[4] 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.

[5] K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Systems*. Texts and Monographs in Computer Science. Springer-Verlag, 3rd edition, 2009.

[6] I. Attali, D. Caromel, and S. O. Ehmety. A natural semantics for Eiffel dynamic binding. *ACM Transactions on Programming Languages and Systems*, 18(6):711–729, 1996.

[7] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[8] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / European Conference on Object-Oriented Programming*, pages 303–311. ACM Press, 1990.

[9] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[10] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2-3):138–164, 1988.

[11] C. Chambers, D. Ungar, B.-W. Chang, and U. Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *Lisp and Symbolic Computation*, 4(3):207–222, 1991.

[12] W.-N. Chin, C. David, H.-H. Nguyen, and S. Qin. Enhancing modular oo verification with separation logic. In POPL'08 [1], pages 87–99.

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

[14] L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: An overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *European Conference on Object-Oriented Programming (ECOOP'87)*, volume 276 of *Lecture Notes in Computer Science*, pages 151–170. Springer-Verlag, 1987.

[15] 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.

[16] 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.

[17] 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.

[18] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. *Journal of Logic and Algebraic Programming*, 79(7):578–607, 2010.

[19] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the Join calculus. *Journal of Logic and Algebraic Programming*, 57(1-2):23–69, 2003.

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

[21] 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.

[22] 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.

[23] E. B. Johnsen and O. Owe. A dynamic binding strategy for multiple inheritance and asynchronously communicating objects. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. 3rd International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *Lecture Notes in Computer Science*, pages 274–295. Springer-Verlag, 2005.

[24] 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.

[25] S. Krogdahl. Multiple inheritance in Simula-like languages. *BIT*, 25(2):318–326, 1985.

[26] 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.

[27] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system (release 3.11). Documentation and user's manual.* Institut National de Recherche en Informatique et en Automatique, Nov. 2008.

[28] 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.

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

[30] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2 edition, 1997.

[31] N. Nystrom, S. Chong, and A. C. Meyers. Scalable extensibility via nested inheritance. In *Ninetheeth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '04*. ACM, 2004. In *SIGPLAN Notices.*

[32] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala. A comprehensive step-by-step guide.* Artima Developer, 2008.

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

[34] M. J. Parkinson and G. M. Biermann. Separation logic, abstraction, and inheritance. In POPL'08 [1].

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

[36] 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.

[37] J. G. Rossie Jr., D. P. Friedman, and M. Wand. Modeling subobject-based inheritance. In P. Cointe, editor, *10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 248–274. Springer-Verlag, July 1996.

[38] 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.

[39] B. Stroustrup. Multiple inheritance for C++. *Computing Systems*, 2(4):367–395, Dec. 1989.

[40] D. Ungar and R. B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–206, 1991. Preliminary version appeared in *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications,* 1987, 227-241.

[41] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety proof for multiple inheritance in C++. In P. L. Tarr and W. R. Cook, editors, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pages 345–362. ACM, 2006.

A. Soundness of *LBS(PL)*

A.1. Auxiliary Function Definitions

To establish soundness of *LBS(PL)*, we define the auxiliary functions as listed in Figure 15. They are needed to track the behavioral constraints during the analysis of an inheritance hierarchy.

Function $S\!\uparrow : \mathit{List}[\mathit{Cid}] \times \mathit{Cid} \times \mathit{Mid} \to \mathit{Set}[\mathit{APair}]$ (taking the environment as an implicit argument) is defined such that $S_{\mathcal{E}}\!\uparrow(D, B.m)$ returns the union of $S_{\mathcal{E}}(C, B.m)$ for all $D \leq_{\mathcal{E}} C \leq_{\mathcal{E}} B$. Function $R\!\uparrow : \mathit{List}[\mathit{Cid}] \times \mathit{Mid} \to \mathit{Set}[\mathit{APair}]$ is defined such that $R_{\mathcal{E}}\!\uparrow(D, m)$ returns the union of all $R_{\mathcal{E}}(C, B\#m)$ for $D \leq_{\mathcal{E}} C \leq_{\mathcal{E}} B$, i.e., all requirements towards m that are imposed by classes above D. This function is defined in terms of the function $\mathit{req} : \mathit{Cid} \times \mathit{List}[\mathit{Cid}] \times \mathit{Mid} \to \mathit{Set}[\mathit{APair}]$. Function $\mathit{commSup} : \mathit{Cid} \to \mathit{Set}[\mathit{Cid}]$ returns the set of common superclasses, and is defined in terms of $\mathit{com} : \mathit{List}[\mathit{Cid}] \to \mathit{Set}[\mathit{Cid}]$ and $\mathit{sup} : \mathit{List}[\mathit{Cid}] \to \mathit{Set}[\mathit{Cid}]$. Function $\mathit{delReq} : \mathit{Cid} \times \mathit{Cid} \times \mathit{Mid} \to \mathit{Set}[\mathit{APair}]$, is used to compute *delayed requirements*. The set $R_{\mathcal{E}}(C, B\#m)$ is contained in $\mathit{delReq}_{\mathcal{E}}(D, B\#m)$ if the following two conditions are met:

- $D <_{\mathcal{E}} C \leq_{\mathcal{E}} B$
- There is *no* class G where $\mathit{bind}_{\mathcal{E}}(G, B\#m) = \mathit{bind}_{\mathcal{E}}(D, B\#m)$ for $D <_{\mathcal{E}} G \leq_{\mathcal{E}} C$, i.e., for all classes G strictly above D and below C, the call $m\#B$ binds to a different implementation than the one found by $\mathit{bind}_{\mathcal{E}}(D, B\#m)$.

The function *delReq* is defined in terms of $\mathit{bel} : \mathit{List}[\mathit{Cid}] \times \mathit{Cid} \to \mathit{List}[\mathit{Cid}]$, $\mathit{belReq} : \mathit{List}[\mathit{Cid}] \times \mathit{Cid} \times \mathit{Cid} \times \mathit{Mid} \times \mathit{Cid} \to \mathit{Set}[\mathit{APair}]$, and $\mathit{clReq} : \mathit{List}[\mathit{Cid}] \times \mathit{Cid} \times \mathit{Cid} \times \mathit{Mid} \times \mathit{Cid} \to \mathit{Set}[\mathit{APair}]$. The function $\mathit{bel}_{\mathcal{E}}(C, B)$ returns the names of all classes above C and below B. For each class C below B, function $\mathit{delReq}_{\mathcal{E}}(D, B\#m)$ computes $\mathit{clReq}_{\mathcal{E}}(L, C, B\#m, E)$, where L are the classes below C, and $E = \mathit{bind}_{\mathcal{E}}(D, B\#m)$. Function $\mathit{clReq}_{\mathcal{E}}(L, C, B\#m, E)$ is evaluated by traversing the list L. No requirements are returned if an element $G \in L$ is found such that $\mathit{bind}_{\mathcal{E}}(G, B\#m) = E$. Otherwise, the set $R_{\mathcal{E}}(C, B\#m)$ is returned.

In later proofs, we need the notion of the *depth* of class D below class C. The depth of a class below another class is given as follows:

- Class C is at depth 0 below C.

$$
\begin{array}{lll}
S_{\mathcal{E}}{\uparrow}(\mathit{nil}, B.m) \triangleq \emptyset & & \\
S_{\mathcal{E}}{\uparrow}(C\ L, B.m) \triangleq S_{\mathcal{E}}(B, B.m) \cup S_{\mathcal{E}}{\uparrow}(L, B.m) & & \text{if } C = B \\
S_{\mathcal{E}}{\uparrow}(C\ L, B.m) \triangleq S_{\mathcal{E}}(C, B.m) \cup S_{\mathcal{E}}{\uparrow}(C.\mathrm{inh}\ L, B.m) & & \text{if } C <_{\mathcal{E}} B \\
S_{\mathcal{E}}{\uparrow}(C\ L, B.m) \triangleq S_{\mathcal{E}}{\uparrow}(L, B.m) & & \text{otherwise} \\
\\
R_{\mathcal{E}}{\uparrow}(\mathit{nil}, m) \triangleq \emptyset & & \\
R_{\mathcal{E}}{\uparrow}(C\ L, m) \triangleq \mathit{req}_{\mathcal{E}}(C, C, m) \cup R_{\mathcal{E}}{\uparrow}(C.\mathrm{inh}\ L, m) & & \\
\\
\mathit{req}_{\mathcal{E}}(C, \mathit{nil}, m) \triangleq \emptyset & & \\
\mathit{req}_{\mathcal{E}}(C, B\ L, m) \triangleq R_{\mathcal{E}}(C, B\#m) \cup \mathit{req}_{\mathcal{E}}(C, B.\mathrm{inh}\ L, m) & & \\
\\
\mathit{commSup}_{\mathcal{E}}(C) \triangleq \mathit{com}_{\mathcal{E}}(C.\mathrm{inh}) & & \\
\mathit{com}_{\mathcal{E}}(\mathit{nil}) \triangleq \emptyset & & \\
\mathit{com}_{\mathcal{E}}(C\ L) \triangleq (\mathit{sup}_{\mathcal{E}}(C) \cap \mathit{sup}_{\mathcal{E}}(L)) \cup \mathit{com}_{\mathcal{E}}(L) & & \\
\mathit{sup}_{\mathcal{E}}(\mathit{nil}) \triangleq \emptyset & & \\
\mathit{sup}_{\mathcal{E}}(C\ L) \triangleq \{C\} \cup \mathit{sup}_{\mathcal{E}}(C.\mathrm{inh}\ L) & & \\
\\
\mathit{delReq}_{\mathcal{E}}(D, B\#m) \triangleq \mathit{belReq}_{\mathcal{E}}(\mathit{bel}_{\mathcal{E}}(D.\mathrm{inh}, B), D, B\#m, \mathit{bind}_{\mathcal{E}}(D, B\#m)) & & \\
\mathit{bel}_{\mathcal{E}}(\mathit{nil}, B) \triangleq \emptyset & & \\
\mathit{bel}_{\mathcal{E}}(D\ L, B) \triangleq D\ \mathit{bel}_{\mathcal{E}}(D.\mathrm{inh}\ L, B) & & \text{if } D \leq_{\mathcal{E}} B \\
\mathit{bel}_{\mathcal{E}}(D\ L, B) \triangleq \mathit{bel}_{\mathcal{E}}(L, B) & & \text{otherwise} \\
\mathit{belReq}_{\mathcal{E}}(\mathit{nil}, D, B\#m, E) \triangleq \emptyset & & \\
\mathit{belReq}_{\mathcal{E}}(C\ L, D, B\#m, E) \triangleq \mathit{clReq}_{\mathcal{E}}(\mathit{bel}_{\mathcal{E}}(D.\mathrm{inh}, C), C, B\#m, E) & & \\
\qquad \cup\ \mathit{belReq}_{\mathcal{E}}(L, D, B\#m, E) & & \\
\mathit{clReq}_{\mathcal{E}}(\mathit{nil}, C, B\#m, E) \triangleq R_{\mathcal{E}}(C, B\#m) & & \\
\mathit{clReq}_{\mathcal{E}}(G\ L, C, B\#m, E) \triangleq \emptyset & & \text{if } \mathit{bind}_{\mathcal{E}}(G, B\#m) = E \\
\mathit{clReq}_{\mathcal{E}}(G\ L, C, B\#m, E) \triangleq \mathit{clReq}_{\mathcal{E}}(L, C, B\#m, E) & & \text{otherwise}
\end{array}
$$

Figure 15: Auxiliary function definitions. Here, we let $B, C, D, E, G : \mathit{Cid}$; $m : \mathit{Mid}$; and $L : \mathit{List}[\mathit{Cid}]$.

- Class D is at depth d (where $d > 0$) below class C if there is a class $D' \in D.inh$ such that D' is at depth $d-1$ below C, and for all classes $D'' \in D.inh$ the depth of D'' below C is less or equal to $d-1$ or D'' is not below C.

A.2. Structural Rules of LBS(PL)

This section presents *LBS(PL)* rules for decomposing list-like structures and handling trivial cases. Here $\mathcal{M}$ is a (possibly empty) list of analysis operations.

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

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

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

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

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

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

$$\text{(decompIf)}\quad \frac{\mathcal{E} \vdash [\langle C : \mathit{anCalls}(D, t_1) \cdot \mathit{anCalls}(D, t_2) \cdot \mathcal{O}\rangle ; \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C : \mathit{anCalls}(D, \texttt{if}\ b\ \texttt{then}\ t_1\ \texttt{else}\ t_2\ \texttt{fi}) \cdot \mathcal{O}\rangle ; \mathcal{L}] \cdot \mathcal{M}}$$

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

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

$$\frac{\mathcal{E} \vdash [\langle C : \mathcal{O} \rangle ; \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C : \mathit{supCls}(\emptyset) \cdot \mathcal{O} \rangle ; \mathcal{L}] \cdot \mathcal{M}} \text{(NOSUPCLS)}$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{supCls}(\overline{D_1}) \cdot \mathit{supCls}(\overline{D_2}) \cdot \mathcal{O} \rangle ; \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C : \mathit{supCls}(\overline{D_1}\ \overline{D_2}) \cdot \mathcal{O} \rangle ; \mathcal{L}] \cdot \mathcal{M}} \text{(DECOMPSUPCLS)}$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathcal{O} \rangle ; \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C : \mathit{supMtd}(D, \emptyset) \cdot \mathcal{O} \rangle ; \mathcal{L}] \cdot \mathcal{M}} \text{(NOSUPMTD)}$$

$$\frac{\mathcal{E} \vdash [\langle C : \mathit{supMtd}(D, \overline{m_1}) \cdot \mathit{supMtd}(D, \overline{m_2}) \cdot \mathcal{O} \rangle ; \mathcal{L}] \cdot \mathcal{M}}{\mathcal{E} \vdash [\langle C : \mathit{supMtd}(D, \overline{m_1}\ \overline{m_2}) \cdot \mathcal{O} \rangle ; \mathcal{L}] \cdot \mathcal{M}} \text{(DECOMPSUPMTD)}$$

A.3. Proof of Lemma 1

Lemma 1 shows the important property of sound environment (cf. Definition 4):

> Assume a sound environment $\mathcal{E} : \mathit{Env}$ and a sound program logic PL. Let $B, D : \mathit{Cid}$, $m : \mathit{Mid}$, and $(p, q) : \mathit{APair}$ such that $B, D \in \mathcal{E}$ and $(p, q) \in S_{\mathcal{E}}\uparrow(D, B.m)$. Then $\models_D m(\overline{x}) : (p, q)\{\mathit{body}_{\mathcal{E}}(B, m)\}$.

Proof. By definition of $S\uparrow$, the assumption $(p, q) \in S_{\mathcal{E}}\uparrow(D, B.m)$ implies that there exists some class C such that $D \leq_{\mathcal{E}} C \leq_{\mathcal{E}} B$ and $(p, q) \in S_{\mathcal{E}}(C, B.m)$. By Definition 4, there must be a proof outline O for the method body such that $O \vdash_{PL} \mathit{body}_{\mathcal{E}}(B, m) : (p, q)$. The proof of the lemma proceeds by induction over the call structure of m.

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

Induction step: Each kind of method call is considered in a separate case. If a call to some method n in $\mathit{body}_{\mathcal{E}}(B, m)$ is bound to a definition in class H for search class G, we take $\models_G n(\overline{y}) : (t, u)\{\mathit{body}_{\mathcal{E}}(H, n)\}$ as

the induction hypothesis, for each $(t, u) \in S_{\mathcal{E}}{\uparrow}(G, H.n)$. For a call to n with precondition r, postcondition s, and actual parameters $\overline{e}$, we let $r' = (r[\overline{z}_0/\overline{z}] \wedge \overline{y} = \overline{e}[\overline{z}_0/\overline{z}])$ and $s' = s[\mathbf{return}/v][\overline{z}_0/\overline{z}]$, where the return value of the method is assigned to the variable v by the caller. We consider each kind of method call in O by a separate case.

Late bound internal calls $\{r\}\, v := n(\overline{e})\, \{s\}$**.** By the assumptions of the lemma, internal calls are bound with search class D. Thus, in the induction hypothesis we thereby have $G = D$, and $H = \mathit{bind}_{\mathcal{E}}(D, B\#n)$. Consequently, it suffices to ensure $S_{\mathcal{E}}{\uparrow}(D, H.n) \rightarrowtriangle (r', s')$, which follows by Definition 4.

Static calls $\{r\}\, v := n@A(\overline{e})\, \{s\}$**.** For the induction hypothesis, we have $G = D$ and $H = \mathit{bind}_{\mathcal{E}}(A, A\#n)$. By Definition 4, we then have $S_{\mathcal{E}}{\uparrow}(C, H.n) \rightarrowtriangle (r', s')$. The desired $S_{\mathcal{E}}{\uparrow}(D, H.n) \rightarrowtriangle (r', s')$ then follows by the definition of $S{\uparrow}$ since $D \leq_{\mathcal{E}} C$, i.e., $S_{\mathcal{E}}{\uparrow}(C, H.n) \subseteq S_{\mathcal{E}}{\uparrow}(D, H.n)$.

External calls $\{r\}\, v := e : E.n(\overline{e})\, \{s\}$**.** This call can bound with respect to any class G such that $G \leq_{\mathcal{E}} E$, and let $H = \mathit{bind}_{\mathcal{E}}(G, E\#n)$. By the induction hypothesis, it suffices to ensure:

$$S_{\mathcal{E}}{\uparrow}(G, H.n) \rightarrowtriangle (r', s') \tag{2}$$

For the external call, Definition 4 gives the following assumption:

$$S_{\mathcal{E}}{\uparrow}(E, \mathit{bind}_{\mathcal{E}}(E, E\#n).n) \rightarrowtriangle (r', s') \text{ and } R_{\mathcal{E}}{\uparrow}(E, n) \rightarrowtriangle (r', s') \tag{3}$$

The relation $S_{\mathcal{E}}{\uparrow}(G, H.n) \rightarrowtriangle (r', s')$ is proved by induction over the depth d of G below E.

Base case: $d = 0$, i.e., $G = E$. In this case, we have $H = \mathit{bind}_{\mathcal{E}}(E, E\#n)$. Relation (2) then follows by assumption (3).

Induction step: $d = d' + 1$, i.e., $G <_{\mathcal{E}} E$ at depth d. As the induction hypothesis, we may assume that for any class G' at depth d' below E, that $S_{\mathcal{E}}{\uparrow}(G', \mathit{bind}_{\mathcal{E}}(G', E\#n).n) \rightarrowtriangle (r', s')$. We consider two cases.

Case 1: $n \in G.\mathit{mtds}$. By Definition 4, Condition 2, we then have $S_{\mathcal{E}}(G, G.n) \rightarrowtriangle R_{\mathcal{E}}{\uparrow}(G, n)$. By Figure 15, we have $S_{\mathcal{E}}{\uparrow}(G, G.n) = S_{\mathcal{E}}(G, G.n)$. Since $\mathit{bind}_{\mathcal{E}}(G, E\#n) = G$ and H in Relation (2) therefore equals G in this case, it suffices to ensure $R_{\mathcal{E}}{\uparrow}(G, n) \rightarrowtriangle (r', s')$. By Figure 15, we have $R_{\mathcal{E}}{\uparrow}(E, n) \subseteq R_{\mathcal{E}}{\uparrow}(G, n)$ for $G \leq_{\mathcal{E}} E$, Relation (2) thereby follows by (3) and transitivity of entailment:

$$R_{\mathcal{E}}{\uparrow}(G, n) \rightarrowtriangle R_{\mathcal{E}}{\uparrow}(E, n) \rightarrowtriangle (r', s')$$

Case 2: $n \notin G.mtds$. Since $H = bind_{\mathcal{E}}(G, E\#n)$ and $n \notin G.mtds$, there must by Definition 1 of function *bind* exist some $G' \in G.inh$ such that $bind_{\mathcal{E}}(G', E\#n) = H$. As the induction hypothesis applies to G', we have $S_{\mathcal{E}}\uparrow(G', H.n) \twoheadrightarrow (r', s')$. The desired relation $S_{\mathcal{E}}\uparrow(G, H.n) \twoheadrightarrow (r', s')$ then follows by the definition of $S\uparrow$, since $G \leq_{\mathcal{E}} G'$. □

A.4. Proof of Theorem 1

The theorem establishes the soundness of the inference system of Section 4.1:

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

Proof. Given a sound environment, we prove that the environment extensions preserve soundness. We consider each of the two conditions of Definition 4 in isolation.

Condition 1 of Definition 4. The proof proceeds by induction over the inference rules. The only rule that extends $S_{\mathcal{E}}(C, B.m)$ is ReqNotDer, and this rule ensures that there is a proof outline for $body_{\mathcal{E}}(B, m)$. The set $S_{\mathcal{E}}(C, B.m)$ is only extended during analysis of C. Thus, for any $(p, q) \in S_{\mathcal{E}}(C, B.m)$ we have an O such that $O \vdash_{PL} body_{\mathcal{E}}(B, m) : (p, q)$ and $C \leq_{\mathcal{E}} B$.

If the specification (p, q) is included in $S_{\mathcal{E}}(C, B.m)$ by ReqNotDer, an operation $anCalls(B, O)$ is generated and analyzed in the context of C. Each call statement in the proof outline is analyzed by either Internal, Static or External. We consider each kind of method call in isolation. For a call to n with precondition r, postcondition s, and actual parameters $\overline{e}$, we let $r' = (r[\overline{z}_0/\overline{z}] \wedge \overline{y} = \overline{e}[\overline{z}_0/\overline{z}])$ and $s' = s[\mathbf{return}/v][\overline{z}_0/\overline{z}]$, where $\overline{y}$ are the formal parameters and the return value of the method is assigned to the variable v by the caller.

Late bound internal calls. For each $\{r\}\, v := n(\overline{e})\, \{s\}$ in O, we have $n \in called_{\mathcal{E}}(B)$ and an operation $anCalls(B, \{r\}\, v := n(\overline{e})\, \{s\})$ is analyzed. Rule Internal applies to this operation, ensuring $(r', s') \in R_{\mathcal{E}}(C, B\#n)$. As the class hierarchy is extended, we then need to ensure

$$S_{\mathcal{E}}\uparrow(D, bind_{\mathcal{E}}(D, B\#n).n) \twoheadrightarrow (r', s') \tag{4}$$

for each class D below C as required by Definition 4. This is done by induction over the depth d of D below C.

Base case: $d = 0$, i.e., $D = C$. By (4), we need to ensure $S_{\mathcal{E}}{\uparrow}(C, H.n) \twoheadrightarrow (r', s')$ for $H = \mathit{bind}_{\mathcal{E}}(C, B\#n)$. This case is handled by analysis of the $\mathit{anCalls}(B, \{r\}\, v := n(\overline{e})\, \{s\})$ operation that is generated by the analysis of C. The application of INTERNAL leads to an operation $\mathit{verify}(H, n, (r', s'))$. Since the analysis of this operation succeeds, REQDER or REQNOTDER is applied. The relation $S_{\mathcal{E}}{\uparrow}(C, H.n) \twoheadrightarrow (r', s')$ must hold directly if REQDER is applied. Otherwise, if REQNOTDER is applied, the set $S_{\mathcal{E}}(C, H.n)$ is extended with (r', s'). The desired relation then holds by transitivity of entailment since $S_{\mathcal{E}}{\uparrow}(C, H.n) \twoheadrightarrow S_{\mathcal{E}}(C, H.n) \twoheadrightarrow (r', s')$.

Induction step: $d = d' + 1$, i.e., $D <_{\mathcal{E}} C$ at depth d. As the induction hypothesis, we may assume $S_{\mathcal{E}}{\uparrow}(D', \mathit{bind}_{\mathcal{E}}(D', B\#n).n) \twoheadrightarrow (r', s')$ for any class D' such that $D <_{\mathcal{E}} D' \leq_{\mathcal{E}} C$, i.e., D' is at depth less or equal to d' below C, We consider the two cases $n \in D.\mathit{mtds}$ and $n \notin D.\mathit{mtds}$ separately.

Case 1: $n \in D.\mathit{mtds}$. Method n is defined in D, which means that $\mathit{bind}_{\mathcal{E}}(D, B\#n) = D$. By (4), the relation $S_{\mathcal{E}}(D, D.n) \twoheadrightarrow (r', s')$ must then be ensured. By the definition of $R{\uparrow}$, we have $R_{\mathcal{E}}(C, B\#n) \subseteq R_{\mathcal{E}}{\uparrow}(D.\mathit{inh}, n)$ since $D <_{\mathcal{E}} C$.

Since $n \in D.\mathit{mtds}$, NEWMTD will be applied to n by the analysis of D, generating an operation $\mathit{verify}(D, n, R_{\mathcal{E}}{\uparrow}(D.\mathit{inh}, n))$, which leads to analysis of $\mathit{verify}(D, n, (r', s'))$ by decomposition. This operation either succeeds by REQDER or REQNOTDER, both ensuring the desired $S_{\mathcal{E}}(D, D.n) \twoheadrightarrow (r', s')$.

Case 2: $n \notin D.\mathit{mtds}$. For this case, we consider $B \notin \mathit{commSup}_{\mathcal{E}}(D)$ and $B \in \mathit{commSup}_{\mathcal{E}}(D)$ separately.

Case 2a: $B \notin \mathit{commSup}_{\mathcal{E}}(D)$. In this case, there is exactly one $D' \in D.\mathit{inh}$ such that $D' \leq_{\mathcal{E}} B$. Since $D <_{\mathcal{E}} C \leq_{\mathcal{E}} B$, we then have $D' \leq_{\mathcal{E}} C \leq_{\mathcal{E}} B$ and that D' is the only class in $D.\mathit{inh}$ that is below C. (If these conditions were not met, then B would be in $\mathit{commSup}_{\mathcal{E}}(D)$.) Therefore, the induction hypothesis applies to D', and we have $S_{\mathcal{E}}{\uparrow}(D', E.n) \twoheadrightarrow (r', s')$, where $E = \mathit{bind}_{\mathcal{E}}(D', B\#n)$. By Definition 1 of *bind*, we then have also $E = \mathit{bind}_{\mathcal{E}}(D, B\#n)$. Proof obligation (4) thereby reduces to $S_{\mathcal{E}}{\uparrow}(D, E.n) \twoheadrightarrow (r', s')$, which follows by the definition of $S{\uparrow}$ and the induction hypothesis.

Case 2b: $B \in \mathit{commSup}_{\mathcal{E}}(D)$. Let $E = \mathit{bind}_{\mathcal{E}}(D, B\#n)$. By (4), we need to ensure $S_{\mathcal{E}}{\uparrow}(D, E.n) \twoheadrightarrow (r', s')$ where $(r', s') \in R_{\mathcal{E}}(C, B\#n)$.

For all classes G such that $D <_{\mathcal{E}} G \leq_{\mathcal{E}} C$, the induction hypothesis gives $S_{\mathcal{E}}{\uparrow}(G, \mathit{bind}_{\mathcal{E}}(G, B\#n).n) \twoheadrightarrow (r', s')$. There are two possibilities:

- There exists a G such that $D <_{\mathcal{E}} G \leq_{\mathcal{E}} C$ and $\mathit{bind}_{\mathcal{E}}(G, B\#n) = E$. By the induction hypothesis, we then have $S_{\mathcal{E}}{\uparrow}(G, E.n) \twoheadrightarrow (r', s')$. Proof

obligation $S_{\mathcal{E}}\!\uparrow\!(D, E.n) \twoheadrightarrow (r', s')$ then follows by the definition of $S\!\uparrow$ since G is above D.

- There exists no class G such that $D <_{\mathcal{E}} G \leq_{\mathcal{E}} C$ and $\mathit{bind}_{\mathcal{E}}(G, B\#n) = E$. By the definition of *delReq*, the set $R_{\mathcal{E}}(C, B\#n)$ is then included in $\mathit{delReq}_{\mathcal{E}}(D, B\#n)$.

  When class D is analyzed by NewClass, a $\mathit{supCls}(\mathit{commSup}_{\mathcal{E}}(D))$ operation is generated and analyzed in the context of D. Since $B \in \mathit{commSup}_{\mathcal{E}}(D)$, decomposition of this operation leads to a $\mathit{supCls}(B)$ operation. Furthermore, since $n \in \mathit{called}_{\mathcal{E}}(B) \setminus D.\mathit{mtds}$, the application of rule SupMtd and further decomposition leads to an operation $\mathit{supMtd}(B, n)$. By rule SupReq, we then arrive at an operation $\mathit{verify}(E, n, \mathit{delReq}_{\mathcal{E}}(D, B\#n))$. By decomposition of the requirement set, we then arrive at an operation $\mathit{verify}(E, n, (r', s'))$. As in the base case, application of either ReqDer or ReqNotDer then ensures $S_{\mathcal{E}}\!\uparrow\!(D, E.n) \twoheadrightarrow (r', s')$.

Static calls. For each call $\{r\}\, v := n@A(\overline{e})\, \{s\}$ in the proof outline O, rule Static will be applied. The operation $\mathit{verify}(\mathit{bind}_{\mathcal{E}}(A, A\#n), n, (r', s'))$ will be generated by this rule, and the operation is verified during analysis of C. This operation succeeds by ReqDer or ReqNotDer, both ensuring the desired relation $S_{\mathcal{E}}\!\uparrow\!(C, \mathit{bind}_{\mathcal{E}}(A, A\#n).n) \twoheadrightarrow (r', s')$ of Definition 4.

External calls. For each $\{r\}\, v := e : E.n(\overline{e})\, \{s\}$ in O, External is applied. This rule will generate a $\mathit{require}(E, n, (r', s'))$ operation. This operation succeeds by extReq, which ensures $S_{\mathcal{E}}\!\uparrow\!(E, \mathit{bind}_{\mathcal{E}}(E, E\#n).n) \twoheadrightarrow (r', s')$ and $R_{\mathcal{E}}\!\uparrow\!(E, n) \twoheadrightarrow (r', s')$ as required by Definition 4.

Condition 2 of Definition 4. For each method $m \in C.\mathit{mtds}$, we must establish $S_{\mathcal{E}}(C, C.m) \twoheadrightarrow R_{\mathcal{E}}\!\uparrow\!(C, m)$. When class C is analyzed by NewClass, an operation *anReq* is generated for each method defined in C. By NewMtd, possibly followed by decomposition of the requirement set, an operation $\mathit{verify}(C, m, (p, q))$ is generated for each $(p, q) \in R_{\mathcal{E}}\!\uparrow\!(C.\mathit{inh}, m)$. These operations succeeds either by ReqDer or ReqNotDer, ensuring $S_{\mathcal{E}}(C, C.m) \twoheadrightarrow R_{\mathcal{E}}\!\uparrow\!(C.\mathit{inh}, m)$.

Since $R_{\mathcal{E}}\!\uparrow\!(C, m) = \mathit{req}_{\mathcal{E}}(C, C, m) \cup R_{\mathcal{E}}\!\uparrow\!(C.\mathit{inh}, m)$ by the definition of $R\!\uparrow$, it then remains to ensure $S_{\mathcal{E}}(C, C.m) \twoheadrightarrow \mathit{req}_{\mathcal{E}}(C, C, m)$. If $(r, s) \in \mathit{req}_{\mathcal{E}}(C, C, m)$, there exists, by the definition of *req*, a class B such that $C \leq_{\mathcal{E}} B$ and $(r, s) \in R_{\mathcal{E}}(C, B\#m)$. In order for (r, s) to be included in this

set, INTERNAL is applied during analysis of C. This rule will generate an operation $\mathit{verify}(\mathit{bind}_{\mathcal{E}}(C, B\#m), m, (r, s))$ which equals $\mathit{verify}(C, m, (r, s))$ since m is defined in C. Again, analysis of this operation succeeds by REQDER or REQNOTDER, which both ensures the desired $S_{\mathcal{E}}(C, C.m) \twoheadrightarrow (r, s)$. □

B. *LBSI(PL)* Inference Rules

This section shows the complete set of inference rules for *LBSI(PL)*. More specifically, the syntax of the analysis operations used by the rules here are given in Section 5.3, together with an explanation of the characteristic rules for dealing with interfaces.

B.1. Main Rules

For the rules INTERNAL, STATIC, and EXTERNAL′, we have that $r' = (r[\overline{z}_0/\overline{z}] \wedge \overline{x} = \overline{e}[\overline{z}_0/\overline{z}])$, $s' = s[\mathbf{return}/v][\overline{z}_0/\overline{z}]$, $\overline{x}$ are the formal parameters of n, and $\overline{z}$ are the local variables of the calling method.

$$\frac{I \notin \mathcal{E} \qquad \overline{I} \in \mathcal{E} \qquad \mathcal{E} \oplus \mathit{extK}(I, \overline{I}, \overline{IS}) \vdash \mathcal{P}}{\mathcal{E} \vdash (\mathbf{interface}\ I\ \mathbf{extends}\ \overline{I}\ \{\overline{IS}\}) \cdot \mathcal{P}}\ (\textsc{NewInt})$$

$$\frac{\begin{array}{c} I \in \mathcal{E} \qquad C \notin \mathcal{E} \qquad \overline{D} \in \mathcal{E} \qquad \overline{E} = \mathit{commSup}_{\mathcal{E}}(C) \\ \mathcal{E} \oplus \mathit{extL}(C, \overline{D}, I, \overline{f}, \overline{M}) \vdash \\ \langle C : \ \mathit{anSpec}(\overline{MS}) \cdot \mathit{anReq}(\overline{M}) \cdot \mathit{supCls}(\overline{E}) \cdot \mathit{intSpec}(\mathit{public}_{\mathcal{E}}(I)) \rangle \cdot \mathcal{P} \end{array}}{\mathcal{E} \vdash (\mathbf{class}\ C\ \mathbf{extends}\ \overline{D}\ \mathbf{implements}\ I\ \{\overline{f}\ \overline{M}\ \overline{MS}\}) \cdot \mathcal{P}}\ (\textsc{NewClass}')$$

$$\frac{\mathcal{E} \vdash \mathcal{P}}{\mathcal{E} \vdash \langle C : \ \epsilon \rangle \cdot \mathcal{P}}\ (\textsc{EmpClass}) \qquad \frac{\mathcal{E} \vdash \langle C : \ \mathit{verify}(\mathit{bind}_{\mathcal{E}}(D, D\#m), m, (p, q)) \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \ \mathit{anSpec}(m@D(\overline{x}) : (p, q)) \rangle \cdot \mathcal{P}}\ (\textsc{NewSpec})$$

$$\frac{\mathcal{E} \vdash \langle C : \ \mathit{verify}(C, m, R_{\mathcal{E}}\!\uparrow\!(C.\mathit{inh}, m)) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \ \mathit{anReq}(m(\overline{x})\{t\}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{NewMtd})$$

$$\frac{O \vdash_{PL} \mathit{body}_{\mathcal{E}}(D, m) : (p, q) \qquad \mathcal{E} \oplus \mathit{extS}(C, D, m, (p, q)) \vdash \langle C : \ \mathit{anCalls}(D, O) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \ \mathit{verify}(D, m, (p, q)) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{ReqNotDer})$$

$$\frac{S_{\mathcal{E}}\!\uparrow\!(C, D.m) \twoheadrightarrow (p, q) \qquad \mathcal{E} \vdash \langle C : \ \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \ \mathit{verify}(D, m, (p, q)) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}\ (\textsc{ReqDer})$$

(Internal)

$$\frac{E = bind_{\mathcal{E}}(C, D\#m) \qquad \mathcal{E} \oplus extR(C, D, m, (p, q)) \vdash \langle C : \ verify(E, m, (r', s')) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \ anCalls(D, \{r\}\, v := m(\overline{e})\, \{s\}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(Static)

$$\frac{\mathcal{E} \vdash \langle C : \ verify(bind_{\mathcal{E}}(B, B\#m), m, (r', s')) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \ anCalls(D, \{r\}\, v := m@B(\overline{e})\, \{s\}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(External′)

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

(IntSpec)

$$\frac{S_{\mathcal{E}}\!\uparrow\!(C, m) \rightarrowtriangle spec_{\mathcal{E}}(C.\mathrm{impl}, m) \qquad \mathcal{E} \vdash \langle C : \ \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \ intSpec(m) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(SupMtd)

$$\frac{\mathcal{E} \vdash \langle C : \ supMtd(D, called(D) \setminus C.\mathrm{mtds}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \ supCls(D) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(SupReq)

$$\frac{E = bind_{\mathcal{E}}(C, D\#m) \qquad \mathcal{E} \vdash \langle C : \ verify(E, m, delReq_{\mathcal{E}}(C, D\#m)) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \ supMtd(D, m) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

B.2. Structural Rules

(noSpec)

$$\frac{\mathcal{E} \vdash \langle C : \ \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \ anSpec(\emptyset) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(decompSpec)

$$\frac{\mathcal{E} \vdash \langle C : \ anSpec(\overline{MS_1}) \cdot anSpec(\overline{MS_2}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \ anSpec(\overline{MS_1}\ \overline{MS_2}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(noMtds)

$$\frac{\mathcal{E} \vdash \langle C : \ \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \ anReq(\emptyset) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(decompMtds)

$$\frac{\mathcal{E} \vdash \langle C : \ anReq(\overline{M_1}) \cdot anReq(\overline{M_2}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \ anReq(\overline{M_1}\ \overline{M_2}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(noReq)
$$\frac{\mathcal{E} \vdash \langle C : \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \mathit{verify}(D, m, \emptyset) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(decompReq)
$$\frac{\mathcal{E} \vdash \langle C : \mathit{verify}(D, m, \overline{R_1}) \cdot \mathit{verify}(D, m, \overline{R_2}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \mathit{verify}(D, m, \overline{R_1}\ \overline{R_2}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(decompSeq)
$$\frac{\mathcal{E} \vdash \langle C : \mathit{anCalls}(D, t_1) \cdot \mathit{anCalls}(D, t_2) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \mathit{anCalls}(D, t_1; t_2) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(decompIf)
$$\frac{\mathcal{E} \vdash \langle C : \mathit{anCalls}(D, t_1) \cdot \mathit{anCalls}(D, t_2) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \mathit{anCalls}(D, \texttt{if}\ b\ \texttt{then}\ t_1\ \texttt{else}\ t_2\ \texttt{fi}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(skip)
$$\frac{\mathcal{E} \vdash \langle C : \mathcal{O} \rangle \cdot \mathcal{P} \qquad t \text{ does not contain call statements}}{\mathcal{E} \vdash \langle C : \mathit{anCalls}(D, t) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(nosupCls)
$$\frac{\mathcal{E} \vdash \langle C : \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \mathit{supCls}(\emptyset) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(decompSupCls)
$$\frac{\mathcal{E} \vdash \langle C : \mathit{supCls}(\overline{D_1}) \cdot \mathit{supCls}(\overline{D_2}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \mathit{supCls}(\overline{D_1}\ \overline{D_2}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(noSupMtd)
$$\frac{\mathcal{E} \vdash \langle C : \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \mathit{supMtd}(D, \emptyset) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(decompSupMtd)
$$\frac{\mathcal{E} \vdash \langle C : \mathit{supMtd}(D, \overline{m_1}) \cdot \mathit{supMtd}(D, \overline{m_2}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \mathit{supMtd}(D, \overline{m_1}\ \overline{m_2}) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

(noIntSpec)
$$\frac{\mathcal{E} \vdash \langle C : \mathcal{O} \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : \mathit{intSpec}(\emptyset) \cdot \mathcal{O} \rangle \cdot \mathcal{P}}$$

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

C. Soundness of *LBSI(PL)*

As for the basic calculus without interfaces in Section A, we show the proofs for two central properties, Lemma 2 (sound environments) and Theorem 3 (soundness of the calculus).

C.1. Proof of Lemma 2

This lemma corresponds to the analogous Lemma 1 (for the setting without interfaces), shown earlier.

> Assume a sound environment $\mathcal{E}$: *IEnv* and a sound program logic *PL*. Let B, D : *Cid*, m : *Mid*, and (p, q) : *APair* such that $B, D \in \mathcal{E}$ and $(p, q) \in S_{\mathcal{E}}{\uparrow}(D, B.m)$. Then $\models_D m(\overline{x})$: $(p, q)\{body_{\mathcal{E}}(B, m)\}$.

Proof. This proof is equal to the proof of Lemma 1, except for the treatment of external calls in the induction step.

By definition of $S{\uparrow}$, the assumption $(p, q) \in S_{\mathcal{E}}{\uparrow}(D, B.m)$ implies that there exists some class C such that $D \leq_{\mathcal{E}} C \leq_{\mathcal{E}} B$ and $(p, q) \in S_{\mathcal{E}}(C, B.m)$. By Definition 6 there must be a proof outline O for the method body such that $O \vdash_{PL} body_{\mathcal{E}}(B, m) : (p, q)$. The proof of the lemma proceeds by induction over the call structure of m.

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

Induction step: Each kind of method call is considered by a separate case. If a call to some method n in $body(B, m)$ is bound to a definition in class H for search class G, we take $\models_G n(\overline{y}) : (t, u)\{body_{\mathcal{E}}(H, n)\}$ as the induction hypothesis, for each $(t, u) \in S_{\mathcal{E}}{\uparrow}(G, H.n)$. For a call to n with precondition r, postcondition s, and actual parameters $\overline{e}$, we let $r' = (r[\overline{z}_0/\overline{z}] \wedge \overline{y} = \overline{e}[\overline{z}_0/\overline{z}])$ and $s' = s[\mathtt{return}/v][\overline{z}_0/\overline{z}]$, where the return value of the method is assigned to the variable v by the caller. We consider each kind of method call in O by itself.

Late bound internal calls $\{r\}\, v := n(\overline{e})\, \{s\}$. By lemma assumptions, internal calls are bound with search class D. Thus, in the induction hypothesis we thereby have $G = D$, and $H = bind_{\mathcal{E}}(D, B\#n)$. Consequently, it suffices to ensure $S_{\mathcal{E}}{\uparrow}(D, H.n) \rightarrowtriangle (r', s')$, which follows by Definition 6.

Static calls $\{r\}\, v := n@A(\overline{e})\, \{s\}$. For the induction hypothesis, we have $G = D$ and $H = bind_{\mathcal{E}}(A, A\#n)$. By Definition 6, we have $S_{\mathcal{E}}{\uparrow}(C, H.n) \rightarrowtriangle$

(r', s'). The desired $S_{\mathcal{E}}\!\uparrow\!(D, H.n) \twoheadrightarrow (r', s')$ then follows by the definition of $S\!\uparrow$ since $D \leq_{\mathcal{E}} C$, i.e., $S_{\mathcal{E}}\!\uparrow\!(C, H.n) \subseteq S_{\mathcal{E}}\!\uparrow\!(D, H.n)$.

External calls $\{r\}\, v := e : I.n(\overline{e})\, \{s\}$. By Definition 6, Condition 1, we have $spec_{\mathcal{E}}(I, n) \twoheadrightarrow (r', s')$. Consider some class E, with $E.impl = J$. If it is possible for the call to bind in context E, then $n \in public_{\mathcal{E}}(J)$ and $J \preceq_{\mathcal{E}} I$, which gives $spec_{\mathcal{E}}(J, n) \twoheadrightarrow spec_{\mathcal{E}}(I, n)$. For the induction hypothesis, we then have $G = E$ and $bind_{\mathcal{E}}(E, E\#n) = H$, and the induction hypothesis ensures that for all $(t, u) \in S_{\mathcal{E}}\!\uparrow\!(E, H.n)$ we have $\models_E n(\overline{y}) : (t, u)\{body_{\mathcal{E}}(H, n)\}$. By Definition 6, Condition 3, we have $S_{\mathcal{E}}\!\uparrow\!(E, H.n) \twoheadrightarrow spec_{\mathcal{E}}(J, n)$. Soundness then follows by the induction hypothesis and transitivity of entailment: $S_{\mathcal{E}}\!\uparrow\!(E, H.n) \twoheadrightarrow spec_{\mathcal{E}}(J, n) \twoheadrightarrow spec_{\mathcal{E}}(I, n) \twoheadrightarrow (r', s')$. □

C.2. Proof of Theorem 3

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

Proof. Soundness is trivially maintained if $\mathcal{E}$ is extended by a new interface, and interfaces are assumed to be contained in the environment before they are used. For the analysis of some class C, we consider each condition of Definition 6 by itself. For Conditions 1 and 2, these proofs are similar to the ones in Appendix A.4. For brevity, we therefore refer to Appendix A.4 whenever some part of the proof is unchanged.

Condition 1 of Definition 6. The proof goes by induction over the inference rules. The only rule that extends $S_{\mathcal{E}}(C, B.m)$ is ReqNotDer, and this rule ensures that there is a proof outline for $body_{\mathcal{E}}(B, m)$. The set $S_{\mathcal{E}}(C, B.m)$ is only extended during analysis of C. Thus, for any $(p, q) \in S_{\mathcal{E}}(C, B.m)$ we have an O such that $O \vdash_{PL} body_{\mathcal{E}}(B, m) : (p, q)$ and $C \leq_{\mathcal{E}} B$.

If the specification (p, q) is included in $S_{\mathcal{E}}(C, B.m)$ by ReqNotDer, and operation $anCalls(B, O)$ is generated and analyzed in the context of C. Each call statement in the proof outline is analyzed by either Internal, Static or External′. We consider each kind of method call in isolation. For a call to n with precondition r, postcondition s, and actual parameters $\overline{e}$, we let $r' = (r[\overline{z}_0/\overline{z}] \wedge \overline{y} = \overline{e}[\overline{z}_0/\overline{z}])$ and $s' = s[\mathbf{return}/v][\overline{z}_0/\overline{z}]$, where $\overline{y}$ are the formal parameters and the return value of the method is assigned to the variable v by the caller.

Late bound internal calls. The proof of this case is unchanged from Appendix A.4.

Static calls. The proof of this case is unchanged from Appendix A.4.

External calls. For each $\{r\}\, v := e : I.n(\overline{e})\, \{s\}$ in O, External′ will be applied. As the rule succeeds, the rule directly ensures $spec_{\mathcal{E}}(I, n) \rightarrowtriangle (r', s')$ as required by Definition 6.

Condition 2 of Definition 6. The proof of this case is unchanged from Appendix A.4.

Condition 3 of Definition 6. For class C and $I = C.impl$, we must ensure that for each $m \in public_{\mathcal{E}}(I)$ we have $S_{\mathcal{E}}{\uparrow}(C, m) \rightarrowtriangle spec_{\mathcal{E}}(I, m)$, where $S_{\mathcal{E}}{\uparrow}(C, m)$ denotes the set $S_{\mathcal{E}}{\uparrow}(C, bind_{\mathcal{E}}(C, C\#m).m)$.

When class C is selected for analysis by rule NewClass′, an operation $intSpec(public_{\mathcal{E}}(I))$ is generated and analyzed in the context of C. For each $m \in public_{\mathcal{E}}(I)$, decomposition leads to an operation $intSpec(m)$. The analysis of this operation succeeds by application of IntSpec (Appendix B), which ensures the required relation $S_{\mathcal{E}}{\uparrow}(C, m) \rightarrowtriangle spec_{\mathcal{E}}(I, m)$ □